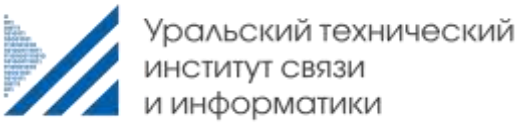
Министерство цифрового развития, связи и массовых коммуникаций РФ
Уральский технический институт связи и информатики (филиал) ФГБОУ ВО
"Сибирский государственный университет телекоммуникаций и информатики" в г. Екатеринбурге (УрТИСИ СибГУТИ)



ОТЧЕТ
По дисциплине «Сетевое программирование»
Практическое занятие №9
«Микросервисная архитектура»

Выполнила: студентка группы ПЕ-21б

Морос Е.Е.

Проверил: преподаватель

Бурумбаев Д.И.

Екатеринбург 2025

1 Цель работы:
1.1. Закрепление знаний по теме ««Микросервисная архитектура»
2 Подготовка к работе:
2.1. Изучить теоретический материал по «Микросервисная архитектура.
3 Задание:
3.1 Ответить письменно на вопросы тестового задания.

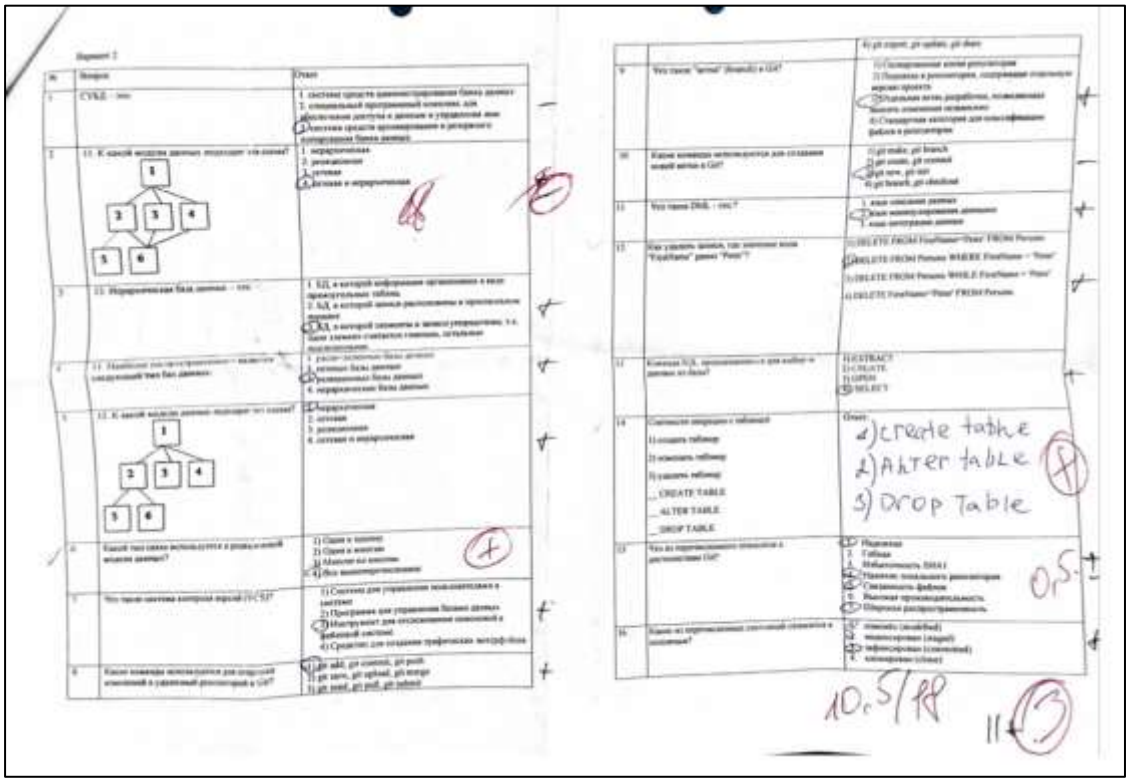


Рисунок 1 – Решение теста

4. Обобщенные вопросы тестового задания:
4.1 Что такое микросервисы и в чем их отличие от монолитной архитектуры?

Микросервисы — это стиль архитектуры, при котором приложение строится как набор отдельных, слабо связанных сервисов, где каждый отвечает за определённую функцию (например, аутентификация, оплата, заказ). Монолитная архитектура — это структура, при которой вся функциональность приложения реализована внутри одного общего кода и развёртывается как единое целое.

Главное отличие:

В монолите все модули объединены в единый код, изменения и масштабирование затрагивают всю систему целиком. В микросервисах компоненты независимы и могут развиваться, деплоиться, масштабироваться отдельно.

4.2 Какие преимущества предоставляет микросервисная архитектура в сравнении с монолитной?

- Лёгкость масштабирования отдельных компонентов
- Независимость разработки, тестирования и развертывания сервисов
- Повышение отказоустойчивости (в случае сбоя — только один сервис)
- Возможность технологий (каждый микросервис может быть написан на своём языке/стеке)

- Проще сопровождать и обновлять отдельные части
- Улучшенная гибкость командной работы (несколько команд над разными сервисами)

4.3 Какие основные принципы следует учитывать при проектировании микросервисной архитектуры?

- Малый размер и ограниченная функциональность каждого микросервиса
- Чёткие API/интерфейсы взаимодействия (REST, gRPC, GraphQL)
- Автономность (отдельное хранение данных, раздельное развертывание)
- Независимость жизненного цикла (разработка, деплой, масштабирование)
- Децентрализованное управление данными (каждый сервис — свой БД)
- Автоматизация тестирования и развёртывания
- Реализация DevOps/CI/CD-процессов

4.4 Какие вызовы возникают при разработке и масштабировании микросервисных систем?

- Сложность интеграции, отслеживания и мониторинга множества сервисов
- Управление зависимостями и версиями сервисов
- Сложности с транзакциями между сервисами (нет единой ACID)
- Однородность данных и их согласованность (реализация eventual consistency)
- Сетевая задержка и отказовоустойчивость каналов связи
- Повышенные требования к автоматизации CI/CD, мониторинг, логирование
- Безопасность и авторизация

4.5 Какие технологии можно использовать для реализации связи между микросервисами?

- HTTP/REST — популярный протокол для синхронного взаимодействия (JSON, XML)
- gRPC — высокопроизводительный двоичный протокол (на основе Protocol Buffers)
- Message brokers (RabbitMQ, Apache Kafka, NATS, ActiveMQ) — для асинхронной передачи сообщений
- GraphQL — для гибких запросов между сервисами
- WebSocket — для обмена сообщениями в реальном времени

4.6 Какие компоненты обычно включаются в микросервисную архитектуру?

- Сами микросервисы
- API gateway — единая точка входа для клиентов, маршрутизация запросов
- Система управления сервисами/service discovery (например, Consul, Eureka)
- Система балансировки нагрузки (Load Balancer)
- Система хранения и управления конфигурациями
- Мониторинг и логирование
- Система управления контейнерами (Docker, Kubernetes)
- Message broker (для обмена сообщениями между сервисами)
- База данных (обычно отдельная для каждого сервиса)

4.7 Как микросервисы обеспечивают автономность и независимость отдельных компонентов системы?

- Каждый сервис проектируется как самостоятельная единица, которая может разрабатываться, тестироваться, внедряться и масштабироваться независимо от других.
- Взаимодействие между сервисами осуществляется через стандартизированные API.
- Каждый микросервис может иметь свою базу данных, технологии, жизненный цикл.

4.8 Какие подходы к обеспечению безопасности можно использовать в микросервисной архитектуре?

- Использование безопасности на уровне API (авторизация/аутентификация — JWT, OAuth2)
- Валидация и ограничение доступа к данным на каждом сервисе (Scoping)
- Использование защищённых протоколов передачи данных (HTTPS, mTLS)
- Регулярное обновление и аудит библиотек
- Centralized identity provider (Single Sign-On)
- Сегментирование сети (например, используя сервис-меш: Istio)
- Мониторинг и обнаружение аномалий

4.9 Каким образом контейнеризация помогает в развертывании и управлении микросервисами?

- Упрощает переносимость, делая сервис независимым от окружения.
- Автоматизирует развертывание (оркестрация с помощью Docker, Kubernetes).
- Изолирует сервисы друг от друга и от операционной системы.
- Обеспечивает быстрое масштабирование и откат к предыдущим версиям.
- Облегчает обновление, развертывание и тестирование сервисов.

4.10 Как можно обеспечить высокую доступность и отказоустойчивость в микросервисной архитектуре?

- Дублирование сервисов и их автоматическое масштабирование (группы реплик)
- Использование распределённых кластеров и балансировщиков нагрузки
- Механизмы автоматического восстановления при сбоях (self-healing)
- Проработка автоматических откатов (rollback)
- Организация ручных и автоматических health check
- Проектирование "без единой точки отказа"
- Использование устойчивого message broker (например, Kafka, RabbitMQ с кластеризацией)
- Разделение и резервное копирование данных (backup)