

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению лабораторных работ
по дисциплине «Микросервисные технологии» для студентов направления
подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Инженерия сложных программных систем»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	3
Лабораторная работа №1. API Gateway на Spring Cloud Gateway.....	4
Лабораторная работа №2. Service Discovery с Consul.....	17
Лабораторная работа №3. Балансировка нагрузки и Health Checks.....	27
Лабораторная работа №4. Circuit Breaker (Resilience4j).....	37
Лабораторная работа №5. Retry и Timeout.....	46
Лабораторная работа №6. RabbitMQ: Producer и Consumer.....	55
Лабораторная работа №7. Kafka: Producer и Consumer.....	65
Лабораторная работа №8. Сравнение RabbitMQ и Kafka.....	75
Лабораторная работа №9. gRPC сервер и клиент (unary).....	84
Лабораторная работа №10. gRPC streaming (bidirectional).....	93
Лабораторная работа №11. Saga (хореография) через Kafka.....	103
Лабораторная работа №12. Saga (оркестрация) с координатором + JWT безопасность.....	112
Лабораторная работа №13. Dapr: Service Invocation и State Management.....	122
Лабораторная работа №14. Dapr: Pub/Sub (абстракция над Kafka/RabbitMQ).....	131

ВВЕДЕНИЕ

Современные программные системы всё чаще проектируются и разрабатываются в виде набора слабо связанных, независимо развёртываемых сервисов. Микросервисная архитектура позволяет масштабировать отдельные части системы, ускоряет вывод новой функциональности и повышает устойчивость приложения к отказам. Вместе с тем переход к микросервисам порождает целый класс новых задач: маршрутизацию внешних запросов, обнаружение и балансировку экземпляров сервисов, обеспечение отказоустойчивости при сетевом взаимодействии, организацию асинхронного обмена сообщениями и согласование распределённых бизнес-операций.

Настоящие методические указания предназначены для студентов направления подготовки 09.03.04 «Программная инженерия», направленность (профиль) «Разработка и сопровождение программного обеспечения», и содержат материалы для выполнения цикла из четырнадцати лабораторных работ по дисциплине «Микросервисные технологии». Цель практикума — сформировать у обучающихся практические навыки проектирования, реализации и эксплуатации микросервисных систем с использованием современных инструментов и шаблонов.

Тематика лабораторных работ выстроена по нарастанию сложности и охватывает основные группы технологий микросервисной архитектуры: единая точка входа и обнаружение сервисов (API Gateway, Service Discovery, балансировка нагрузки и контроль работоспособности); шаблоны устойчивости при межсервисном взаимодействии (Circuit Breaker, Retry, Timeout); асинхронный обмен сообщениями (RabbitMQ, Apache Kafka и их сравнение); высокопроизводительное взаимодействие по протоколу gRPC, включая потоковую передачу; управление распределёнными транзакциями по шаблону Saga в хореографическом и оркестрационном вариантах с применением механизмов безопасности; а также использование среды Dapr для вызова сервисов, управления состоянием и публикации-подписки.

Каждая лабораторная работа имеет единую структуру: формулируется цель, приводится теоретическое обоснование, описываются требования к рабочему окружению и порядок выполнения, предлагаются контрольные вопросы и требования к содержанию отчёта. Выполнение работы предполагает не только повторение приведённого примера, но и адаптацию

решения под индивидуальный вариант, а также подтверждение результата скриншотами, журналами работы и ответами сервисов.

Результатом освоения практикума является способность самостоятельно проектировать и реализовывать взаимодействие микросервисов, обоснованно выбирать средства обмена данными и шаблоны обеспечения отказоустойчивости, а также анализировать поведение распределённой системы в штатных и нештатных ситуациях.

ЛАБОРАТОРНАЯ РАБОТА № 1

Цель работы: реализовать единую точку входа для двух микросервисов Product Service и Order Service, настроить маршрутизацию /products/ и /orders/, добавить Gateway-фильтр AddRequestHeader и проверить прохождение запросов через шлюз.

Теоретическое обоснование

1. Микросервисная система состоит из набора автономных сервисов, каждый из которых имеет собственную ответственность и может развёртываться независимо. Прямое обращение клиента к каждому сервису быстро приводит к росту связанности: клиент должен знать множество адресов, портов, форматов ошибок и правил безопасности.

2. API Gateway используется как единая точка входа. Он принимает внешний запрос, сопоставляет его с маршрутом, применяет предикаты и фильтры, после чего передаёт запрос во внутренний сервис. В учебной работе Gateway решает три задачи: маршрутизацию, добавление технического заголовка и изоляцию внутренних портов сервисов от клиента.

3. Spring Cloud Gateway строится вокруг понятия Route. Route содержит идентификатор, URI целевого сервиса, набор Predicate для выбора подходящего маршрута и набор Filter для изменения запроса или ответа. Predicate Path=/products/** означает, что маршрут подходит только для запросов с указанным шаблоном пути.

4. Фильтр AddRequestHeader удобен для демонстрации того, что запрос прошёл через Gateway. Шлюз добавляет, например, X-Gateway-Source: spring-cloud-gateway, а внутренний сервис выводит этот заголовок в ответе или логах.

5. Результатом лабораторной работы считается не только запуск трёх Spring Boot приложений, но и понимание границ ответственности: Gateway

не реализует бизнес-логику каталога или заказов, а управляет сетевым входом в систему.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Product Service	REST-сервис каталога товаров, порт 8081, endpoint /products
Order Service	REST-сервис заказов, порт 8082, endpoint /orders
API Gateway	Spring Cloud Gateway, порт 8080, маршруты к двум сервисам
AddRequestHeader	Фильтр, добавляющий служебный заголовок к запросу

Требования к рабочему окружению

Перед выполнением лабораторной работы необходимо подготовить рабочее место. Все сервисы рекомендуется запускать локально в отдельных терминалах или из IDE. Контейнерные зависимости запускаются через Docker. Версии инструментов фиксируются в отчёте, потому что различия версий Spring Boot, Spring Cloud и брокеров сообщений могут менять команды запуска и формат конфигурации.

Таблица 2 – Базовый состав рабочего окружения

Инструмент	Назначение
JDK 21 или 17	Запуск Spring Boot сервисов, gRPC и клиентских приложений
Maven 3.9+	Сборка сервисов, генерация gRPC-кода, управление зависимостями
Docker Desktop / Docker Engine	Запуск Consul, RabbitMQ, Kafka, Keycloak, Redis и Dapr sidecar
curl / Postman	Проверка REST API, Gateway-маршрутов, Dapr API и JWT-сценариев
IDEA / VS Code	Редактирование Java-кода, YAML-конфигураций и

	proto-файлов
--	--------------

Схема взаимодействия компонентов

Схема лабораторной работы № 1



Рисунок 1 – Логическая схема выполнения лабораторной работы № 1

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Создание Product Service.

Создайте Spring Boot проект с зависимостью Spring Web. Сервис должен возвращать список товаров и показывать полученный от Gateway заголовок. Это позволяет отличить прямой запрос от запроса через шлюз.

Листинг 1 – Создание Product Service

```

@RestController
@RequestMapping("/products")
public class ProductController {
    @GetMapping
    public Map<String, Object> getProducts(
        @RequestHeader(value = "X-Gateway-Source",
            required = false) String source) {
        return Map.of(

```

```

        "service", "product-service",
        "gatewayHeader", source,
        "items", List.of("Laptop", "Mouse", "Keyboard")
    );
    }
}

# application.yml
server:
    port: 8081

```

Шаг 2. Создание Order Service.

Второй сервис реализуется аналогично, но отвечает за заказы. Наличие двух сервисов важно: Gateway должен различать маршруты по URI, а не просто проксировать всё на один адрес.

Листинг 2 – Создание Order Service

```

@RestController
@RequestMapping("/orders")
public class OrderController {
    @GetMapping
    public Map<String, Object> getOrders(
        @RequestHeader(value = "X-Gateway-Source",
required = false) String source) {
        return Map.of(
            "service", "order-service",
            "gatewayHeader", source,
            "orders", List.of("ORD-1001", "ORD-1002")
        );
    }
}

server:
    port: 8082

```

Шаг 3. Настройка Gateway.

В отдельном Spring Boot проекте добавьте зависимость `spring-cloud-starter-gateway`. В конфигурации укажите два маршрута: один для `/products/**`, второй для `/orders/**`. URI пока задаётся статически, так как Service Discovery будет изучаться в следующей работе.

Листинг 3 – Настройка Gateway

```
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: product-route
          uri: http://localhost:8081
          predicates:
            - Path=/products/**
          filters:
            - AddRequestHeader=X-Gateway-Source, spring-
cloud-gateway
        - id: order-route
          uri: http://localhost:8082
          predicates:
            - Path=/orders/**
          filters:
            - AddRequestHeader=X-Gateway-Source, spring-
cloud-gateway
```

Шаг 4. Проверка маршрутизации.

Запустите три приложения и выполните запросы только на порт Gateway. В ответе внутренних сервисов должен появиться gatewayHeader. Если обратиться напрямую к localhost:8081/products, заголовок будет null.

Листинг 4 – Проверка маршрутизации

```
curl http://localhost:8080/products
curl http://localhost:8080/orders
curl http://localhost:8081/products

# ожидаем: через Gateway gatewayHeader=spring-cloud-gateway,
напрямую - null
```

Шаг 5. Анализ результата.

В отчёте объясните, почему клиенту теперь достаточно знать только адрес Gateway. Отдельно укажите недостаток статических URI: при смене порта сервиса конфигурацию Gateway придётся менять вручную.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.
7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Создать микросервис Product Service для предметной области «каталог товаров» с endpoint /products. 2) Создать связанный сервис Product Service Admin с endpoint /products-admin. 3) Настроить Gateway: /products/** и /products-admin/**, добавить заголовок X-Student-Variant=1. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.

2	1) Создать микросервис Order Service для предметной области «оформление заказов» с endpoint /orders. 2) Создать связанный сервис Order Service Admin с endpoint /orders-admin. 3) Настроить Gateway: /orders/** и /orders-admin/**, добавить заголовок X-Student-Variant=2. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
3	1) Создать микросервис User Service для предметной области «пользователи» с endpoint /users. 2) Создать связанный сервис User Service Admin с endpoint /users-admin. 3) Настроить Gateway: /users/** и /users-admin/**, добавить заголовок X-Student-Variant=3. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
4	1) Создать микросервис Payment Service для предметной области «оплата» с endpoint /payments. 2) Создать связанный сервис Payment Service Admin с endpoint /payments-admin. 3) Настроить Gateway: /payments/** и /payments-admin/**, добавить заголовок X-Student-Variant=4. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
5	1) Создать микросервис Inventory Service для предметной области «складские остатки» с endpoint /inventory. 2) Создать связанный сервис Inventory Service Admin с endpoint /inventory-admin. 3) Настроить Gateway: /inventory/** и /inventory-admin/**, добавить заголовок X-Student-Variant=5. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
6	1) Создать микросервис Delivery Service для предметной области «доставка» с endpoint /delivery. 2) Создать связанный сервис Delivery Service Admin с endpoint /delivery-admin. 3) Настроить Gateway: /delivery/** и /delivery-admin/**, добавить заголовок X-Student-Variant=6. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
7	1) Создать микросервис Notification Service для предметной области «уведомления» с endpoint /notifications. 2) Создать связанный сервис Notification Service Admin с endpoint /notifications-admin. 3)

	Настроить Gateway: /notifications/** и /notifications-admin/**, добавить заголовок X-Student-Variant=7. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
8	1) Создать микросервис Discount Service для предметной области «скидки и промокоды» с endpoint /discounts. 2) Создать связанный сервис Discount Service Admin с endpoint /discounts-admin. 3) Настроить Gateway: /discounts/** и /discounts-admin/**, добавить заголовок X-Student-Variant=8. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
9	1) Создать микросервис Analytics Service для предметной области «аналитика продаж» с endpoint /analytics. 2) Создать связанный сервис Analytics Service Admin с endpoint /analytics-admin. 3) Настроить Gateway: /analytics/** и /analytics-admin/**, добавить заголовок X-Student-Variant=9. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
10	1) Создать микросервис Review Service для предметной области «отзывы покупателей» с endpoint /reviews. 2) Создать связанный сервис Review Service Admin с endpoint /reviews-admin. 3) Настроить Gateway: /reviews/** и /reviews-admin/**, добавить заголовок X-Student-Variant=10. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
11	1) Создать микросервис Cart Service для предметной области «корзина» с endpoint /cart. 2) Создать связанный сервис Cart Service Admin с endpoint /cart-admin. 3) Настроить Gateway: /cart/** и /cart-admin/**, добавить заголовок X-Student-Variant=11. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
12	1) Создать микросервис Return Service для предметной области «возвраты» с endpoint /returns. 2) Создать связанный сервис Return Service Admin с endpoint /returns-admin. 3) Настроить Gateway: /returns/** и /returns-admin/**, добавить заголовок X-Student-Variant=12. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса

	пользователем не используется.
13	1) Создать микросервис Billing Service для предметной области «биллинг подписок» с endpoint /billing. 2) Создать связанный сервис Billing Service Admin с endpoint /billing-admin. 3) Настроить Gateway: /billing/** и /billing-admin/**, добавить заголовок X-Student-Variant=13. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.
14	1) Создать микросервис Support Service для предметной области «поддержка клиентов» с endpoint /support. 2) Создать связанный сервис Support Service Admin с endpoint /support-admin. 3) Настроить Gateway: /support/** и /support-admin/**, добавить заголовок X-Student-Variant=14. 4) Выполнить не менее 6 запросов через Gateway и показать, что прямой порт сервиса пользователем не используется.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Зачем в микросервисной системе нужен API Gateway?
2. Чем отличаются Route, Predicate и Filter в Spring Cloud Gateway?
3. Почему статические URL сервисов неудобны при масштабировании?
4. Какие задачи не следует переносить в Gateway?
5. Как доказать, что запрос действительно прошёл через Gateway?

ЛАБОРАТОРНАЯ РАБОТА № 2

Цель работы: настроить динамическое обнаружение Product Service и Order Service через Consul, убрать жёстко заданные URL из Gateway и проверить работоспособность после перезапуска сервиса на новом порту.

Теоретическое обоснование

В микросервисной архитектуре экземпляры сервисов могут запускаться на разных узлах и портах. Если Gateway хранит статический адрес, то при масштабировании или перезапуске приложение становится хрупким.

Service Discovery решает эту проблему через реестр сервисов. Сервис при старте регистрирует своё имя, адрес, порт и параметры health check. Клиент или Gateway обращается к реестру и получает актуальный список доступных экземпляров.

Consul предоставляет каталог сервисов, механизм health checks и веб-интерфейс. В связке Spring Cloud Consul приложение регистрируется автоматически, если указаны зависимости и параметры spring.cloud.consul.discovery.

URI вида lb://product-service означает, что Gateway использует Spring Cloud LoadBalancer и DiscoveryClient. Он не знает конкретный порт, а работает с логическим именем сервиса.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Consul	Реестр сервисов и health checks, порт 8500
DiscoveryClient	Компонент Spring Cloud для получения адресов сервисов из Consul
spring.application.name	Логическое имя сервиса в реестре
lb://	Схема URI для балансируемого обращения по имени сервиса

Требования к рабочему окружению

Перед выполнением лабораторной работы необходимо подготовить рабочее место. Все сервисы рекомендуется запускать локально в отдельных терминалах или из IDE. Контейнерные зависимости запускаются через Docker. Версии инструментов фиксируются в отчёте, потому что различия версий Spring Boot, Spring Cloud и брокеров сообщений могут менять команды запуска и формат конфигурации.

Таблица 2 – Базовый состав рабочего окружения

Инструмент	Назначение
JDK 21 или 17	Запуск Spring Boot сервисов, gRPC и клиентских приложений
Maven 3.9+	Сборка сервисов, генерация gRPC-кода, управление зависимостями
Docker Desktop / Docker Engine	Запуск Consul, RabbitMQ, Kafka, Keycloak, Redis и Dapr sidecar
curl / Postman	Проверка REST API, Gateway-маршрутов, Dapr API и JWT-сценариев
IDEA / VS Code	Редактирование Java-кода, YAML-конфигураций и proto-файлов

Схема взаимодействия компонентов

Схема лабораторной работы № 2

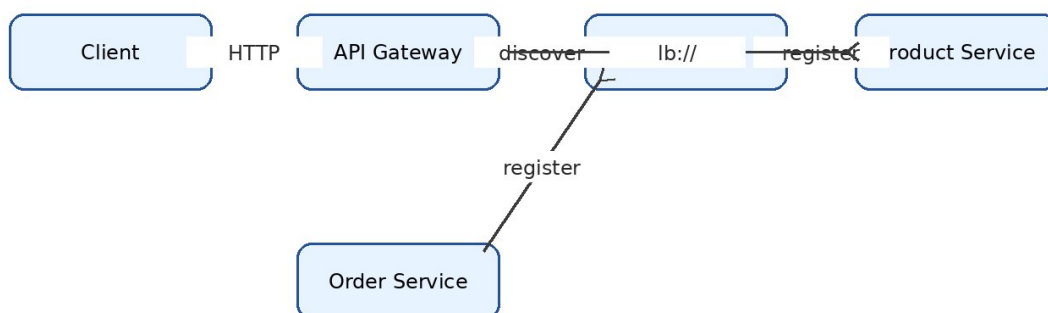


Рисунок 1 – Логическая схема выполнения лабораторной работы № 2

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Запуск Consul.

Consul удобно запустить в Docker. После старта веб-интерфейс доступен на <http://localhost:8500>. На этом этапе список сервисов будет пустым.

Листинг 1 – Запуск Consul

```
docker run --rm --name consul-dev \
  -p 8500:8500 -p 8600:8600/udp \
  hashicorp/consul:1.20 agent -dev -client=0.0.0.0
```

Шаг 2. Добавление зависимостей в сервисы.

В Product Service и Order Service добавьте Spring Cloud Consul Discovery и Actuator. Actuator нужен для endpoint `/actuator/health`, который Consul будет вызывать как проверку работоспособности.

Листинг 2 – Добавление зависимостей в сервисы

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-
discovery</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Шаг 3. Регистрация Product Service.

Настройте имя `product-service` и порт `0`. При `server.port=0` Spring Boot выбирает свободный порт, а Consul регистрирует фактический порт экземпляра.

Листинг 3 – Регистрация Product Service

```
spring:
  application:
    name: product-service
  cloud:
    consul:
      host: localhost
      port: 8500
    discovery:
      health-check-path: /actuator/health
      health-check-interval: 10s
  server:
    port: 0
  management:
    endpoints:
      web:
        exposure:
          include: health,info
```

Шаг 4. Перевод Gateway на discovery.

В Gateway больше не указываются `http://localhost:8081` и `http://localhost:8082`. Вместо них используются логические имена сервисов.

Листинг 4 – Перевод Gateway на discovery

```
spring:
  cloud:
    gateway:
      routes:
        - id: product-route
          uri: lb://product-service
          predicates:
            - Path=/products/**
        - id: order-route
          uri: lb://order-service
          predicates:
            - Path=/orders/**
```

Шаг 5. Проверка динамического обнаружения.

Запустите сервисы, откройте Consul UI и убедитесь, что оба сервиса имеют статус `passing`. Затем перезапустите Product Service: порт изменится, но запрос через Gateway должен продолжить работать.

Листинг 5 – Проверка динамического обнаружения

```
curl http://localhost:8080/products  
# перезапустить Product Service  
curl http://localhost:8080/products  
# конфигурация Gateway не менялась
```

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.
7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Зарегистрировать Product Service в Consul под именем products-service. 2) Настроить Gateway на lb://products-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 6s и приложить скрин Consul UI.
2	1) Зарегистрировать Order Service в Consul под именем orders-service. 2) Настроить Gateway на lb://orders-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 7s и приложить скрин Consul UI.
3	1) Зарегистрировать User Service в Consul под именем users-service. 2) Настроить Gateway на lb://users-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 8s и приложить скрин Consul UI.
4	1) Зарегистрировать Payment Service в Consul под именем payments-service. 2) Настроить Gateway на lb://payments-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 9s и приложить скрин Consul UI.
5	1) Зарегистрировать Inventory Service в Consul под именем inventory-service. 2) Настроить Gateway на lb://inventory-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 5s и приложить скрин Consul UI.
6	1) Зарегистрировать Delivery Service в Consul под именем delivery-service. 2) Настроить Gateway на lb://delivery-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 6s и приложить скрин Consul UI.
7	1) Зарегистрировать Notification Service в Consul под именем notifications-service. 2) Настроить Gateway на lb://notifications-service. 3) Запускать сервис с server.port=0 и показывать смену порта после перезапуска. 4) Настроить health check с интервалом 7s и приложить скрин Consul UI.
8	1) Зарегистрировать Discount Service в Consul под именем discounts-

	service. 2) Настроить Gateway на lb://discounts-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 8s и приложить скрин Consul UI.
9	1) Зарегистрировать Analytics Service в Consul под именем analytics-service. 2) Настроить Gateway на lb://analytics-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 9s и приложить скрин Consul UI.
10	1) Зарегистрировать Review Service в Consul под именем reviews-service. 2) Настроить Gateway на lb://reviews-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 5s и приложить скрин Consul UI.
11	1) Зарегистрировать Cart Service в Consul под именем cart-service. 2) Настроить Gateway на lb://cart-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 6s и приложить скрин Consul UI.
12	1) Зарегистрировать Return Service в Consul под именем returns-service. 2) Настроить Gateway на lb://returns-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 7s и приложить скрин Consul UI.
13	1) Зарегистрировать Billing Service в Consul под именем billing-service. 2) Настроить Gateway на lb://billing-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 8s и приложить скрин Consul UI.
14	1) Зарегистрировать Support Service в Consul под именем support-service. 2) Настроить Gateway на lb://support-service. 3) Запускать сервис с server.port=0 и показать смену порта после перезапуска. 4) Настроить health check с интервалом 9s и приложить скрин Consul UI.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия КОМПОНЕНТОВ;

- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Что такое Service Discovery и какую проблему он решает?
2. Как работает схема lb://service-name в Gateway?
3. Почему health check важен для Consul?
4. Зачем использовать server.port=0 при демонстрации discovery?
5. Что произойдёт при ошибке в имени сервиса?

ЛАБОРАТОРНАЯ РАБОТА № 3

Цель работы: запустить несколько экземпляров Product Service, настроить балансировку через Spring Cloud LoadBalancer и проверить автоматическое исключение неработоспособных экземпляров по health check.

Теоретическое обоснование

1. Балансировка нагрузки распределяет запросы между несколькими экземплярами одного сервиса. Для клиента все экземпляры выглядят как один логический сервис, например product-service.

2. В связке Spring Cloud Gateway + Consul балансировка обычно выполняется на стороне клиента. Gateway получает из Consul список healthy-экземпляров и выбирает один из них через LoadBalancer.

3. Health check нужен, чтобы не отправлять запросы на неработающий экземпляр. В Spring Boot стандартная точка проверки - /actuator/health. Если endpoint возвращает DOWN или недоступен, Consul помечает сервис как critical.

4. Для демонстрации балансировки удобно добавить в ответ сервиса номер порта и instanceId. Тогда серия запросов покажет, как Gateway переключается между экземплярами.

5. Важно понимать, что балансировка не делает сервис отказоустойчивым автоматически: состояние должно быть вынесено из памяти экземпляра или синхронизировано через БД/кэш.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Несколько экземпляров	Одинаковый spring.application.name, разные порты
LoadBalancer	Выбор конкретного healthy-инстанса
Actuator Health	Проверка состояния сервиса
instanceId	Идентификатор экземпляра для диагностики

Требования к рабочему окружению

Перед выполнением лабораторной работы необходимо подготовить рабочее место. Все сервисы рекомендуется запускать локально в отдельных терминалах или из IDE. Контейнерные зависимости запускаются через Docker. Версии инструментов фиксируются в отчёте, потому что различия версий Spring Boot, Spring Cloud и брокеров сообщений могут менять команды запуска и формат конфигурации.

Таблица 2 – Базовый состав рабочего окружения

Инструмент	Назначение
JDK 21 или 17	Запуск Spring Boot сервисов, gRPC и клиентских приложений
Maven 3.9+	Сборка сервисов, генерация gRPC-кода, управление зависимостями
Docker Desktop / Docker Engine	Запуск Consul, RabbitMQ, Kafka, Keycloak, Redis и Dapr sidecar
curl / Postman	Проверка REST API, Gateway-маршрутов, Dapr API и JWT-сценариев
IDEA / VS Code	Редактирование Java-кода, YAML-конфигураций и proto-файлов

Схема взаимодействия компонентов

Схема лабораторной работы № 3

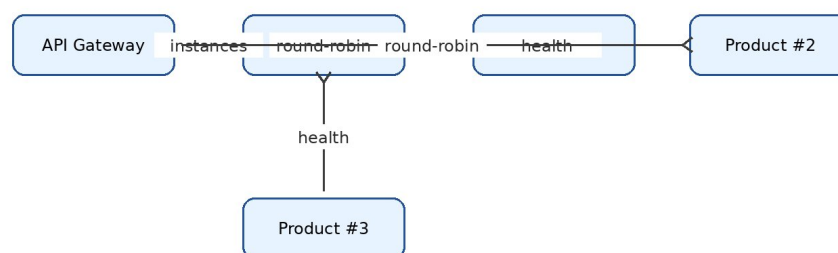


Рисунок 1 – Логическая схема выполнения лабораторной работы № 3

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Подготовка ответа с портом.

В Product Service добавьте вывод `local.server.port`. Это позволит увидеть, какой экземпляр обработал запрос.

Листинг 1 – Подготовка ответа с портом

```
@Value("${local.server.port}")
private int port;

@GetMapping("/products/instance")
public Map<String, Object> instance() {
    return Map.of(
        "service", "product-service",
        "port", port,
        "time", Instant.now().toString()
    );
}
```

Шаг 2. Запуск нескольких экземпляров.

Запустите один и тот же `jar` несколько раз с разными портами или с `server.port=0`. Все экземпляры должны иметь одинаковое имя `product-service`.

Листинг 2 – Запуск нескольких экземпляров

```
java -jar product-service.jar --server.port=8081
java -jar product-service.jar --server.port=8083
java -jar product-service.jar --server.port=8084
```

Шаг 3. Проверка в Consul.

Откройте `http://localhost:8500` и убедитесь, что у `product-service` несколько `passing`-инстансов. Если один `critical`, проверьте путь `/actuator/health` и порт приложения.

Листинг 3 – Проверка в Consul

```
curl http://localhost:8081/actuator/health
```

```
curl http://localhost:8083/actuator/health
```

Шаг 4. Серия запросов через Gateway.

Выполните цикл запросов. В ответах должны чередоваться разные порты. Если все ответы идут на один порт, проверьте, что Gateway использует lb://product-service.

Листинг 4 – Серия запросов через Gateway

```
for i in {1..20}; do  
  curl -s http://localhost:8080/products/instance; echo  
done
```

Шаг 5. Исключение неработающего инстанса.

Остановите один экземпляр и дождитесь, пока Consul изменит статус. Повторите серию запросов: остановленный порт больше не должен появляться в ответах.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Запустить 3 экземпляра Product Service с одним <code>spring.application.name=products-service</code> . 2) В ответе <code>/products</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
2	1) Запустить 4 экземпляра Order Service с одним <code>spring.application.name=orders-service</code> . 2) В ответе <code>/orders</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
3	1) Запустить 2 экземпляра User Service с одним <code>spring.application.name=users-service</code> . 2) В ответе <code>/users</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
4	1) Запустить 3 экземпляра Payment Service с одним <code>spring.application.name=payments-service</code> . 2) В ответе <code>/payments</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.

5	1) Запустить 4 экземпляра Inventory Service с одним <code>spring.application.name=inventory-service</code> . 2) В ответе <code>/inventory</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
6	1) Запустить 2 экземпляра Delivery Service с одним <code>spring.application.name=delivery-service</code> . 2) В ответе <code>/delivery</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
7	1) Запустить 3 экземпляра Notification Service с одним <code>spring.application.name=notifications-service</code> . 2) В ответе <code>/notifications</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
8	1) Запустить 4 экземпляра Discount Service с одним <code>spring.application.name=discounts-service</code> . 2) В ответе <code>/discounts</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
9	1) Запустить 2 экземпляра Analytics Service с одним <code>spring.application.name=analytics-service</code> . 2) В ответе <code>/analytics</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
10	1) Запустить 3 экземпляра Review Service с одним <code>spring.application.name=reviews-service</code> . 2) В ответе <code>/reviews</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
11	1) Запустить 4 экземпляра Cart Service с одним <code>spring.application.name=cart-service</code> . 2) В ответе <code>/cart</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по

	экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
12	1) Запустить 2 экземпляра Return Service с одним <code>spring.application.name=returns-service</code> . 2) В ответе <code>/returns</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
13	1) Запустить 3 экземпляра Billing Service с одним <code>spring.application.name=billing-service</code> . 2) В ответе <code>/billing</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.
14	1) Запустить 4 экземпляра Support Service с одним <code>spring.application.name=support-service</code> . 2) В ответе <code>/support</code> выводить <code>instanceId</code> и <code>port</code> . 3) Выполнить 20 запросов через Gateway и показать распределение по экземплярам. 4) Остановить один экземпляр и доказать, что трафик идёт только на healthy-инстансы.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;

— ответы на вопросы для защиты.

Вопросы для защиты работы

1. В чём отличие клиентской балансировки от серверной?
2. Почему все экземпляры должны иметь одинаковое имя сервиса?
3. Как Actuator влияет на исключение неработоспособного инстанса?
4. Почему health check не должен быть тяжёлым?
5. Как проверить round-robin распределение запросов?

ЛАБОРАТОРНАЯ РАБОТА № 4

Цель работы: защитить Gateway от отказа Product Service, настроить Circuit Breaker, реализовать fallback и показать переходы состояний при ошибках и таймаутах.

Теоретическое обоснование

В распределённой системе отказ одного сервиса не должен блокировать всю цепочку вызовов. Если Gateway продолжает отправлять запросы в неисправный Product Service, пользователи ждут таймауты, а потоковые ресурсы шлюза исчерпываются.

Circuit Breaker работает как автоматический предохранитель. В состоянии CLOSED запросы проходят. Если доля ошибок превышает порог, цепь переходит в OPEN и новые запросы сразу направляются в fallback. Через заданное время включается HALF_OPEN, где проверяется восстановление сервиса.

Resilience4j предоставляет CircuitBreaker, Retry, TimeLimiter и другие модули отказоустойчивости. В Gateway Circuit Breaker можно подключить как фильтр маршрута.

Fallback должен возвращать контролируемый ответ: кэшированные данные, сообщение о временной недоступности или ограниченный набор функций. Он не должен выдавать пользователю устаревшие критические данные без пометки.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Circuit Breaker	Предотвращает лавинообразные отказы
Sliding window	Окно статистики успешных и ошибочных вызовов
Failure threshold	Порог ошибок для открытия цепи
Fallback	Контролируемый ответ при недоступности сервиса

Схема взаимодействия компонентов

Схема лабораторной работы № 4



Рисунок 1 – Логическая схема выполнения лабораторной работы № 4

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Создание нестабильного endpoint.

В Product Service добавьте endpoint, который иногда отвечает ошибкой или задержкой. Это имитация реального сбоя: перегруженная БД, сеть или внешний API.

Листинг 1 – Создание нестабильного endpoint

```
private final AtomicInteger counter = new AtomicInteger();

@GetMapping("/products/unstable")
public Map<String, Object> unstable() throws
InterruptedException {
    int n = counter.incrementAndGet();
    if (n % 2 == 0) {
        Thread.sleep(3000);
        throw new
        ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
        "temporary failure");
    }
    return Map.of("status", "ok", "request", n);
}
```

Шаг 2. Подключение зависимостей Gateway.

В Gateway добавьте Spring Cloud Circuit Breaker с Resilience4j. Для реактивного Gateway используется соответствующий starter.

Листинг 2 – Подключение зависимостей Gateway

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-circuitbreaker-reactor-
resilience4j</artifactId>
</dependency>
```

Шаг 3. Настройка маршрута с CircuitBreaker.

Фильтр CircuitBreaker указывает имя цепи и fallbackUri. forward:/fallback/products означает, что fallback обрабатывается локальным контроллером Gateway.

Листинг 3 – Настройка маршрута с CircuitBreaker

```
spring:
  cloud:
    gateway:
      routes:
        - id: product-unstable
          uri: lb://product-service
          predicates:
            - Path=/products/unstable
          filters:
            - name: CircuitBreaker
              args:
                name: productCircuitBreaker
                fallbackUri: forward:/fallback/products

resilience4j:
  circuitbreaker:
    instances:
      productCircuitBreaker:
        slidingWindowSize: 10
        minimumNumberOfCalls: 5
        failureRateThreshold: 30
        waitDurationInOpenState: 10s
        permittedNumberOfCallsInHalfOpenState: 2
```

Шаг 4. Реализация fallback.

Fallback находится в Gateway. Он не вызывает Product Service, поэтому быстро возвращает предсказуемый ответ.

Листинг 4 – Реализация fallback

```
@RestController
public class FallbackController {
    @GetMapping("/fallback/products")
    public Mono<Map<String, Object>> productFallback() {
        return Mono.just(Map.of(
            "status", "fallback",
            "message", "Product Service временно недоступен"
        ));
    }
}
```

Шаг 5. Нагрузочная проверка.

Выполните 20 запросов и наблюдайте, что после накопления ошибок fallback начинает возвращаться сразу. В отчёте приведите фрагменты логов и объясните состояние цепи.

Листинг 5 – Нагрузочная проверка

```
for i in {1..20}; do
    curl -s http://localhost:8080/products/unstable; echo
    sleep 1
done
```

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.

6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Добавить в Product Service endpoint /products/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 30%. 3) Реализовать fallback для объекта «товар». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
2	1) Добавить в Order Service endpoint /orders/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 40%. 3) Реализовать fallback для объекта «заказ». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
3	1) Добавить в User Service endpoint /users/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 50%. 3) Реализовать fallback для объекта «пользователь». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
4	1) Добавить в Payment Service endpoint /payments/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 60%. 3) Реализовать fallback для объекта «платёж». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.

5	1) Добавить в Inventory Service endpoint /inventory/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 20%. 3) Реализовать fallback для объекта «остаток». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
6	1) Добавить в Delivery Service endpoint /delivery/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 30%. 3) Реализовать fallback для объекта «доставка». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
7	1) Добавить в Notification Service endpoint /notifications/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 40%. 3) Реализовать fallback для объекта «уведомление». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
8	1) Добавить в Discount Service endpoint /discounts/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 50%. 3) Реализовать fallback для объекта «промокод». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
9	1) Добавить в Analytics Service endpoint /analytics/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 60%. 3) Реализовать fallback для объекта «метрика». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
10	1) Добавить в Review Service endpoint /reviews/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 20%. 3) Реализовать fallback для объекта «отзыв». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
11	1) Добавить в Cart Service endpoint /cart/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 30%. 3) Реализовать fallback для объекта «позиция корзины». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.

12	1) Добавить в Return Service endpoint /returns/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 40%. 3) Реализовать fallback для объекта «возврат». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
13	1) Добавить в Billing Service endpoint /billing/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 50%. 3) Реализовать fallback для объекта «счёт». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.
14	1) Добавить в Support Service endpoint /support/unstable, который выдаёт ошибки и задержки. 2) Настроить Circuit Breaker с окном 10 запросов и порогом 60%. 3) Реализовать fallback для объекта «тикет». 4) Зафиксировать переход CLOSED → OPEN → HALF_OPEN в логах и метриках Actuator.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Что такое Circuit Breaker и от каких отказов он защищает?

2. Чем fallback отличается от retry?
3. Опишите состояния CLOSED, OPEN и HALF_OPEN.
4. Как подобрать sliding window size и failureRateThreshold?
5. Почему fallback не должен скрывать критическую ошибку данных?

ЛАБОРАТОРНАЯ РАБОТА № 5

Цель работы: настроить автоматические повторные попытки и таймауты для временных сбоев микросервиса, добавить exponential backoff и логирование попыток.

Теоретическое обоснование

Retry применяется при кратковременных сбоях: сетевой пакет потерян, сервис не успел ответить, временно недоступен внешний ресурс. Повтор не должен использоваться бездумно, особенно для операций изменения состояния.

Timeout задаёт верхнюю границу ожидания ответа. Без timeout клиентский поток может зависнуть надолго, а пользовательская операция будет выглядеть как бесконечная загрузка.

Exponential backoff увеличивает паузу между попытками и снижает давление на перегруженный сервис. Это особенно важно при массовых ошибках, когда множество клиентов одновременно начинают повторять запросы.

Retry и Circuit Breaker следует сочетать аккуратно. Если повторять каждый запрос много раз, можно быстрее открыть Circuit Breaker или, наоборот, перегрузить сервис повторными попытками.

В лабораторной работе нужно показать сценарий временного сбоя, где повтор помогает, и сценарий долгого ответа, где timeout прерывает ожидание.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Retry	Повтор временно неуспешного вызова
Backoff	Пауза между попытками
Timeout	Ограничение времени ожидания
Idempotency	Свойство операции безопасно повторяться

Схема взаимодействия компонентов

Схема лабораторной работы № 5



Рисунок 1 – Логическая схема выполнения лабораторной работы № 5

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Endpoint с временной ошибкой.

Добавьте endpoint, где первые две попытки завершаются ошибкой, а третья успешна. Это позволит увидеть пользу Retry.

Листинг 1 – Endpoint с временной ошибкой

```
private final AtomicInteger retryCounter = new
AtomicInteger();

@GetMapping("/products/retry-demo")
public Map<String, Object> retryDemo() {
    int attempt = retryCounter.incrementAndGet();
    if (attempt % 3 != 0) {
        throw new
ResponseStatusException(HttpStatus.BAD_GATEWAY, "temporary
error");
    }
}
```

```
        return Map.of("status", "success", "attempt", attempt);
    }
```

Шаг 2. Endpoint с долгим ответом.

Для проверки timeout создайте endpoint, который спит дольше допустимого времени. В реальном проекте это может быть долгий SQL-запрос или зависший внешний API.

Листинг 2 – Endpoint с долгим ответом

```
@GetMapping("/products/slow")
public Map<String, Object> slow() throws
InterruptedException {
    Thread.sleep(5000);
    return Map.of("status", "too late");
}
```

Шаг 3. Настройка Retry в Gateway.

Фильтр Retry повторяет запрос при указанных статусах. Для лабораторной работы достаточно GET-запросов, потому что они должны быть идемпотентными.

Листинг 3 – Настройка Retry в Gateway

```
spring:
  cloud:
    gateway:
      routes:
        - id: retry-route
          uri: lb://product-service
          predicates:
            - Path=/products/retry-demo
          filters:
            - name: Retry
              args:
                retries: 3
                statuses: BAD_GATEWAY,SERVICE_UNAVAILABLE
                methods: GET
                backoff:
                  firstBackoff: 200ms
                  maxBackoff: 2s
                  factor: 2
```

Шаг 4. Настройка timeout.

Timeout можно задать на уровне HTTP-клиента Gateway или через metadata маршрута. Главное - зафиксировать, что ответ после 5 секунд не ждётся бесконечно.

Листинг 4 – Настройка timeout

```
spring:
  cloud:
    gateway:
      httpclient:
        response-timeout: 2s
        connect-timeout: 1000
```

Шаг 5. Анализ логов.

Добавьте логирование на стороне Product Service и Gateway. В отчёте покажите номера попыток и объясните, почему повтор безопасен для GET, но опасен для операции создания заказа без idempotency key.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Для вызова Product Service настроить Retry на 3 попытки. 2) Установить timeout 2 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «товар».
2	1) Для вызова Order Service настроить Retry на 4 попытки. 2) Установить timeout 3 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «заказ».
3	1) Для вызова User Service настроить Retry на 2 попытки. 2) Установить timeout 1 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «пользователь».
4	1) Для вызова Payment Service настроить Retry на 3 попытки. 2) Установить timeout 2 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «платёж».
5	1) Для вызова Inventory Service настроить Retry на 4 попытки. 2) Установить timeout 3 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли

	повтор операции изменения сущности «остаток».
6	1) Для вызова Delivery Service настроить Retry на 2 попытки. 2) Установить timeout 1 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «доставка».
7	1) Для вызова Notification Service настроить Retry на 3 попытки. 2) Установить timeout 2 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «уведомление».
8	1) Для вызова Discount Service настроить Retry на 4 попытки. 2) Установить timeout 3 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «промокод».
9	1) Для вызова Analytics Service настроить Retry на 2 попытки. 2) Установить timeout 1 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «метрика».
10	1) Для вызова Review Service настроить Retry на 3 попытки. 2) Установить timeout 2 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «отзыв».
11	1) Для вызова Cart Service настроить Retry на 4 попытки. 2) Установить timeout 3 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «позиция корзины».
12	1) Для вызова Return Service настроить Retry на 2 попытки. 2) Установить timeout 1 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «возврат».
13	1) Для вызова Billing Service настроить Retry на 3 попытки. 2) Установить timeout 2 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «счёт».

14	1) Для вызова Support Service настроить Retry на 4 попытки. 2) Установить timeout 3 секунды. 3) Реализовать endpoint, где каждый третий запрос медленный или ошибочный. 4) В отчёте объяснить, допустим ли повтор операции изменения сущности «тикет».
----	--

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Когда Retry полезен, а когда вреден?
2. Почему Timeout должен быть меньше пользовательского SLA?
3. Что даёт exponential backoff?
4. Почему повтор POST-запроса может быть опасен?
5. Как Retry сочетается с Circuit Breaker?

ЛАБОРАТОРНАЯ РАБОТА № 6

Цель работы: реализовать асинхронную отправку события OrderCreatedEvent через RabbitMQ, создать Notification Service Consumer и проверить доставку сообщений через веб-интерфейс RabbitMQ.

Теоретическое обоснование

Асинхронная коммуникация снижает связанность микросервисов. Order Service не обязан знать адрес Notification Service: он публикует событие о факте создания заказа, а подписчик обрабатывает его независимо.

RabbitMQ реализует модель AMQP: producer отправляет сообщение в exchange, exchange по правилам binding передаёт сообщение в одну или несколько очередей, consumer читает очередь.

Для событий предметной области важно проектировать отдельный DTO, а не отправлять внутреннюю Entity. Событие должно содержать только необходимые поля: eventId, orderId, userId, amount, occurredAt.

Если consumer остановлен, очередь может накапливать сообщения. После запуска consumer получит накопленные события. Это демонстрирует слабую связанность по времени.

В лабораторной работе используется Notification Service, но в реальных системах подписчиков может быть много: email, SMS, аналитика, антифрод.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Producer	Order Service публикует OrderCreatedEvent
Exchange	Точка маршрутизации сообщений
Queue	Буфер сообщений для consumer
Consumer	Notification Service читает события и логирует уведомления

Схема взаимодействия компонентов

Схема лабораторной работы № 6



Рисунок 1 – Логическая схема выполнения лабораторной работы № 6

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Запуск RabbitMQ.

Запустите RabbitMQ с management UI. Веб-интерфейс позволит проверить exchange, queue, bindings и количество сообщений.

Листинг 1 – Запуск RabbitMQ

```
docker run --rm --name rabbitmq \
  -p 5672:5672 -p 15672:15672 \
  rabbitmq:3-management

# UI: http://localhost:15672, login/password: guest/guest
```

Шаг 2. Описание события.

Создайте DTO события. Поле `eventId` помогает обеспечить идемпотентность `consumer`, а `occurredAt` фиксирует момент возникновения бизнес-события.

Листинг 2 – Описание события

```
public record OrderCreatedEvent(  
    String eventId,  
    String orderId,  
    String userId,  
    BigDecimal amount,  
    Instant occurredAt  
) {}
```

Шаг 3. Конфигурация RabbitMQ.

В `Order Service` создайте `exchange`, `queue` и `binding`. Для простого сценария используется `topic exchange` и `routing key order.created`.

Листинг 3 – Конфигурация RabbitMQ

```
@Configuration  
public class RabbitConfig {  
    public static final String EXCHANGE = "orders.exchange";  
    public static final String QUEUE = "notifications.order-  
created";  
    public static final String ROUTING_KEY =  
"order.created";  
  
    @Bean TopicExchange exchange() { return new  
TopicExchange(EXCHANGE); }  
    @Bean Queue queue() { return  
QueueBuilder.durable(QUEUE).build(); }  
    @Bean Binding binding() {  
        return  
BindingBuilder.bind(queue()).to(exchange()).with(ROUTING_KEY);  
    }  
}
```

Шаг 4. Публикация события.

После создания заказа `Order Service` отправляет событие в `RabbitMQ`. Метод `createOrder` не вызывает `Notification Service` напрямую.

Листинг 4 – Публикация события

```
@Service
```

```

@RequiredArgsConstructor
public class OrderEventPublisher {
    private final RabbitTemplate rabbitTemplate;

    public void publishCreated(OrderCreatedEvent event) {
        rabbitTemplate.convertAndSend(
            RabbitConfig.EXCHANGE,
            RabbitConfig.ROUTING_KEY,
            event
        );
    }
}

```

Шаг 5. Consumer уведомлений.

Notification Service объявляет listener. В отчёте покажите лог consumer и состояние очереди в RabbitMQ UI.

Листинг 5 – Consumer уведомлений

```

@RabbitListener(queues = "notifications.order-created")
public void handle(OrderCreatedEvent event) {
    log.info("Send notification for order {}, amount={}",
        event.orderId(), event.amount());
}

```

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Определить событие ProductsCreatedEvent. 2) Создать exchange типа topic, очередь products.events и routing key products.created. 3) Product Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
2	1) Определить событие OrdersCreatedEvent. 2) Создать exchange типа fanout, очередь orders.events и routing key orders.created. 3) Order Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
3	1) Определить событие UsersCreatedEvent. 2) Создать exchange типа topic, очередь users.events и routing key users.created. 3) User Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
4	1) Определить событие PaymentsCreatedEvent. 2) Создать exchange типа direct, очередь payments.events и routing key payments.created. 3) Payment Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
5	1) Определить событие InventoryCreatedEvent. 2) Создать exchange типа

	topic, очередь inventory.events и routing key inventory.created. 3) Inventory Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
6	1) Определить событие DeliveryCreatedEvent. 2) Создать exchange типа fanout, очередь delivery.events и routing key delivery.created. 3) Delivery Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
7	1) Определить событие NotificationsCreatedEvent. 2) Создать exchange типа topic, очередь notifications.events и routing key notifications.created. 3) Notification Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
8	1) Определить событие DiscountsCreatedEvent. 2) Создать exchange типа direct, очередь discounts.events и routing key discounts.created. 3) Discount Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
9	1) Определить событие AnalyticsCreatedEvent. 2) Создать exchange типа topic, очередь analytics.events и routing key analytics.created. 3) Analytics Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
10	1) Определить событие ReviewsCreatedEvent. 2) Создать exchange типа fanout, очередь reviews.events и routing key reviews.created. 3) Review Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
11	1) Определить событие CartCreatedEvent. 2) Создать exchange типа topic, очередь cart.events и routing key cart.created. 3) Cart Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.

12	1) Определить событие ReturnsCreatedEvent. 2) Создать exchange типа direct, очередь returns.events и routing key returns.created. 3) Return Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
13	1) Определить событие BillingCreatedEvent. 2) Создать exchange типа topic, очередь billing.events и routing key billing.created. 3) Billing Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.
14	1) Определить событие SupportCreatedEvent. 2) Создать exchange типа fanout, очередь support.events и routing key support.created. 3) Support Service публикует событие, Consumer Service логирует его. 4) Остановить consumer, отправить 5 событий, запустить consumer и показать доставку накопленных сообщений.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Чем exchange отличается от queue в RabbitMQ?
2. Для чего нужен routing key?
3. Когда выбирать direct, topic и fanout exchange?
4. Что происходит с сообщениями при остановленном consumer?
5. Почему событие не должно быть копией JPA entity?

ЛАБОРАТОРНАЯ РАБОТА № 7

Цель работы: реализовать потоковую обработку событий ProductPriceChangedEvent через Apache Kafka, настроить producer, consumer group и показать replay событий.

Теоретическое обоснование

Kafka предназначена для потоковой обработки событий и долговременного хранения журнала сообщений. В отличие от классической очереди, события сохраняются в topic в течение заданного времени и могут быть перечитаны другой consumer group.

Topic делится на partitions. Ключ сообщения влияет на выбор partition, а значит и на порядок обработки сообщений с одинаковым ключом. Для изменения цены товара логично использовать productId как key.

Consumer group позволяет масштабировать обработку: разные consumer одной группы делят partitions между собой. Новый groupId может прочитать события заново с начала, если offset отсутствует.

Kafka удобна для аналитики, аудита, интеграции и event sourcing. Но она сложнее RabbitMQ и требует внимательной настройки topics, retention, сериализации и consumer groups.

В лабораторной работе Product Service публикует ProductPriceChangedEvent, а Analytics Service считает количество изменений и суммарную дельту цен.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Kafka broker	Хранение topic и partitions
Producer	Product Service отправляет события
Consumer group	Группа потребителей Analytics Service
Offset	Позиция чтения consumer group

Схема взаимодействия компонентов

Схема лабораторной работы № 7



Рисунок 1 – Логическая схема выполнения лабораторной работы № 7

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Запуск Kafka.

Для локальной работы можно использовать официальный образ Kafka в KRaft-режиме. В отчёте укажите используемый образ и параметры запуска.

Листинг 1 – Запуск Kafka

```
docker run --rm --name kafka -p 9092:9092 apache/kafka:4.0.0
```

Шаг 2. Создание topic.

Создайте topic product.price-changed. Количество partitions влияет на параллелизм чтения.

Листинг 2 – Создание topic

```
docker exec -it kafka /opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --create --topic product.price-changed \
  --partitions 3 --replication-factor 1
```

Шаг 3. Описание события.

Событие содержит старую и новую цену. По ним Analytics Service сможет посчитать величину изменения.

Листинг 3 – Описание события

```
public record ProductPriceChangedEvent(  
    String eventId,  
    String productId,  
    BigDecimal oldPrice,  
    BigDecimal newPrice,  
    Instant occurredAt  
) {}
```

Шаг 4. Producer в Product Service.

KafkaTemplate отправляет событие в topic. Ключом выбирается productId, чтобы события одного товара попадали в одну partition и сохраняли порядок.

Листинг 4 – Producer в Product Service

```
@Service  
@RequiredArgsConstructor  
public class ProductEventProducer {  
    private final KafkaTemplate<String,  
ProductPriceChangedEvent> kafkaTemplate;  
  
    public void priceChanged(ProductPriceChangedEvent event)  
    {  
        kafkaTemplate.send("product.price-changed",  
event.productId(), event);  
    }  
}
```

Шаг 5. Consumer в Analytics Service.

Consumer group analytics-group читает события и обновляет агрегаты в памяти. Для отчёта достаточно вывести счётчики в лог.

Листинг 5 – Consumer в Analytics Service

```
@KafkaListener(topics = "product.price-changed", groupId =  
"analytics-group")  
public void handle(ProductPriceChangedEvent event) {  
    BigDecimal delta =
```

```
event.newPrice().subtract(event.oldPrice());  
    log.info("product={}    delta={}",    event.productId(),  
    delta);  
}
```

Шаг 6. Проверка replay.

Создайте consumer с новым groupId, например analytics-replay. Он прочитает события с начала, если auto-offset-reset=earliest.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.
7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Создать Kafka topic products.events с 2 partition. 2) Публиковать событие об изменении объекта «товар» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
2	1) Создать Kafka topic orders.events с 3 partition. 2) Публиковать событие об изменении объекта «заказ» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
3	1) Создать Kafka topic users.events с 4 partition. 2) Публиковать событие об изменении объекта «пользователь» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
4	1) Создать Kafka topic payments.events с 1 partition. 2) Публиковать событие об изменении объекта «платёж» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
5	1) Создать Kafka topic inventory.events с 2 partition. 2) Публиковать событие об изменении объекта «остаток» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
6	1) Создать Kafka topic delivery.events с 3 partition. 2) Публиковать событие об изменении объекта «доставка» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
7	1) Создать Kafka topic notifications.events с 4 partition. 2) Публиковать событие об изменении объекта «уведомление» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.

8	1) Создать Kafka topic discounts.events с 1 partition. 2) Публиковать событие об изменении объекта «промокод» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
9	1) Создать Kafka topic analytics.events с 2 partition. 2) Публиковать событие об изменении объекта «метрика» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
10	1) Создать Kafka topic reviews.events с 3 partition. 2) Публиковать событие об изменении объекта «отзыв» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
11	1) Создать Kafka topic cart.events с 4 partition. 2) Публиковать событие об изменении объекта «позиция корзины» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
12	1) Создать Kafka topic returns.events с 1 partition. 2) Публиковать событие об изменении объекта «возврат» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
13	1) Создать Kafka topic billing.events с 2 partition. 2) Публиковать событие об изменении объекта «счёт» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.
14	1) Создать Kafka topic support.events с 3 partition. 2) Публиковать событие об изменении объекта «тикет» с key=id. 3) Analytics consumer group агрегирует количество событий и последнюю версию. 4) Создать второй groupId и показать replay с начала топика.

Содержание отчёта

— тема и цель лабораторной работы;

- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Критерии оценки

Таблица 4 – Критерии проверки работы

Критерий	Пояснение
Корректность реализации	Сервисы запускаются, взаимодействуют по требуемому механизму, индивидуальный вариант выполнен.
Доказательность отчёта	Приведены команды, логи, ответы запросов, скриншоты и объяснение результата.
Понимание архитектуры	Студент может объяснить, зачем используется изучаемый паттерн или инструмент и какие проблемы он решает.
Качество кода и конфигурации	Имена сервисов, endpoint-ов, topics, queues и ролей согласованы; нет случайных портов без объяснения.
Анализ ограничений	В отчёте указаны риски, ограничения и сценарии, где решение применять не следует.

Вопросы для защиты работы

1. Что такое topic, partition и offset в Kafka?
2. Как consumer group влияет на параллельное чтение?
3. Почему key события влияет на порядок обработки?
4. Что такое replay и зачем он нужен?
5. В каких сценариях Kafka предпочтительнее RabbitMQ?

ЛАБОРАТОРНАЯ РАБОТА № 8

Цель работы: провести практическое сравнение RabbitMQ и Kafka по задержке, пропускной способности, удобству отладки и сценариям применения на тесте из 10 000 сообщений.

Теоретическое обоснование

RabbitMQ и Kafka оба используются для асинхронной коммуникации, но их архитектурные цели различаются. RabbitMQ ориентирован на маршрутизацию сообщений между очередями и гибкие exchange. Kafka ориентирована на распределённый журнал событий и потоковую обработку.

Сравнение должно учитывать не только скорость. Для микросервисов важны задержка доставки, гарантия доставки, возможность replay, удобство отладки, сложность эксплуатации, требования к порядку и масштабированию.

RabbitMQ часто удобен для команд, задач, уведомлений и интеграции, где важны routing key, acknowledgements и простая модель очередей. Kafka сильна в аналитике, event sourcing, CDC, обработке больших потоков и повторном чтении.

Эксперимент с 10 000 сообщений не является промышленным benchmark, но помогает увидеть порядок величин и сформировать инженерный вывод.

В отчёте обязательно указываются параметры: размер сообщения, число producers/consumers, durable настройки, число partitions, hardware, JVM и Docker.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Benchmark Producer	Генерирует одинаковые сообщения
RabbitMQ path	Отправка в exchange/queue и чтение consumer

Kafka path	Отправка в topic и чтение consumer group
Metrics	throughput, avg latency, p95 latency, errors

Схема взаимодействия компонентов

Схема лабораторной работы № 8

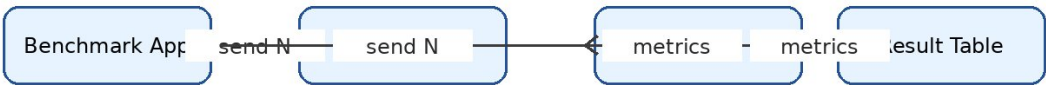


Рисунок 1 – Логическая схема выполнения лабораторной работы № 8

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Подготовка сообщения.

Создайте универсальный DTO BenchMessage. Размер payload должен быть фиксированным, чтобы сравнение было честным.

Листинг 1 – Подготовка сообщения

```

public record BenchMessage(
    String id,
    long createdAtNanos,
    String payload
) {}
  
```

Шаг 2. Методика измерения.

Перед отправкой фиксируйте время `createdAtNanos`. Consumer вычисляет задержку как `System.nanoTime() - createdAtNanos`. Для throughput измеряйте общее время доставки N сообщений.

Листинг 2 – Методика измерения

```
long started = System.nanoTime();
for (int i = 0; i < 10_000; i++) {
    BenchMessage msg = new
    BenchMessage(UUID.randomUUID().toString(),
        System.nanoTime(), payload);
    sender.send(msg);
}
long elapsedMs =
    TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - started);
```

Шаг 3. Тест RabbitMQ.

Настройте durable queue и один consumer. Укажите, включены ли publisher confirms. Без описания этих параметров результаты нельзя сравнивать.

Листинг 3 – Тест RabbitMQ

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    publisher-confirm-type: correlated
```

Шаг 4. Тест Kafka.

Создайте topic с несколькими partitions и настройте producer. Зафиксируйте acks и batch.size.

Листинг 4 – Тест Kafka

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    producer:
      acks: all
      properties:
```

```
linger.ms: 5  
batch.size: 32768
```

Шаг 5. Таблица результатов.

Сделайте минимум три прогона и заполните таблицу: брокер, сообщений/с, avg latency, p95 latency, потери/ошибки, удобство диагностики. Затем сформулируйте вывод не в стиле «кто быстрее», а «для какого сценария что выбрать».

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.
7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с

изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Подготовить сообщение размером 512 байт по теме «каталог товаров». 2) Отправить 10000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
2	1) Подготовить сообщение размером 1024 байт по теме «оформление заказов». 2) Отправить 15000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
3	1) Подготовить сообщение размером 2048 байт по теме «пользователи». 2) Отправить 20000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
4	1) Подготовить сообщение размером 256 байт по теме «оплата». 2) Отправить 5000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
5	1) Подготовить сообщение размером 512 байт по теме «складские остатки». 2) Отправить 10000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
6	1) Подготовить сообщение размером 1024 байт по теме «доставка». 2) Отправить 15000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить

	сравнительную таблицу и обосновать выбор брокера для данного сценария.
7	1) Подготовить сообщение размером 2048 байт по теме «уведомления». 2) Отправить 20000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
8	1) Подготовить сообщение размером 256 байт по теме «скидки и промокоды». 2) Отправить 5000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
9	1) Подготовить сообщение размером 512 байт по теме «аналитика продаж». 2) Отправить 10000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
10	1) Подготовить сообщение размером 1024 байт по теме «отзывы покупателей». 2) Отправить 15000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
11	1) Подготовить сообщение размером 2048 байт по теме «корзина». 2) Отправить 20000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
12	1) Подготовить сообщение размером 256 байт по теме «возвраты». 2) Отправить 5000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
13	1) Подготовить сообщение размером 512 байт по теме «биллинг подписок». 2) Отправить 10000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.

14	1) Подготовить сообщение размером 1024 байт по теме «поддержка клиентов». 2) Отправить 15000 сообщений через RabbitMQ и Kafka. 3) Выполнить 3 прогона, посчитать throughput, среднюю и p95 задержку. 4) Заполнить сравнительную таблицу и обосновать выбор брокера для данного сценария.
----	--

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Почему нельзя сравнивать брокеры только по throughput?
2. В каких задачах RabbitMQ удобнее Kafka?
3. В каких задачах Kafka удобнее RabbitMQ?
4. Как размер сообщения влияет на задержку?
5. Зачем делать несколько прогонов эксперимента?

ЛАБОРАТОРНАЯ РАБОТА № 9

Цель работы: реализовать синхронный unary-вызов ProductService через gRPC, сгенерировать Java-код из proto-файла и сравнить время ответа с REST/JSON.

Теоретическое обоснование

1. gRPC использует Protocol Buffers для описания контракта и HTTP/2 как транспорт. В отличие от REST/JSON, контракт строго типизирован, а клиентский и серверный код генерируются из .proto файла.

2. Unary RPC похож на обычный запрос-ответ: клиент отправляет один message, сервер возвращает один message. Это хороший первый шаг для сравнения с REST endpoint.

3. Protobuf field numbers являются частью бинарного формата. Их нельзя менять без миграционного плана, иначе старые клиенты могут некорректно прочитать новые сообщения.

4. gRPC особенно полезен для внутренней коммуникации сервисов, где важны производительность, строгий контракт и поддержка streaming. Для публичного браузерного API REST часто проще.

5. В лабораторной работе реализуется ProductService.getProduct и REST-аналог /products/{id}. Результаты сравнения должны учитывать warm-up JVM.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
.proto	Контракт сервиса и сообщений
protoc plugin	Генерация Java-классов
gRPC server	Реализация ProductServiceImplBase
gRPC client	Blocking stub для unary-вызова

Схема взаимодействия компонентов

Схема лабораторной работы № 9



Рисунок 1 – Логическая схема выполнения лабораторной работы № 9

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Описание proto-файла.

Создайте `src/main/proto/product.proto`. `package` и `java_package` должны соответствовать структуре проекта.

Листинг 1 – Описание proto-файла

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "ru.example.product.grpc";

service ProductService {
    rpc      GetProduct      (GetProductRequest)      returns
(ProductResponse) ;
}

message GetProductRequest {
    string id = 1;
}
```

```
message ProductResponse {
    string id = 1;
    string name = 2;
    double price = 3;
}
```

Шаг 2. Генерация кода.

Настройте `protobuf-maven-plugin` или используйте `grpc-spring-boot-starter`. После `mvn clean compile` в `target/generated-sources` появятся классы `ProductServiceGrpc`, `GetProductRequest` и `ProductResponse`.

Листинг 2 – Генерация кода

```
mvn clean compile
ls target/generated-sources/protobuf/java
ls target/generated-sources/protobuf/grpc-java
```

Шаг 3. Реализация сервера.

Сервер наследуется от `ProductServiceImplBase`. Метод возвращает `ProductResponse` через `responseObserver`.

Листинг 3 – Реализация сервера

```
@GrpcService
public class ProductGrpcService extends
ProductServiceGrpc.ProductServiceImplBase {
    @Override
    public void getProduct(GetProductRequest request,
                           StreamObserver<ProductResponse>
responseObserver) {
        ProductResponse response =
ProductResponse.newBuilder()
            .setId(request.getId())
            .setName("Laptop")
            .setPrice(1200.0)
            .build();
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

Шаг 4. Реализация клиента.

Для unary-вызова достаточно blocking stub. В реальных системах обязательно задавайте deadline.

Листинг 4 – Реализация клиента

```
ManagedChannel channel = ManagedChannelBuilder
    .forAddress("localhost", 9090)
    .usePlaintext()
    .build();

ProductServiceGrpc.ProductServiceBlockingStub stub =
    ProductServiceGrpc.newBlockingStub(channel)
        .withDeadlineAfter(2, TimeUnit.SECONDS);

ProductResponse response = stub.getProduct(
    GetProductRequest.newBuilder().setId("p-1").build());
```

Шаг 5. Сравнение с REST.

Создайте REST endpoint `/products/{id}`, выполните серию запросов к REST и gRPC. Первые 100 запросов используйте как прогрев и не включайте в расчёт.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Описать proto-сервис ProductsGrpcService с методом GetProducts. 2) Реализовать gRPC-сервер, возвращающий объект «товар». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
2	1) Описать proto-сервис OrdersGrpcService с методом GetOrders. 2) Реализовать gRPC-сервер, возвращающий объект «заказ». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
3	1) Описать proto-сервис UsersGrpcService с методом GetUsers. 2) Реализовать gRPC-сервер, возвращающий объект «пользователь». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
4	1) Описать proto-сервис PaymentsGrpcService с методом GetPayments. 2) Реализовать gRPC-сервер, возвращающий объект «платёж». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
5	1) Описать proto-сервис InventoryGrpcService с методом GetInventory. 2) Реализовать gRPC-сервер, возвращающий объект «остаток». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.

	среднее время ответа.
6	1) Описать proto-сервис DeliveryGrpcService с методом GetDelivery. 2) Реализовать gRPC-сервер, возвращающий объект «доставка». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
7	1) Описать proto-сервис NotificationsGrpcService с методом GetNotifications. 2) Реализовать gRPC-сервер, возвращающий объект «уведомление». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
8	1) Описать proto-сервис DiscountsGrpcService с методом GetDiscounts. 2) Реализовать gRPC-сервер, возвращающий объект «промокод». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
9	1) Описать proto-сервис AnalyticsGrpcService с методом GetAnalytics. 2) Реализовать gRPC-сервер, возвращающий объект «метрика». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
10	1) Описать proto-сервис ReviewsGrpcService с методом GetReviews. 2) Реализовать gRPC-сервер, возвращающий объект «отзыв». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
11	1) Описать proto-сервис CartGrpcService с методом GetCart. 2) Реализовать gRPC-сервер, возвращающий объект «позиция корзины». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
12	1) Описать proto-сервис ReturnsGrpcService с методом GetReturns. 2) Реализовать gRPC-сервер, возвращающий объект «возврат». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
13	1) Описать proto-сервис BillingGrpcService с методом GetBilling. 2) Реализовать gRPC-сервер, возвращающий объект «счёт». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.

14	1) Описать proto-сервис SupportGrpcService с методом GetSupport. 2) Реализовать gRPC-сервер, возвращающий объект «тикет». 3) Реализовать REST-аналог. 4) Провести не менее 1000 запросов к REST и gRPC и сравнить среднее время ответа.
----	---

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Для чего нужен .proto файл?
2. Чем unary gRPC отличается от REST/JSON?
3. Почему field number в protobuf нельзя менять бездумно?
4. Что такое deadline в gRPC?
5. Почему gRPC сложнее тестировать из браузера?

ЛАБОРАТОРНАЯ РАБОТА № 10

Цель работы: реализовать двунаправленный gRPC streaming для чат-подобного взаимодействия, добавить interceptor для логирования и проверить обмен потоками сообщений.

Теоретическое обоснование

Bidirectional streaming RPC позволяет клиенту и серверу независимо отправлять последовательности сообщений. Это отличается от unary-вызова: соединение живёт дольше, а ответы могут приходить не строго после завершения отправки всех запросов.

В Java gRPC streaming реализуется через StreamObserver. Сервер возвращает observer для входящего потока, а ответы отправляет в responseObserver.

Порядок сообщений сохраняется внутри каждого направления потока, но клиентский и серверный потоки независимы. Сервер может отвечать на каждое сообщение сразу или агрегировать данные.

Interceptors позволяют логировать metadata, добавлять correlationId, измерять длительность вызова или проверять авторизацию.

В лабораторной работе streaming используется как чат: клиент отправляет сообщения, сервер возвращает подтверждения и время получения.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Bidirectional RPC	rpc Chat(stream ChatMessage) returns (stream ChatAck)
StreamObserver	API обработки входящего и исходящего потока
Interceptor	Сквозное логирование вызовов
Metadata	Технические заголовки gRPC

Схема взаимодействия компонентов

Схема лабораторной работы № 10



Рисунок 1 – Логическая схема выполнения лабораторной работы № 10

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Описание proto.

Опишите ChatService с bidirectional streaming. Каждое сообщение содержит clientId, text и timestamp.

Листинг 1 – Описание proto

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "ru.example.chat.grpc";

service ChatService {
  rpc Chat(stream ChatMessage) returns (stream ChatAck);
}

message ChatMessage {
  string client_id = 1;
  string text = 2;
  int64 sent_at = 3;
}

message ChatAck {
  string client_id = 1;
  string status = 2;
  int64 received_at = 3;
}
```

Шаг 2. Серверный поток.

Метод `chat` возвращает `StreamObserver` для обработки входящих сообщений. На каждое сообщение сервер отправляет `ack`.

Листинг 2 – Серверный поток

```
@Override
public StreamObserver<ChatMessage>
chat(StreamObserver<ChatAck> responseObserver) {
    return new StreamObserver<>() {
        public void onNext(ChatMessage msg) {
            responseObserver.onNext(ChatAck.newBuilder()
                .setClientId(msg.getClientId())
                .setStatus("received: " + msg.getText())
                .setReceivedAt(System.currentTimeMillis())
                .build());
        }
        public void onError(Throwable t) { log.warn("stream
error", t); }
        public void onCompleted()
        { responseObserver.onCompleted(); }
    };
}
```

Шаг 3. Асинхронный клиент.

Клиент использует `async stub`. Сначала создаётся `responseObserver` для входящих ответов, затем `requestObserver` для отправки сообщений.

Листинг 3 – Асинхронный клиент

```
StreamObserver<ChatAck> responseObserver = new
StreamObserver<>() {
    public void onNext(ChatAck ack)
    { System.out.println(ack.getStatus()); }
    public void onError(Throwable t)
    { t.printStackTrace(); }
    public void onCompleted()
    { System.out.println("completed"); }
};

StreamObserver<ChatMessage> requestObserver =
asyncStub.chat(responseObserver);
for (int i = 1; i <= 10; i++) {
    requestObserver.onNext(ChatMessage.newBuilder()
        .setClientId("client-1")
        .setText("message-" + i)
        .setSentAt(System.currentTimeMillis())
    );
}
```

```
        .build());  
    }  
    requestObserver.onCompleted();
```

Шаг 4. Interceptor логирования.

Добавьте `ServerInterceptor`, который пишет имя метода и `metadata`. В отчёте покажите, что `interceptor` срабатывает для `streaming`-вызова один раз на соединение.

Листинг 4 – Interceptor логирования

```
@Component  
public class LoggingInterceptor implements ServerInterceptor  
{  
    public <ReqT, RespT> ServerCall.Listener<ReqT>  
    interceptCall(  
        ServerCall<ReqT, RespT> call, Metadata headers,  
        ServerCallHandler<ReqT, RespT> next) {  
        log.info("gRPC method={} headers={}",  
            call.getMethodDescriptor().getFullName(), headers);  
        return next.startCall(call, headers);  
    }  
}
```

Шаг 5. Проверка нескольких клиентов.

Запустите два клиента с разными `clientId`. Сообщения каждого клиента должны получать собственные подтверждения, а ошибки одного потока не должны завершать второй поток.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.

5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.

6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Описать bidirectional streaming для обмена сообщениями о «каталог товаров». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
2	1) Описать bidirectional streaming для обмена сообщениями о «оформление заказов». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
3	1) Описать bidirectional streaming для обмена сообщениями о «пользователи». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.

4	1) Описать bidirectional streaming для обмена сообщениями о «оплата». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
5	1) Описать bidirectional streaming для обмена сообщениями о «складские остатки». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
6	1) Описать bidirectional streaming для обмена сообщениями о «доставка». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
7	1) Описать bidirectional streaming для обмена сообщениями о «уведомления». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
8	1) Описать bidirectional streaming для обмена сообщениями о «скидки и промокоды». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
9	1) Описать bidirectional streaming для обмена сообщениями о «аналитика продаж». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
10	1) Описать bidirectional streaming для обмена сообщениями о «отзывы покупателей». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata

	и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
11	1) Описать bidirectional streaming для обмена сообщениями о «корзина». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
12	1) Описать bidirectional streaming для обмена сообщениями о «возвраты». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
13	1) Описать bidirectional streaming для обмена сообщениями о «биллинг подписок». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.
14	1) Описать bidirectional streaming для обмена сообщениями о «поддержка клиентов». 2) Клиент отправляет поток из 10 сообщений, сервер возвращает подтверждения. 3) Добавить interceptor для логирования metadata и времени обработки. 4) Запустить 2 параллельных клиента и показать независимость потоков.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;

— результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;

— анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;

— ответы на вопросы для защиты.

Вопросы для защиты работы

1. Какие четыре типа gRPC-вызовов существуют?
2. Что такое StreamObserver?
3. Зачем вызывать onCompleted?
4. Что делают interceptors?
5. Какие риски есть у долгоживущих streaming-соединений?

ЛАБОРАТОРНАЯ РАБОТА № 11

Цель работы: реализовать распределённую транзакцию Заказ - Оплата - Доставка без центрального координатора, используя Kafka-события и компенсирующие транзакции.

Теоретическое обоснование

В микросервисах каждый сервис владеет своей БД. Классическая ACID-транзакция между несколькими БД становится сложной и плохо масштабируемой. Паттерн Saga разбивает бизнес-процесс на локальные транзакции и компенсирующие действия.

Хореография означает, что нет центрального координатора. Каждый сервис реагирует на события и публикует новые события. Order Service публикует OrderCreated, Payment Service резервирует оплату и публикует PaymentCompleted или PaymentFailed, Delivery Service создаёт доставку.

Компенсация - это бизнес-операция отмены, а не технический rollback. Например, если оплата не прошла, Order Service переводит заказ в CANCELLED. Если доставка не создана, Payment Service может отменить резервирование средств.

Ключевые требования Saga: наличие sagaId, идемпотентность обработчиков, журнал событий или таблица обработанных eventId, понятные состояния заказа.

Лабораторная работа требует показать успешный сценарий и ошибочный сценарий с компенсацией.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Order Service	Создаёт заказ и меняет его состояние
Payment Service	Резервирует оплату или публикует отказ
Delivery Service	Создаёт доставку после успешной оплаты
Kafka events	Связь сервисов через события

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Модель событий.

Определите общий набор событий. Для учебного проекта достаточно JSON-сообщений с eventId и sagaId.

Листинг 1 – Модель событий

```
public record OrderCreated(String eventId, String sagaId,
String orderId, BigDecimal amount) {}
public record PaymentCompleted(String eventId, String
sagaId, String orderId, String paymentId) {}
public record PaymentFailed(String eventId, String sagaId,
String orderId, String reason) {}
public record DeliveryCreated(String eventId, String sagaId,
String orderId, String deliveryId) {}
public record OrderCancelled(String eventId, String sagaId,
String orderId, String reason) {}
```

Шаг 2. Order Service публикует OrderCreated.

Локальная транзакция Order Service сохраняет заказ в состоянии NEW и публикует событие. В реальном проекте лучше использовать outbox pattern; в лабораторной допускается прямой publish после сохранения.

Листинг 2 – Order Service публикует OrderCreated

```
@Transactional
public Order createOrder(CreateOrderRequest request) {
    Order order = repository.save(new
Order(request.amount(), OrderStatus.NEW));
    kafkaTemplate.send("orders", order.getId(),
new OrderCreated(UUID.randomUUID().toString(),
UUID.randomUUID().toString(),
order.getId(), order.getAmount()));
    return order;
}
```

Шаг 3. Payment Service реагирует на событие.

Если сумма меньше лимита, публикуется `PaymentCompleted`; иначе `PaymentFailed`. Условие ошибки нужно для демонстрации компенсации.

Листинг 3 – Payment Service реагирует на событие

```
@KafkaListener(topics = "orders", groupId = "payment-
service")
public void onOrderCreated(OrderCreated event) {
    if (event.amount().compareTo(new BigDecimal("1000")) >
0) {
        kafkaTemplate.send("payments", event.orderId(),
            new PaymentFailed(newId(), event.sagaId(),
event.orderId(), "limit exceeded"));
    } else {
        kafkaTemplate.send("payments", event.orderId(),
            new PaymentCompleted(newId(), event.sagaId(),
event.orderId(), "pay-" + event.orderId()));
    }
}
```

Шаг 4. Компенсация в Order Service.

Order Service слушает `PaymentFailed` и переводит заказ в `CANCELLED`. Обработчик должен быть идемпотентным: повтор события не должен повторно менять бизнес-состояние некорректно.

Листинг 4 – Компенсация в Order Service

```
@KafkaListener(topics = "payments", groupId = "order-
service")
public void onPaymentFailed(PaymentFailed event) {
    Order order =
repository.findById(event.orderId()).orElseThrow();
    if (order.getStatus() != OrderStatus.CANCELLED) {
        order.setStatus(OrderStatus.CANCELLED);
        repository.save(order);
    }
}
```

Шаг 5. Тестирование двух сценариев.

Создайте заказ на 500 рублей и заказ на 5000 рублей. Первый должен пройти оплату и доставку, второй - перейти в `CANCELLED`. В отчёте приведите цепочку событий для каждого `sagaId`.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.
7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Построить Saga-хореографию для сценария «каталог товаров»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и

	итоговые состояния сервисов.
2	1) Построить Saga-хореографию для сценария «оформление заказов»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
3	1) Построить Saga-хореографию для сценария «пользователи»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
4	1) Построить Saga-хореографию для сценария «оплата»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
5	1) Построить Saga-хореографию для сценария «складские остатки»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
6	1) Построить Saga-хореографию для сценария «доставка»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
7	1) Построить Saga-хореографию для сценария «уведомления»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
8	1) Построить Saga-хореографию для сценария «скидки и промокоды»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и

	итоговые состояния сервисов.
9	1) Построить Saga-хореографию для сценария «аналитика продаж»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
10	1) Построить Saga-хореографию для сценария «отзывы покупателей»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
11	1) Построить Saga-хореографию для сценария «корзина»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
12	1) Построить Saga-хореографию для сценария «возвраты»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
13	1) Построить Saga-хореографию для сценария «биллинг подписок»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.
14	1) Построить Saga-хореографию для сценария «поддержка клиентов»: Created → Reserved/Paid → Completed или Failed. 2) Использовать sagaId/correlationId. 3) Реализовать успешный и ошибочный сценарии с компенсирующим событием. 4) В отчёте привести таблицу событий и итоговые состояния сервисов.

Содержание отчёта

— тема и цель лабораторной работы;

- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Почему 2PC редко используют в микросервисах?
2. Что такое Saga?
3. Чем хореография отличается от оркестрации?
4. Почему компенсации должны быть идемпотентными?
5. Зачем нужен sagaId/correlationId?

ЛАБОРАТОРНАЯ РАБОТА № 12

Цель работы: реализовать централизованный Saga Orchestrator с сохранением состояния, компенсациями и защитой API через Keycloak JWT/RBAC.

Теоретическое обоснование

Оркестрация Saga отличается от хореографии наличием координатора. Оркестратор знает порядок шагов, вызывает сервисы, сохраняет состояние процесса и запускает компенсации при сбое.

Централизованный подход удобен для сложных бизнес-процессов, где нужно явно видеть состояние: `STARTED`, `PAYMENT_RESERVED`, `DELIVERY_CREATED`, `COMPENSATING`, `COMPLETED`, `FAILED`.

Оркестратор должен сохранять состояние в БД, чтобы после рестарта продолжить или корректно завершить Saga. Иначе сбой координатора создаст зависшие платежи или заказы.

Вторая часть работы посвящена безопасности. Keycloak выдаёт JWT, Gateway и микросервисы проверяют подпись токена и роли. RBAC позволяет ограничить административные операции.

В отчёте нужно показать не только успешную авторизацию, но и отрицательные сценарии: 401 без токена и 403 при недостаточной роли.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Saga Orchestrator	Координация шагов и компенсаций
Saga state table	Хранение статуса процесса
Keycloak	Identity Provider и выдача JWT
Resource Server	Проверка JWT в Spring Security

Схема взаимодействия компонентов

Схема лабораторной работы № 12

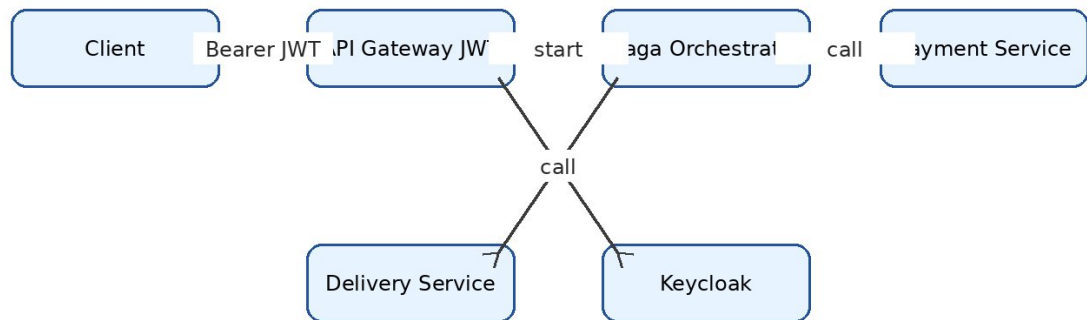


Рисунок 1 – Логическая схема выполнения лабораторной работы № 12

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Таблица состояния Saga.

Создайте таблицу, где хранится текущий шаг, orderId и ошибка. Это основа восстановления после рестарта.

Листинг 1 – Таблица состояния Saga

```
create table saga_state (  
  saga_id varchar(64) primary key,  
  order_id varchar(64) not null,  
  status varchar(64) not null,  
  current_step varchar(64),  
  last_error text,  
  updated_at timestamp not null  
);
```

Шаг 2. Логика оркестратора.

Оркестратор последовательно вызывает Payment Service и Delivery Service. При ошибке доставки вызывается компенсация оплаты.

Листинг 2 – Логика оркестратора

```
public void startSaga(String orderId, BigDecimal amount) {  
    SagaState saga =  
sagaRepository.save(SagaState.started(orderId));  
    try {  
        paymentClient.reserve(orderId, amount);  
        saga.markPaymentReserved();  
        deliveryClient.createDelivery(orderId);  
        saga.markCompleted();  
    } catch (Exception ex) {  
        saga.markCompensating(ex.getMessage());  
        paymentClient.cancelReservation(orderId);  
        saga.markFailed();  
    } finally {  
        sagaRepository.save(saga);  
    }  
}
```

Шаг 3. Проверка восстановления.

Запустите Saga, искусственно остановите orchestrator после PAYMENT_RESERVED, затем поднимите его снова. При старте сервис должен найти незавершённые Saga и продолжить или компенсировать.

Листинг 3 – Проверка восстановления

```
@EventListener(ApplicationReadyEvent.class)  
public void recover() {  
  
sagaRepository.findUnfinished().forEach(this::recoverSaga);  
}
```

Шаг 4. Запуск Keycloak.

Для учебной работы используйте контейнер Keycloak в dev-режиме. Создайте realm microservices, client gateway-client, роли USER и ADMIN.

Листинг 4 – Запуск Keycloak

```
docker run --rm --name keycloak -p 8085:8080 \  
-e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin \  
quay.io/keycloak/keycloak:26.0 start-dev
```

Шаг 5. Настройка Spring Security.

Gateway и микросервисы настраиваются как OAuth2 Resource Server. issuer-uri должен указывать на realm Keycloak.

Листинг 5 – Настройка Spring Security

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri:
http://localhost:8085/realms/microservices

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    SecurityFilterChain security(HttpSecurity http) throws
Exception {
        return http.authorizeHttpRequests(auth -> auth
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated())
            .oauth2ResourceServer(oauth -> oauth.jwt())
            .build();
    }
}
```

Шаг 6. Проверка 401/403/200.

Выполните запрос без токена, с токеном USER и с токеном ADMIN. В отчёте приведите HTTP-статусы и расшифруйте различие между authentication и authorization.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.

5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.

6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Реализовать Saga Orchestrator для сценария «каталог товаров» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /products/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
2	1) Реализовать Saga Orchestrator для сценария «оформление заказов» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /orders/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
3	1) Реализовать Saga Orchestrator для сценария «пользователи» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /users/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
4	1) Реализовать Saga Orchestrator для сценария «оплата» с сохранением

	состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /payments/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
5	1) Реализовать Saga Orchestrator для сценария «складские остатки» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /inventory/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
6	1) Реализовать Saga Orchestrator для сценария «доставка» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /delivery/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
7	1) Реализовать Saga Orchestrator для сценария «уведомления» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /notifications/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
8	1) Реализовать Saga Orchestrator для сценария «скидки и промокоды» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /discounts/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
9	1) Реализовать Saga Orchestrator для сценария «аналитика продаж» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /analytics/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
10	1) Реализовать Saga Orchestrator для сценария «отзывы покупателей» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /reviews/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.

11	1) Реализовать Saga Orchestrator для сценария «корзина» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /cart/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
12	1) Реализовать Saga Orchestrator для сценария «возвраты» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /returns/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
13	1) Реализовать Saga Orchestrator для сценария «биллинг подписок» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль USER; защитить endpoint /billing/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.
14	1) Реализовать Saga Orchestrator для сценария «поддержка клиентов» с сохранением состояния. 2) Добавить компенсации и восстановление после рестарта. 3) В Keycloak создать роль ADMIN; защитить endpoint /support/admin. 4) Проверить 401 без токена, 403 без роли и успешный доступ с нужной ролью.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;

- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Когда оркестрация удобнее хореографии?
2. Зачем оркестратор хранит состояние?
3. Что должно происходить после рестарта оркестратора?
4. Как Keuscloak участвует в выдаче JWT?
5. Чем 401 отличается от 403?

ЛАБОРАТОРНАЯ РАБОТА № 13

Цель работы: запустить микросервисы с Dapr sidecar, выполнить межсервисный вызов через Dapr API и сохранить состояние заказа в Redis state store.

Теоретическое обоснование

Dapr - Distributed Application Runtime - добавляет к микросервису sidecar-процесс, который предоставляет стандартные building blocks: service invocation, pub/sub, state management, bindings и observability.

Service Invocation позволяет вызывать сервис по app-id, а не по адресу и порту. Dapr берёт на себя discovery, retry, timeout и, в Kubernetes-режиме, mTLS между sidecar.

State Management API позволяет сохранять пары key/value в разные хранилища через одинаковый HTTP/gRPC интерфейс. Для локальной работы обычно используется Redis.

Главная идея Dapr - вынести повторяющуюся инфраструктурную логику из кода приложения. Однако sidecar увеличивает число процессов и требует понимания конфигурации компонентов.

В лабораторной работе Order Service вызывает Product Service через Dapr и сохраняет состояние заказа в Redis без прямой зависимости от Redis-клиента в Java-коде.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
dapr run	Запуск приложения вместе с sidecar
app-id	Логическое имя сервиса
Service Invocation API	/v1.0/invoke/{app-id}/method/{path}
State Store	Компонент statestore на базе Redis

Схема взаимодействия компонентов

Схема лабораторной работы № 13



Рисунок 1 – Логическая схема выполнения лабораторной работы № 13

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Инициализация Dapr.

Установите Dapr CLI и выполните `dapr init`. Локально будут подняты контейнеры `dapr_redis` и `dapr_zipkin`.

Листинг 1 – Инициализация Dapr

```
dapr init
dapr --version
docker ps --filter name=dapr
```

Шаг 2. Компонент state store.

Проверьте файл `components/statestore.yaml`. Он описывает Redis как state store с именем `statestore`.

Листинг 2 – Компонент state store

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: statestore
spec:
  type: state.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ''
```

Шаг 3. Запуск Product Service.

Запустите Product Service на порту 8081 с app-id product-service. Dapr sidecar получит отдельный HTTP-порт 3501.

Листинг 3 – Запуск Product Service

```
dapr run --app-id product-service \
  --app-port 8081 --dapr-http-port 3501 \
  -- java -jar product-service.jar
```

Шаг 4. Вызов через Dapr.

Order Service или curl может вызвать Product Service без знания его внутреннего адреса. Путь после method соответствует REST endpoint целевого сервиса.

Листинг 4 – Вызов через Dapr

```
curl http://localhost:3500/v1.0/invoke/product-
service/method/products/p-1
```

Шаг 5. Сохранение состояния.

Order Service сохраняет заказ через State Management API. В Java можно использовать обычный WebClient к sidecar на localhost.

Листинг 5 – Сохранение состояния

```
curl -X POST http://localhost:3500/v1.0/state/statestore \
  -H 'Content-Type: application/json' \
  -d '[{"key":"order-1","value":{"status":"CREATED"}}]'
```

```
curl http://localhost:3500/v1.0/state/statestore/order-1
```

Шаг 6. Анализ результата.

В отчёте объясните, почему приложение не зависит от Redis API напрямую и что придётся поменять при переходе на PostgreSQL state store.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.
7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Запустить Product Service с Dapr app-id=products-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «товар» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
2	1) Запустить Order Service с Dapr app-id=orders-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «заказ» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
3	1) Запустить User Service с Dapr app-id=users-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «пользователь» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
4	1) Запустить Payment Service с Dapr app-id=payments-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «платёж» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
5	1) Запустить Inventory Service с Dapr app-id=inventory-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «остаток» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
6	1) Запустить Delivery Service с Dapr app-id=delivery-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «доставка» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
7	1) Запустить Notification Service с Dapr app-id=notifications-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «уведомление» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
8	1) Запустить Discount Service с Dapr app-id=discounts-service. 2)

	Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «промокод» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
9	1) Запустить Analytics Service с Dapr app-id=analytics-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «метрика» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
10	1) Запустить Review Service с Dapr app-id=reviews-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «отзыв» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
11	1) Запустить Cart Service с Dapr app-id=cart-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «позиция корзины» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
12	1) Запустить Return Service с Dapr app-id=returns-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «возврат» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
13	1) Запустить Billing Service с Dapr app-id=billing-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «счёт» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.
14	1) Запустить Support Service с Dapr app-id=support-service. 2) Вызвать сервис через Dapr Service Invocation. 3) Сохранить состояние объекта «тикет» в Redis state store. 4) Прочитать состояние через Dapr API и объяснить, где в коде исчезает зависимость от конкретного хранилища.

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;

- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Что такое Dapr sidecar?
2. Как Dapr Service Invocation упрощает межсервисные вызовы?
3. Что даёт State Management API?
4. Какие минусы есть у sidecar-подхода?
5. Чем Dapr отличается от Service Mesh?

ЛАБОРАТОРНАЯ РАБОТА № 14

Цель работы: использовать Dapr Pub/Sub для публикации OrderCreated, настроить компонент RabbitMQ, затем переключить конфигурацию на Kafka без изменения кода приложения.

Теоретическое обоснование

Dapr Pub/Sub предоставляет единый API публикации и подписки поверх разных брокеров сообщений. Приложение публикует событие в pubsub component и topic, не используя напрямую RabbitTemplate или KafkaTemplate.

Компонент Dapr описывает конкретный брокер: RabbitMQ, Kafka, Redis Streams и другие. Смена брокера выполняется через YAML-конфигурацию компонента.

Subscriber объявляет подписки через endpoint /dapr/subscribe или через declarative subscription. Dapr доставляет сообщения в HTTP endpoint приложения.

Абстракция Dapr скрывает часть различий брокеров, но не отменяет их полностью. Retention, ordering, partitions, routing и delivery semantics всё равно нужно понимать.

В лабораторной работе сначала используется RabbitMQ, затем Kafka. Java-код публикации и обработки события должен остаться неизменным.

Таблица 1 – Основные компоненты лабораторной работы

Компонент	Назначение
Pub/Sub API	POST /v1.0/publish/{pubsub}/{topic}
Component YAML	Описание RabbitMQ или Kafka
Subscriber	Endpoint приложения для получения событий
CloudEvent	Формат события, передаваемый Dapr

Схема взаимодействия компонентов

Схема лабораторной работы № 14



Рисунок 1 – Логическая схема выполнения лабораторной работы № 14

Пример выполнения задания

Ниже приведён учебный пример выполнения работы. Его цель – показать не только итоговые листинги, но и последовательность проектных решений: какой компонент создаётся, зачем он нужен, каким способом проверяется и какой результат должен быть отражён в отчёте.

Шаг 1. Компонент RabbitMQ.

Создайте `components/pubsub.yaml`. Имя `pubsub` будет использоваться в URL публикации.

Листинг 1 – Компонент RabbitMQ

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: pubsub
spec:
  type: pubsub.rabbitmq
  version: v1
  metadata:
    - name: host
      value: amqp://guest:guest@localhost:5672
    - name: durable
      value: 'true'
```

Шаг 2. Публикация события.

Order Service может публиковать событие через HTTP sidecar. Код не содержит RabbitMQ API.

Листинг 2 – Публикация события

```
curl -X POST \
http://localhost:3500/v1.0/publish/pubsub/order-created \
-H 'Content-Type: application/json' \
-d '{"orderId":"ord-1","amount":500}'
```

Шаг 3. Подписка приложения.

Notification Service сообщает Dapr, на какие topics он подписан. Dapr будет отправлять события на route /events/order-created.

Листинг 3 – Подписка приложения

```
@GetMapping("/dapr/subscribe")
public List<Map<String, String>> subscribe() {
    return List.of(Map.of(
        "pubsubname", "pubsub",
        "topic", "order-created",
        "route", "/events/order-created"
    ));
}

@PostMapping("/events/order-created")
public void onOrderCreated(@RequestBody Map<String, Object>
event) {
    log.info("Dapr event: {}", event);
}
```

Шаг 4. Запуск subscriber с Dapr.

Запустите Notification Service через dapr run. Sidecar прочитает подписку и подключится к pubsub component.

Листинг 4 – Запуск subscriber с Dapr

```
dapr run --app-id notification-service \
--app-port 8083 --dapr-http-port 3503 \
--resources-path ./components \
-- java -jar notification-service.jar
```

Шаг 5. Переключение на Kafka.

Замените pubsub.yaml на Kafka component с тем же metadata.name=pubsub. URL публикации и Java-код остаются теми же; меняется только конфигурация компонента.

Листинг 5 – Переключение на Kafka

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: pubsub
spec:
  type: pubsub.kafka
  version: v1
  metadata:
    - name: brokers
      value: localhost:9092
    - name: consumerGroup
      value: notification-service
```

Шаг 6. Сравнение.

В отчёте приложите две версии pubsub.yaml и объясните, какие свойства RabbitMQ/Kafka всё равно нужно учитывать несмотря на одинаковый Dapr API.

Порядок выполнения работы

1. Изучить цель работы и теоретическое обоснование, выписать назначение каждого компонента схемы.
2. Создать минимальный набор проектов Spring Boot или конфигураций, указанных в примере выполнения.
3. Запустить инфраструктурные компоненты через Docker и проверить их доступность отдельными командами.
4. Реализовать программную часть по примеру, затем адаптировать её под индивидуальный вариант.
5. Выполнить функциональную проверку: положительный сценарий, ошибочный сценарий и повторный запуск после изменения состояния.
6. Собрать доказательства выполнения: скриншоты, логи, ответы curl/Postman, фрагменты конфигурации и выводы.

7. Оформить отчёт с анализом результата и ответами на контрольные вопросы.

Задания по вариантам

Номер варианта определяется по порядковому номеру студента в журнале группы. Если номер больше 14, используется формула: вариант = $((N - 1) \bmod 14) + 1$. Вариант выполняется на основе общего примера, но с изменённой предметной областью, именами сервисов, событиями, endpoint-ами и параметрами эксперимента.

Таблица 3 – Индивидуальные задания по вариантам

Вариант	Индивидуальное задание
1	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие ProductsCreated через /v1.0/publish/pubsub/products-created. 3) Реализовать подписчик /events/products-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
2	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие OrdersCreated через /v1.0/publish/pubsub/orders-created. 3) Реализовать подписчик /events/orders-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
3	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие UsersCreated через /v1.0/publish/pubsub/users-created. 3) Реализовать подписчик /events/users-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
4	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие PaymentsCreated через /v1.0/publish/pubsub/payments-created. 3) Реализовать подписчик /events/payments-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
5	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие InventoryCreated через /v1.0/publish/pubsub/inventory-created. 3) Реализовать подписчик /events/inventory-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.

6	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие DeliveryCreated через /v1.0/publish/pubsub/delivery-created. 3) Реализовать подписчик /events/delivery-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
7	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие NotificationsCreated через /v1.0/publish/pubsub/notifications-created. 3) Реализовать подписчик /events/notifications-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
8	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие DiscountsCreated через /v1.0/publish/pubsub/discounts-created. 3) Реализовать подписчик /events/discounts-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
9	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие AnalyticsCreated через /v1.0/publish/pubsub/analytics-created. 3) Реализовать подписчик /events/analytics-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
10	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие ReviewsCreated через /v1.0/publish/pubsub/reviews-created. 3) Реализовать подписчик /events/reviews-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
11	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие CartCreated через /v1.0/publish/pubsub/cart-created. 3) Реализовать подписчик /events/cart-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
12	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие ReturnsCreated через /v1.0/publish/pubsub/returns-created. 3) Реализовать подписчик /events/returns-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
13	1) Настроить Dapr Pub/Sub поверх RabbitMQ. 2) Опубликовать событие BillingCreated через /v1.0/publish/pubsub/billing-created. 3) Реализовать подписчик /events/billing-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.

14	1) Настроить Dapr Pub/Sub поверх Kafka. 2) Опубликовать событие SupportCreated через /v1.0/publish/pubsub/support-created. 3) Реализовать подписчик /events/support-created. 4) Переключить компонент на другой брокер без изменения Java-кода и сравнить конфигурации.
----	---

Содержание отчёта

- тема и цель лабораторной работы;
- краткое описание архитектуры решения и схема взаимодействия компонентов;
- использованные версии JDK, Spring Boot, Spring Cloud, Docker-образов и вспомогательных инструментов;
- ключевые листинги кода и YAML-конфигураций с пояснением назначения;
- результаты проверки: команды запуска, HTTP/gRPC/Kafka/RabbitMQ/Dapr-запросы, логи сервисов, скриншоты UI при наличии;
- анализ ошибок, ограничений и вывод о применимости изученного механизма в микросервисной архитектуре;
- ответы на вопросы для защиты.

Вопросы для защиты работы

1. Как Dapr Pub/Sub скрывает конкретный брокер?
2. Почему код приложения не меняется при смене RabbitMQ на Kafka?
3. Какие различия брокеров Dapr не может полностью скрыть?
4. Как описывается подписка в Dapr?
5. Когда Dapr предпочтительнее прямого SDK брокера?

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине
«Микросервисные технологии» для студентов направления подготовки
09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ	3
МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА..	5
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ	7
ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ	8
КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ.....	10

ВВЕДЕНИЕ

Настоящие методические указания определяют порядок организации и проведения самостоятельной работы студентов по дисциплине «Микросервисные технологии» и являются дополнением к методическим указаниям по выполнению лабораторных работ. Самостоятельная работа рассматривается как обязательная составляющая образовательного процесса, обеспечивающая закрепление и углубление знаний, полученных на аудиторных занятиях.

Цель самостоятельной работы — сформировать у студентов устойчивые навыки самостоятельного освоения современных инструментов и шаблонов проектирования и реализации микросервисных систем (API Gateway, Service Discovery, шаблоны обеспечения отказоустойчивости, брокеры сообщений RabbitMQ и Kafka, gRPC, шаблон Saga, среда Dapr) и подготовить их к осознанному выполнению и защите лабораторных работ.

Указания содержат общую характеристику видов самостоятельной работы, рекомендации по изучению теоретического материала, порядок подготовки к выполнению практических (лабораторных) работ, перечень вопросов для собеседования и критерии оценивания сформированности компетенций.

ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа студента по дисциплине включает проработку теоретического материала, подготовку к лабораторным работам, самостоятельное выполнение индивидуальных заданий и подготовку к их защите. Основные виды самостоятельной работы и формы контроля приведены в таблице 1.

Таблица 1 — Виды самостоятельной работы и формы контроля

Вид самостоятельной работы	Содержание деятельности	Форма контроля
Проработка теоретического материала	Изучение лекций, учебной и официальной технической документации по микросервисным технологиям (Spring Cloud, Consul, Resilience4j, RabbitMQ, Kafka, gRPC, Dapr)	Собеседование, тестирование
Подготовка к лабораторной работе	Изучение теоретического обоснования и примера, подготовка рабочего окружения, проработка порядка выполнения	Допуск к выполнению работы
Выполнение индивидуального задания	Самостоятельное выполнение варианта задания, оформление отчёта	Проверка отчёта
Подготовка к защите работы	Подготовка ответов на контрольные вопросы, обоснование принятых технических решений	Защита отчёта

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

Изучение теоретического материала рекомендуется проводить последовательно, в соответствии с порядком следования лабораторных работ. По каждой теме необходимо ознакомиться с основными понятиями, разобрать приведённый в работе пример и проверить понимание материала

по вопросам для самоконтроля. Перечень тем для самостоятельного изучения приведён в таблице 2.

Таблица 2 — Темы для самостоятельного изучения теоретического материала

№	Тема для самостоятельного изучения	Вопросы для проработки
1	API Gateway на Spring Cloud Gateway	назначение шлюза, понятия Route, Predicate и Filter, фильтрация запросов
2	Service Discovery с Consul	регистрация сервисов, схема lb://service-name, проверки работоспособности
3	Балансировка нагрузки и Health Checks	клиентская балансировка, исключение неработающих экземпляров
4	Circuit Breaker (Resilience4j)	состояния предохранителя, резервное поведение, каскадные отказы
5	Retry и Timeout	повторные попытки, идемпотентность, согласование тайм-аутов
6	RabbitMQ: Producer и Consumer	exchange, queue, binding, подтверждения доставки
7	Kafka: Producer и Consumer	topic, partition, offset, consumer group, журнал фиксации
8	Сравнение RabbitMQ и Kafka	модель очередей и модель журнала, критерии выбора брокера

№	Тема для самостоятельного изучения	Вопросы для проработки
9	gRPC сервер и клиент (unary)	Protocol Buffers, файл .proto, унарный вызов, транспорт HTTP/2
10	gRPC streaming (bidirectional)	серверный, клиентский и двунаправленный потоковые вызовы
11	Saga (хореография) через Kafka	шаблон Saga, хореография, компенсирующие транзакции
12	Saga (оркестрация) с координатором + JWT	оркестрация, роль координатора, структура и проверка JWT
13	Dapr: Service Invocation и State Management	модель building blocks, sidecar, вызов сервисов, управление состоянием
14	Dapr: Pub/Sub (абстракция над Kafka/RabbitMQ)	публикация-подписка, компоненты, абстракция над брокерами

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Выполнение каждой лабораторной работы осуществляется самостоятельно в соответствии с полученным вариантом индивидуального задания. Рекомендуется придерживаться последовательности этапов, приведённой в таблице 3.

Таблица 3 — Этапы выполнения лабораторной работы

Этап	Содержание деятельности
Подготовительный	Изучение теоретического обоснования и

Этап	Содержание деятельности
	примера, подготовка рабочего окружения (JDK, Docker, инфраструктурные сервисы), уточнение варианта задания
Основной	Самостоятельное выполнение индивидуального задания, фиксация выполняемых команд, конфигураций и полученных результатов
Оформление отчёта	Подготовка отчёта в соответствии с требованиями работы: цель, ход выполнения, листинги, скриншоты, выводы
Защита	Ответы на контрольные вопросы, обоснование принятых решений и полученных результатов

Отчёт оформляется в текстовом редакторе, листинги кода и конфигураций приводятся моноширинным шрифтом. По итогам работы формулируются выводы, отражающие достигнутые результаты и приобретённые навыки.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

Для подготовки к собеседованию по результатам самостоятельного изучения литературы рекомендуется проработать следующие вопросы:

1. Зачем в микросервисной системе нужен API Gateway и чем различаются понятия Route, Predicate и Filter в Spring Cloud Gateway?
2. Что такое Service Discovery и как работает обращение к сервису по схеме lb://service-name в связке с Consul?
3. Как балансировка нагрузки и проверки работоспособности (health checks) обеспечивают распределение запросов между экземплярами?
4. Какие состояния имеет Circuit Breaker (Resilience4j) и для чего предназначено резервное поведение (fallback)?

5. Почему повторные попытки (Retry) требуют идемпотентности и как согласуются тайм-ауты между уровнями вызова?
6. Из каких элементов состоит модель RabbitMQ (exchange, queue, binding) и как обеспечивается подтверждение обработки сообщений?
7. Что такое topic, partition, offset и consumer group в Apache Kafka?
8. По каким критериям выбирается брокер сообщений и чем принципиально различаются RabbitMQ и Kafka?
9. Какова роль Protocol Buffers и файла .proto при разработке gRPC-сервисов и как устроен унарный вызов?
10. Какие типы потоковых вызовов поддерживает gRPC и в чём особенность двунаправленного потока?
11. В чём суть шаблона Saga в хореографическом варианте и для чего нужны компенсирующие транзакции?
12. Какова роль координатора в оркестрационной саге и как применяется JWT для защиты межсервисного взаимодействия?
13. Что представляет собой среда Dapr, как реализуются модель sidecar, вызов сервисов и управление состоянием?
14. Как Dapr реализует публикацию-подписку и какие различия брокеров скрывает абстракция pub/sub?

КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Оценивание результатов самостоятельной работы осуществляется по итогам выполнения и защиты лабораторных работ, а также собеседования. Критерии оценивания приведены в таблице 4.

Таблица 4 — Критерии оценивания

Оценка	Критерии оценивания
«Отлично»	Задание выполнено в полном объёме и без ошибок; отчёт оформлен в соответствии со всеми требованиями; студент свободно владеет

Оценка	Критерии оценивания
	теоретическим материалом, обосновывает выбор технологий и анализирует поведение системы.
«Хорошо»	Задание выполнено в полном объёме с незначительными недочётами; отчёт оформлен в основном корректно; студент владеет материалом и отвечает на вопросы, допуская отдельные неточности.
«Удовлетворительно»	Задание выполнено частично; отчёт содержит ошибки и недочёты; студент демонстрирует фрагментарные знания и отвечает на вопросы с затруднениями.
«Неудовлетворительно»	Задание не выполнено или выполнено с грубыми ошибками; отчёт отсутствует или не соответствует требованиям; студент не владеет теоретическим материалом.