Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное

учреждения высшего образования

«Национальный исследовательский технологический университет «МИСиС»

Курсовая Работа

по дисциплине

«Архитектурирование»

Направление подготовки:

09.03.01 «Информатика и вычислительная техника» Семестр 6

|  |  |
| --- | --- |
| Выполнил: | Проверил: |
| Пронин Г. Е.  (Ф.И.О. студента) | Романов К.А.  (Ф.И.О преподавателя) |
| БИВТ-22-СП-5  (№ группы) | (оценка) |
| 17.06.2025  (дата сдачи) | 17.06.2025  (дата проверки) |
| Подпись: | Подпись: |

Москва – 2025

## **Введение**

Целью данной работы является проектирование и реализация микросервисного приложения с использованием современных технологий: контейнеризации, кэширования и событийно-ориентированной архитектуры. Основная задача — создание трех сервисов (ProductService, CacheLayer и OrderService), взаимодействующих между собой через REST и Apache Kafka, с применением Redis для кэширования.

Используемые технологии:

* **Docker** – для упаковки и запуска микросервисов в изолированных контейнерах.
* **Redis** – в качестве распределённого кэша.
* **Apache Kafka** – для обмена сообщениями между микросервисами в модели producer/consumer.
* **.NET 9.0** – платформа для реализации логики микросервисов.

## **Теоретическая часть**

## **Контейнеризация**

## Контейнеризация — это технология, позволяющая упаковать приложение с его зависимостями в единый изолированный блок — контейнер. Это обеспечивает переносимость, масштабируемость и согласованность среды выполнения. Docker — ведущая технология контейнеризации. Использование контейнеров также позволяет легко воссоздавать окружение для CI/CD и упрощает управление зависимостями.

## **Кэширование**

## Кэширование — это процесс сохранения часто используемых данных для быстрого доступа. В данном проекте используется Redis, хранилище данных в памяти, которое позволяет значительно снизить нагрузку на ProductService. Использование кэша помогает сократить время отклика системы и уменьшает количество обращений к первичным источникам данных.

## **Event-Driven архитектура**

## Событийно-ориентированная архитектура (EDA) подразумевает, что компоненты системы обмениваются сообщениями-событиями. Kafka используется как брокер сообщений между OrderService и другими потребителями. Такой подход повышает масштабируемость и слабую связанность компонентов. Каждый сервис может публиковать или подписываться на события, не зная напрямую о других участниках системы.

## **Микросервисная архитектура**

## Микросервисы — это стиль архитектуры, при котором приложение состоит из небольших независимых сервисов, каждый из которых выполняет конкретную бизнес-функцию и взаимодействует с другими через API или сообщения. Это облегчает масштабирование, обновление и тестирование отдельных компонентов системы.

## **Observability: Prometheus, Grafana и Alertmanager**

## Observability (наблюдаемость) — это способность системы предоставлять информацию о своём внутреннем состоянии на основе внешних выходов. В современных распределённых системах крайне важно собирать метрики, логировать события и отслеживать аномалии.

## Prometheus — система мониторинга и сбора метрик. Сервисы публикуют метрики через HTTP, а Prometheus опрашивает их с заданной периодичностью. Примеры метрик: количество запросов, среднее время ответа, количество ошибок.

## Grafana — инструмент визуализации, подключаемый к Prometheus. Позволяет строить дашборды в реальном времени, отображать графики, таблицы и настраивать визуальные панели.

## Alertmanager — компонент, отвечающий за обработку и маршрутизацию алертов из Prometheus. Настраиваются правила, по которым в случае срабатывания триггера (например, "доступность сервиса ниже 90%") отправляется уведомление (например, в Telegram, Email, Slack и др.).

## Такой стек наблюдаемости позволяет не только анализировать текущее состояние системы, но и оперативно реагировать на отклонения и сбои.

## **CQRS и Event Sourcing**

## CQRS (Command Query Responsibility Segregation) — это архитектурный паттерн, разделяющий операции чтения и записи. В микросервисной архитектуре это позволяет использовать разные модели данных и даже разные хранилища для чтения и записи.

## Команды (Command) — операции, изменяющие состояние системы (например, создание заказа).

## Запросы (Query) — операции, возвращающие данные, не изменяя их (например, получение списка продуктов).

## В рамках CQRS API сервиса делится на два направления, и каждое может масштабироваться или оптимизироваться независимо.

## Event Sourcing (ES) — паттерн, при котором изменения состояния системы сохраняются в виде последовательности событий. Вместо хранения текущего состояния объекта, хранится список всех событий, приведших к этому состоянию.

## Например, создание заказа и последующие действия по его изменению/отмене записываются как события в Kafka. Эти события могут быть переиграны для восстановления полного состояния системы.

## Преимущества подхода:

## Возможность полной трассировки изменений (audit trail);

## Гибкость в построении дополнительных представлений данных;

## Надёжность восстановления после сбоев.

## В рамках проекта OrderService реализует паттерн CQRS: он только создаёт заказы (command) и публикует события, а потенциальный сервис для чтения (query) может быть реализован отдельно и подписан на Kafka для построения актуального состояния.

## **Проектирование архитектуры**

### **Общая архитектура**

Архитектура включает следующие сервисы:

* **ProductService**: предоставляет список продуктов.
* **CacheLayer**: кэширует результат запроса к ProductService.
* **OrderService**: создает заказ и публикует сообщение в Kafka.

### Контейнеризация

Каждый из сервисов упакован в отдельный Docker-контейнер. Docker Compose используется для оркестрации всех компонентов:

* Redis запускается как отдельный контейнер.
* Kafka и Zookeeper — контейнеры для поддержки брокера сообщений.

Dockerfile (универсальный для всех сервисов):

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build  
WORKDIR /app  
COPY \*.csproj ./  
RUN dotnet restore  
COPY . ./  
RUN dotnet publish -c Release -o out  
  
FROM mcr.microsoft.com/dotnet/aspnet:9.0  
WORKDIR /app  
COPY --from=build /app/out .  
ENTRYPOINT ["dotnet", "CacheLayer.dll"] # CacheLayer.dll заменяется на нужный сервис

Рисунок 1 Листинг Dockerfile

### **Кэширование**

В CacheLayer используется StackExchange.Redis. Данные продуктов кэшируются на 5 минут при первом запросе.

### **Event-Driven**

OrderService отправляет заказы в Kafka в топик orders с помощью библиотеки Confluent.Kafka. Потребителя для заказов в данной реализации **НЕ РЕАЛИЗОВАНО**.

### Бизнес-логика

* ProductService отдает список фиксированных продуктов (“Keyboard”, “Mouse”, “Monitor”, “CPU”).
* CacheLayer проверяет Redis и при отсутствии кэша запрашивает данные у ProductService.
* OrderService при POST-запросе создает заказ и отправляет его в Kafka.

## **Реализация**

### **Структура проекта**

ProjectRoot/  
├── docker-compose.yml  
├── ProductService/  
│ ├── Controllers/  
│ └── Program.cs  
├── CacheLayer/  
│ ├── Controllers/  
│ └── Program.cs  
└── OrderService/  
 ├── Controllers/  
 └── Program.cs

Рисунок 2 Структура проекта

### Контроллеры

CacheLayer/ProductsController.cs реализует проверку и установку кэша:

[HttpGet]  
public async Task<IActionResult> Get()  
{  
 var cachedData = await \_cache.GetStringAsync("products");  
  
 if (!string.IsNullOrEmpty(cachedData))  
 return Ok(JsonSerializer.Deserialize<object>(cachedData));  
  
 var client = \_httpClientFactory.CreateClient();  
 var response = await client.GetAsync("http://productservice:8080/products");  
 if (!response.IsSuccessStatusCode)  
 return StatusCode(500, "Не удалось получить продукты");  
  
 var content = await response.Content.ReadAsStringAsync();  
 await \_cache.SetStringAsync("products", content, new DistributedCacheEntryOptions  
 {  
 AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5)  
 });  
  
 return Ok(JsonSerializer.Deserialize<object>(content));  
}

Рисунок 3 Листинг контроллера CacheLayer/ProductsController.cs

OrderService/OrderController.cs реализует отправку заказа в Kafka:

[HttpPost]  
public async Task<IActionResult> CreateOrder()  
{  
 var order = new { OrderId = Guid.NewGuid(), Timestamp = DateTime.UtcNow };  
 var json = JsonSerializer.Serialize(order);  
  
 await \_producer.ProduceAsync("orders", new Message<Null, string> { Value = json });  
 return Ok(new { message = "Order sent to Kafka", order });  
}

Рисунок 3 Листинг контроллера OrderService/OrderController.cs

### **Ссылка на репозиторий**

<https://github.com/GlebPronin/ArchitecturirovanieKursovayaRabota>

## **Тестирование**

### CacheLayer





Рисунок 1 Пример результата GET-запроса из CacheLayer

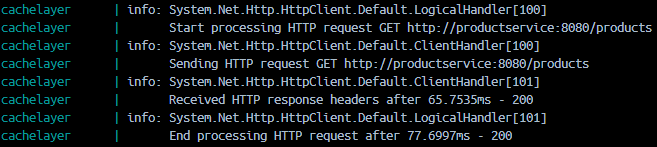


Рисунок 2 Обработка GET-запроса

* GET /products при первом запросе обращается к ProductService.
* GET /products при последующих — возвращает кэшированные данные.

### OrderService



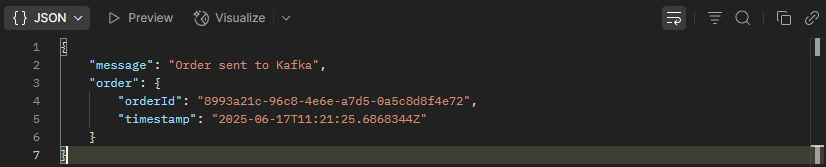


Рисунок 3 Пример результата POST-запроса из OrderService

POST /order — создаёт заказ, возвращает OrderId и сообщение.



Рисунок 4 Проверка в Kafka через kafka-console-consumer.sh

## Заключение

В ходе работы был реализован микросервисный проект с использованием Docker, Redis и Kafka. Были продемонстрированы ключевые архитектурные подходы: кэширование, контейнеризация и event-driven взаимодействие. Проект позволяет масштабировать функциональность и легко расширяется дополнительными потребителями Kafka.

Результаты:

* Повышена производительность за счёт кэширования.
* Продемонстрирована отправка событий в Kafka.
* Обеспечена модульность благодаря Docker.

## Список литературы

1. Redislabs. Redis documentation. <https://redis.io/docs/>
2. Docker Docs. <https://docs.docker.com/>
3. Kafka Documentation. <https://kafka.apache.org/documentation/>
4. Microsoft Learn: Distributed caching in ASP.NET Core. <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed>