МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ ПО ДИСЦИПЛИНЕ «Фреймворки back-end разработки»

для студентов направления подготовки 09.03.04 Программная инженерия
«Разработка и сопровождение программного обеспечения»

Ставрополь, 2026

Содержание

Введение

Современная back-end разработка предъявляет высокие требования к производительности, масштабируемости и отказоустойчивости серверных приложений. Традиционные многопоточные блокирующие модели, эффективные при умеренных нагрузках, начинают исчерпывать свои возможности при необходимости обслуживания десятков тысяч одновременно подключённых клиентов. В ответ на этот вызов сформировалась реактивная (неблокирующая) парадигма, которая опирается на событийно-ориентированный подход, асинхронный ввод-вывод и управление обратным давлением (backpressure). Освоение этих концепций и их практическая реализация являются важнейшей составляющей подготовки квалифицированных разработчиков программного обеспечения.

Настоящие методические указания предназначены для студентов направления 09.03.04 «Программная инженерия» (профиль «Разработка и сопровождение программного обеспечения») и охватывают полный цикл создания высокопроизводительных back-end систем — от первых шагов в реактивном программировании до построения законченного микросервисного решения на основе комбинации современных архитектурных паттернов.

Целью выполнения лабораторных работ является формирование у студентов системы практических навыков и компетенций в области:

- разработки неблокирующих веб-сервисов с использованием реактивных фреймворков (Spring WebFlux, FastAPI, gRPC на основе Project Reactor и аналогов);
- оценки производительности через нагрузочное тестирование и сравнение реактивного и блокирующего стеков;
- организации асинхронного доступа к реляционным базам данных на основе R2DBC (или аналогов в других языках);
- реализации gRPC-сервисов с бинарным протоколом Protocol Buffers;
- применения архитектурных паттернов CQRS и Event Sourcing с синхронными и асинхронными проекциями;
- использования брокеров сообщений (Redis Streams, RabbitMQ) для построения событийно-ориентированных систем;
- создания гибких API с помощью GraphQL и решения проблемы N+1 через DataLoader;
- контейнеризации разработанных компонентов с помощью Docker и Docker Compose;
- подготовки и защиты итогового проекта, демонстрирующего интеграцию всех изученных технологий.

Методические указания построены по модульному принципу: каждая лабораторная работа посвящена отдельной технологии или паттерну, содержит теоретическое обоснование, чёткое задание, примеры реализации на нескольких языках (Java/Kotlin, Python, Go, C#), порядок выполнения, контрольные вопросы и критерии оценки. Такой подход позволяет студенту выстроить индивидуальную траекторию обучения, выбрать наиболее подходящий язык и фреймворк, а также поэтапно наращивать сложность разрабатываемой системы.

Особое внимание уделяется экспериментальной части: практически в каждой работе предусмотрено нагрузочное тестирование, сбор метрик (RPS, перцентили задержки) и сравнение с альтернативными реализациями. Это формирует у студентов научно-инженерное мышление, где архитектурные решения принимаются не умозрительно, а на основе количественных измерений.

Итоговая лабораторная работа представляет собой сквозной проект, в котором студент должен самостоятельно спроектировать и реализовать реактивный бэкенд с CQRS, асинхронными проекциями, gRPC или GraphQL API, упаковать решение в контейнеры и обосновать его производительность. Защита проекта позволяет оценить не только технические навыки, но и способность аргументировать архитектурный выбор.

Методические указания составлены в соответствии с требованиями Федерального государственного образовательного стандарта высшего образования по направлению подготовки 09.03.04 «Программная инженерия» и могут использоваться как для аудиторной работы под руководством преподавателя, так и для самостоятельного изучения современных подходов к back-end разработке.

Лабораторная работа №1

Настройка реактивного проекта и базовый эндпоинт

1. Цель работы

Сформировать у студентов практические навыки создания неблокирующих веб-сервисов с использованием реактивного программирования и проведения первичного нагрузочного тестирования для сравнительного анализа производительности.

Знать:

- Принципы реактивного программирования (неблокирующий I/O, Event Loop, Backpressure).
- Отличия монопоточной неблокирующей модели от многопоточной блокирующей.
- Основные единицы реактивных потоков (на примере Project Reactor: Mono и Flux, аналоги в других стеках).

Уметь:

- Создавать проект на выбранном реактивном фреймворке.
- Реализовывать GET-эндпоинт, возвращающий асинхронный поток данных.
- Запускать нагрузочное тестирование с помощью wrk или bombardier и интерпретировать метрики RPS и задержки.

Владеть:

- Методикой сравнения блокирующего и неблокирующего серверов.
- Базовыми приёмами оценки эффективности архитектурных решений по данным нагрузочного тестирования.

2. Теоретическая часть

2.1. Реактивная модель против блокирующей

В традиционных блокирующих веб-фреймворках (Spring MVC, Flask, ASP.NET Core с синхронными действиями) каждый входящий запрос обрабатывается в отдельном потоке. Пока поток ожидает I/O (например, запрос к базе данных), он заблокирован и не может обслуживать другие запросы. При большом количестве одновременных соединений это приводит к исчерпанию пула потоков и резкому падению пропускной способности.

Реактивный подход основан на **неблокирующем I/O** и **событийном цикле (Event Loop)**. Малое число рабочих потоков (часто равное числу ядер CPU) асинхронно обрабатывает множество соединений. Когда операция I/O завершается, цикл вызывает

соответствующий callback, не простаивая в ожидании. Это позволяет обрабатывать тысячи параллельных запросов при минимальных накладных расходах.

2.2. Реактивные потоки и Backpressure

Реактивная парадигма опирается на спецификацию Reactive Streams, определяющую взаимодействие издателя (Publisher) и подписчика (Subscriber) с обязательной поддержкой обратного давления (backpressure). Backpressure даёт подписчику возможность сигнализировать издателю о своей готовности принять следующую порцию данных, предотвращая переполнение памяти.

Ключевые типы Project Reactor (Java):

- **Mono<T>** — асинхронный источник, возвращающий **0 или 1** элемент.
- **Flux<T>** — асинхронный источник, возвращающий **0..N** элементов, возможно бесконечный поток.

Аналоги в других экосистемах:

- **Kotlin + coroutines** — Flow<T> (холодный поток) и suspend-функции.
- **Python FastAPI** — асинхронные генераторы (async def с yield).
- **Go** — каналы (chan) и горутины, обеспечивающие конкурентную передачу данных.
- **C#** — IAsyncEnumerable<T> и каналы System.Threading.Channels.

2.3. Нагрузочное тестирование и метрики

Для сравнения производительности будут использованы утилиты wrk и bombardier. Основные метрики:

- **RPS (Requests Per Second)** — количество запросов, обрабатываемых сервером в секунду.
- **Latency (Avg / Stdev / Max)** — средняя, среднеквадратичное отклонение и максимальная задержка ответа.

Общепринятая методика: фиксированное количество соединений и длительность теста, повторение замеров для получения устойчивых значений.

3. Задание на лабораторную работу

Выполните следующие этапы:

1. **Разверните два HTTP-сервера** на разных портах (например, 8080 и 8081):
    - **Сервер А (реактивный)** — на выбранном реактивном фреймворке.
    - **Сервер Б (блокирующий)** — на том же языке, но с использованием традиционной многопоточной модели.

2. **Реализуйте на каждом сервере эндпоинт**
   GET /numbers?count=100
   Эндпоинт должен возвращать массив целых чисел от 1 до count в формате JSON.
   - Реактивная версия должна генерировать числа **потоково** (без накопления в список перед отправкой), эмулируя длительный процесс, например, с искусственной задержкой **10 мс** на каждом элементе (через асинхронный таймер, не блокируя поток).
   - Блокирующая версия делает то же самое, но с синхронной задержкой (sleep(10ms)), блокируя рабочий поток.
3. **Нагрузочное тестирование**
   Запустите утилиту wrk или bombardier для каждого сервера со следующими параметрами:
   - 100 одновременных соединений;
   - длительность теста 30 секунд;
   - эндпоинт /numbers?count=100.

Зафиксируйте RPS и среднюю задержку (Latency Avg). Проведите не менее трёх запусков для каждого сервера и усредните результаты.

4. **Проанализируйте результаты**
   - Постройте сравнительную таблицу: сервер, RPS (средний), Avg Latency.
   - Объясните, почему реактивный сервер показывает значительно более высокий RPS при тех же условиях нагрузки. Свяжите объяснение с моделью потоков.
5. **Подготовьте отчёт**
   Включите в отчёт:
   - Скриншоты запуска серверов и вывода утилит тестирования.
   - Таблицу с результатами.
   - Краткое текстовое обоснование разницы в производительности.

4. Вариативность по языкам и фреймворкам

Java (Spring WebFlux)

- **Реактивный сервер:** Spring Boot + WebFlux. Зависимости: spring-boot-starter-webflux, reactor-test.
  Контроллер:

java

```
@RestController
public class ReactiveController {
   @GetMapping(value = "/numbers", produces = MediaType.APPLICATION_NDJSON_VALUE)
   public Flux<Integer> numbers(@RequestParam(defaultValue = "100") int count) {
      return Flux.range(1, count)
            .delayElements(Duration.ofMillis(10));
   }
}
```

- **Блокирующий сервер:** Spring Boot + Web MVC (зависимость spring-boot-starter-web). Использовать Thread.sleep(10) в цикле.

Kotlin (Spring WebFlux + coroutines)

- **Реактивный:** spring-boot-starter-webflux, добавить kotlinx-coroutines-reactor. Контроллер:

```
kotlin
@GetMapping("/numbers")
suspend fun numbers(@RequestParam count: Int = 100): Flow<Int> = flow {
   for (i in 1..count) {
      delay(10)
      emit(i)
   }
}
```

- **Блокирующий:** Spring MVC с runBlocking { ... } и Thread.sleep.

Python (FastAPI)

- **Реактивный сервер:** FastAPI + uvicorn.
  Эндпоинт:

```
python
@app.get("/numbers")
async def numbers(count: int = 100):
   for i in range(1, count+1):
      await asyncio.sleep(0.01)
```

```
    yield i
```

Для отправки JSON-массива без накопления можно использовать StreamingResponse или NDJSON. Рекомендуется для чистоты сравнения возвращать NDJSON (поток объектов).

- **Блокирующий:** Flask или синхронный эндпоинт FastAPI (def numbers(): time.sleep(0.01)), запускать с uvicorn для одинакового рантайма.

Go

- **Реактивный:** Чистый Go, горутины + каналы. HTTP-сервер на net/http. Для потоковой записи использовать флэшинг: в обработчике задать заголовок Content-Type: application/x-ndjson и писать в w объекты,
  вызывая w.(http.Flusher).Flush() после каждого числа.
- **Блокирующий:** Простой сервер, в цикле time.Sleep(10ms) перед записью, без горутин, буферизация в слайс.

C# (ASP.NET Core)

- **Реактивный:** Minimal API или контроллер с IAsyncEnumerable<T>:

csharp

```
app.MapGet("/numbers", async (int count=100) => {
  async IAsyncEnumerable<int> StreamNumbers() {
    for (int i=1; i<=count; i++) {
      await Task.Delay(10);
      yield return i;
    }
  }
  return StreamNumbers();
});
```

- **Блокирующий:** синхронный контроллер с Thread.Sleep(10).

5. Порядок выполнения

1. **Установите необходимые инструменты:**
   - Среда разработки для выбранного языка (JDK 17+, Go 1.21+, Python 3.10+, .NET 8).

- wrk (https://github.com/wg/wrk)
  или bombardier (https://github.com/codesenberg/bombardier).
  Рекомендуется bombardier для кросс-платформенности.
- Docker (опционально для изоляции, можно обойтись локальным запуском).

2. **Создайте проект реактивного сервера.**
   Используйте примеры из раздела 4. Убедитесь, что при GET-запросе на http://localhost:8080/numbers?count=100 клиент получает поток данных. Для наглядности проверьте через curl, видна ли потоковая передача.
   Замечание: для корректного сравнения оба сервера должны возвращать одинаковый контент. Рекомендуется использовать формат NDJSON (каждое число на новой строке), чтобы клиент получал данные по мере генерации.
3. **Создайте проект блокирующего сервера** на порту 8081 с той же функциональностью, но синхронной задержкой. Проверьте работу.
4. **Запустите оба сервера одновременно** в разных терминалах.
5. **Выполните нагрузочные тесты:**

bash

Для реактивного сервера

bombardier -c 100 -d 30s http://localhost:8080/numbers?count=100

Для блокирующего

bombardier -c 100 -d 30s http://localhost:8081/numbers?count=100

Повторите каждый тест трижды. Запишите в отчёт средние значения RPS и Latency.

6. **Проведите анализ и оформление отчёта.**

6. Контрольные вопросы

1. Почему в блокирующей модели при 100 одновременных соединениях резко падает пропускная способность, если на сервере меньше 100 потоков?
2. Какую роль играет Event Loop в реактивном сервере? Может ли он заблокироваться?
3. Что случится, если в реактивном обработчике случайно вызвать блокирующую операцию (например, Thread.sleep)?
4. Объясните, почему при тестировании использовалась искусственная задержка в 10 мс на элемент. Как изменится разница в RPS, если задержку убрать совсем?

5. Каким образом реактивная модель обрабатывает обратное давление, если клиент читает медленно? Приведите пример с использованием Flux и медленного потребителя.

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
Реактивный сервер запущен и корректно отдаёт поток	3
Блокирующий аналог реализован верно	2
Проведены нагрузочные тесты (три итерации), получены воспроизводимые значения	2
Отчёт содержит сравнительную таблицу и аргументированное объяснение результатов	2
Ответы на контрольные вопросы при защите	1

Минимальный порог для зачёта — 6 баллов.

Методические рекомендации

- Для чистоты эксперимента оба сервера должны быть построены на одном языке, использовать одинаковый формат ответа и одинаковые операции.
- Если при тестировании реактивного сервера RPS оказывается ненамного выше блокирующего, проверьте, действительно ли блокирующая задержка реализована синхронно, а не асинхронно (например, Thread.sleep в Java вместо delayElements).
- При работе с wrk на macOS может потребоваться увеличение лимита открытых файлов (ulimit -n 10000).

Лабораторная работа №2

Реактивный REST API с R2DBC (без JPA)

1. Цель работы

Сформировать у студентов практические навыки построения полноценного неблокирующего CRUD-сервиса с использованием реактивного доступа к реляционной базе данных PostgreSQL. В ходе работы студенты исключат все блокирующие операции JDBC и ORM-подходы JPA/Hibernate, освоят асинхронное взаимодействие с БД и научатся контролировать неблокируемость всей цепочки запроса.

Знать:

- Архитектуру реактивного доступа к реляционным БД на основе спецификации R2DBC.
- Отличия Spring Data R2DBC от Spring Data JPA: отсутствие контекста персистентности, ленивых ассоциаций и автоматических запросов.
- Механизмы логирования SQL-запросов и анализа их количества в реактивных репозиториях.

Уметь:

- Создавать реактивные репозитории (ReactiveCrudRepository).
- Реализовывать все CRUD-операции без единого блокирующего вызова.
- Настраивать подключение к PostgreSQL через асинхронный драйвер (r2dbc-postgresql).
- Убеждаться в неблокирующем характере выполнения с помощью мониторинга потоков или нагрузочного тестирования.

Владеть:

- Методикой построения неблокирующей цепочки «Контроллер → Сервис → База данных» без скрытых блокировок.
- Приёмами контроля числа SQL-запросов и предотвращения проблемы N+1 в реактивной среде.

2. Теоретическая часть

2.1. Почему не JPA? Отказ от блокирующего наследия

Предшествующие курсы знакомили студентов с JPA и Hibernate — мощными, но **блокирующими** инструментами. Под капотом они используют JDBC, который синхронно ожидает ответа от базы данных, занимая поток на всё время запроса. Для

реактивных систем такой подход неприемлем: каждый блокирующий вызов «замораживает» Event Loop, уничтожая все преимущества неблокирующей модели.

Кроме того, JPA привносит:

- **Контекст персистентности (L1 cache)** — потенциальные проблемы с памятью и согласованностью в асинхронном потоке.
- **Ленивые ассоциации (lazy loading)** — источники скрытых дополнительных запросов (N+1), которые в реактивном коде трудно контролировать.
- **Автоматическое управление состоянием сущностей** — несовместимо с функциональным стилем и иммутабельными структурами данных, принятыми в реактивном программировании.

Решением является **R2DBC** (Reactive Relational Database Connectivity) — спецификация, определяющая асинхронный, неблокирующий доступ к SQL-базам данных. Драйверы R2DBC (например, для PostgreSQL) построены на базе Netty, не занимают потоки на время ожидания ответа и полностью совместимы с реактивными фреймворками.

2.2. Spring Data R2DBC и реактивные репозитории

Spring Data R2DBC предоставляет знакомую модель репозиториев, но без магии JPA. Основные отличия:

- **Нет EntityManager и кэша первого уровня.** Каждый вызов метода репозитория сразу транслируется в SQL и выполняется.
- **Нет ленивых ассоциаций.** Все связи (@OneToMany, @ManyToOne) не поддерживаются автоматически. При необходимости разработчик явно пишет агрегирующий запрос или использует дополнительные методы репозитория.
- **Прозрачное управление запросами.** Вы сами контролируете, какой SQL будет сгенерирован или написан вручную.

Интерфейс ReactiveCrudRepository<T, ID> предоставляет базовые операции: save(), findAll(), findById(), deleteById() и т.д., возвращающие Mono или Flux.

2.3. Как убедиться, что запросы не блокируют Event Loop?

Самый прямой способ — наблюдать за потоками, в которых выполняется код. В Spring WebFlux на базе Reactor Netty все операции по умолчанию исполняются в потоках Event Loop (обычно их число равно количеству ядер). Если блокировок нет, **любое количество параллельных запросов к БД будет обслужено без увеличения пула потоков** и без падения RPS вплоть до предела базы данных.

Методы проверки:

1. **Логирование имени потока** в обработчике и в репозитории (добавление log() в цепочку Reactor или временное внедрение Thread.currentThread().getName()).

Студент увидит, что один и тот же поток-демон обрабатывает подготовку и завершение запроса, а ожидание ответа от БД не удерживает его.

2. **Нагрузочное тестирование с параллельными запросами** и наблюдение за RPS. На блокирующем JDBC при 200 одновременных соединениях RPS резко падает; на R2DBC — держится стабильно.
3. **Специализированные тесты с медленным запросом,** например, SELECT 1, pg_sleep(2) внутри одного эндпоинта. Остальные «быстрые» эндпоинты при этом должны сохранять высокую пропускную способность, доказывая, что Event Loop не заблокирован.

2.4. Логирование SQL и подсчёт запросов

Для предотвращения избыточных обращений к базе данных студент должен уметь видеть каждый выполняемый SQL. Настройка:

yaml

```
logging:
  level:
    io.r2dbc.postgresql.QUERY: DEBUG
    io.r2dbc.postgresql.PARAM: DEBUG
```

Это выводит в лог все подготовленные запросы и их параметры, позволяя вручную или с помощью счётчика проверить, что на одну операцию CRUD действительно пришёлся ровно один SQL-запрос.

3. Задание на лабораторную работу

3.1. Подготовка стенда

- Разверните PostgreSQL (локально или в Docker). Рекомендуемый docker-compose:

yaml

```
services:
  postgres:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: reactive_crud
      POSTGRES_USER: student
      POSTGRES_PASSWORD: student
```

ports:

- "5432:5432"

- Создайте таблицу products:

sql

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  price DECIMAL(10,2) NOT NULL
);
```

3.2. Реализуйте реактивный CRUD-сервис (Сервер A)

Используя **только R2DBC**, без JPA/Hibernate, создайте реактивный REST API для управления сущностью «Продукт».
Требуемые эндпоинты:

- GET /products — вернуть все продукты (Flux).
- GET /products/{id} — получить продукт по id (Mono).
- POST /products — создать новый продукт (Mono), тело {"name":"...","price":...}.
- PUT /products/{id} — обновить существующий (Mono).
- DELETE /products/{id} — удалить продукт (Mono<Void>).

Обязательные условия:

- На уровне репозитория использовать ReactiveCrudRepository<Product, Integer>.
- Никаких блокирующих вызовов в цепочке (нельзя использовать block(), Thread.sleep() и т.п.).
- Включить логирование SQL на уровне DEBUG, чтобы фиксировать все запросы.
- Для операции обновления (PUT) должен выполняться ровно один SQL-запрос (проверяется через лог).
- Реализовать проверку существования продукта перед обновлением/удалением, возвращая 404 при отсутствии. При этом количество запросов должно быть минимальным (желательно 1 запрос с UPDATE/DELETE ... WHERE id = ..., и проверять количество изменённых строк).

3.3. Доказательство неблокируемости (Сервер A)

Добавьте в сервис эндпоинт GET /products/slow, который внутри вызывает хранимую функцию или анонимный блок, выполняющий SELECT 1, pg_sleep(3) (задержка 3 секунды) и после этого возвращает фиксированный текст. Это эмулирует длительный запрос к БД.

- Через отдельное нагрузочное тестирование (инструмент из ЛР1) с 50 одновременными соединениями к /products/slow и одновременно к /products покажите, что «быстрые» запросы списка продуктов продолжают обслуживаться без задержек, а Event Loop не исчерпывается.

Методика проверки:
Запустите в фоне бесконечные запросы к /products/slow (например, 20 соединений), и одновременно выполните бенчмарк /products?count=100 (как в ЛР1). Сравните полученный RPS с эталонным значением, измеренным при отключённом /products/slow. Снижение RPS не должно превышать 10–15%, что доказывает неблокируемость.

3.4. (Опционально) Контраст с блокирующим JDBC

Для углублённого понимания можете на том же языке создать маленький сервис на JDBC/JPA с аналогичным эндпоинтом /products/slow и показать, что параллельный быстрый запрос проседает практически до нуля. Этот пункт не обязателен, но приветствуется при защите.

3.5. Отчёт

Отчёт должен содержать:

- Схему таблицы БД.
- Код контроллера, репозитория (основные моменты).
- Скриншоты логов SQL для каждого типа операций (создание, чтение, обновление, удаление) с ручным подсчётом числа запросов.
- Результаты нагрузочного теста с медленным эндпоинтом: сравнительная таблица RPS для /products при работающем /products/slow и без него.
- Выводы о неблокируемости R2DBC и отсутствии JPA-магии.

4. Вариативность по языкам и фреймворкам

Язык	Реактивный фреймворк / БД доступ	Рекомендации
Java	Spring WebFlux + Spring Data R2DBC (драйвер r2dbc-postgresql)	Использовать ReactiveCrudRepository. Для обновления с проверкой — @Query("UPDATE products SET … WHERE id = :id") возвращает Mono<Integer> (rows

Язык	Реактивный фреймворк / БД доступ	Рекомендации
		affected).
Kotlin	Spring WebFlux + coroutines + Spring Data R2DBC	Репозиторий может использовать suspend-функции и Flow. Подключение аналогично Java.
Python	FastAPI + asyncpg или SQLAlchemy async	asyncpg — самый низкоуровневый и быстрый драйвер. Для CRUD придётся писать сырые SQL-запросы через asyncpg.Connection. Аналог репозитория — собственный класс.
Go	net/http + pgxpool (драйвер pgx)	Нативные горутины. Запросы к базе через pgxpool.Pool.QueryRow / Query. Никаких ORM, всё вручную. Потоковая отправка JSON через json.Encoder с флэшингом.
C#	ASP.NET Core Minimal API + Npgsql (асинхронный)	NpgsqlDataSource или NpgsqlConnection с async-методами. Для «репозитория» использовать await using и асинхронные методы.

Общее для всех стеков:

- Должен использоваться исключительно асинхронный драйвер БД.
- Не допускается применение классических синхронных драйверов (JDBC, psycopg2, database/sql с блокирующим драйвером, SqlClient синхронный).
- Медленный запрос реализовать через выполнение SELECT 1, pg_sleep(3) (или хранимую функцию).

5. Порядок выполнения

1. **Подготовьте инфраструктуру:** PostgreSQL в Docker, создайте базу и таблицу products.
2. **Создайте проект реактивного сервера** (пример для Java):

- Зависимости: spring-boot-starter-webflux, spring-boot-starter-data-r2dbc, r2dbc-postgresql, lombok (опционально).
- Конфигурация application.yml:

yaml

```
spring:
  r2dbc:
    url: r2dbc:postgresql://localhost:5432/reactive_crud
    username: student
    password: student
logging:
  level:
    io.r2dbc.postgresql.QUERY: DEBUG
    io.r2dbc.postgresql.PARAM: DEBUG
```

3. **Опишите сущность** Product (id, name, price). Для Spring Data R2DBC это обычный POJO, аннотации @Id, @Column (из пакета org.springframework.data.relational.core.mapping).
4. **Создайте репозиторий** interface ProductRepository extends ReactiveCrudRepository<Product, Integer>. При необходимости добавьте кастомные методы с @Query.
5. **Реализуйте сервисный слой** (необязательно отдельно, но рекомендуется), инкапсулирующий логику проверки и обновления. Используйте операторы Reactor (switchIfEmpty, flatMap, filter) для построения неблокирующей цепочки.
6. **Реализуйте контроллер**, возвращающий Mono<ResponseEntity<…>> для каждого эндпоинта, с корректной обработкой 404.
7. **Добавьте эндпоинт** /products/slow:

java

```
@GetMapping("/products/slow")
public Mono<String> slow() {
   return databaseClient.sql("SELECT 1, pg_sleep(3)")
                .fetch()
                .rowsUpdated()
                .thenReturn("slept 3s");
```

}

(или через @Query в репозитории)

8. **Проверьте логи SQL** для всех операций: создайте, прочитайте, обновите и удалите продукт, убедитесь, что каждое действие порождает ровно один запрос.
9. **Проведите нагрузочный тест неблокируемости:**
    - Запустите сервер.
    - В первом терминале запустите «медленные» запросы: bombardier -c 20 -d 60s http://localhost:8080/products/slow
    - Во втором терминале запустите замер быстрого эндпоинта: bombardier -c 50 -d 30s http://localhost:8080/products
    - Повторите тест быстрого эндпоинта **без** нагрузки медленными запросами. Сравните RPS.
10. **Оформите отчёт** согласно разделу 3.5.

6. Контрольные вопросы

1. Почему JPA и Hibernate считаются «нереактивными»? Какие именно механизмы внутри них вызывают блокировку?
2. Как в Spring Data R2DBC выполнить кастомный UPDATE-запрос и узнать, были ли изменены строки? Как на основе этого вернуть 404, если продукт не существует?
3. Что выводится в логах при включении io.r2dbc.postgresql.QUERY: DEBUG? Опишите, как по этим логам можно подсчитать количество запросов.
4. Если в реактивный метод вставить вызов Thread.sleep(100), что произойдёт с пропускной способностью сервера? Почему?
5. Объясните результаты нагрузочного теста: почему медленный запрос с pg_sleep не замедлил остальные запросы? Какие системные ресурсы использовала база данных, а какие — приложение?
6. Есть ли в Spring Data R2DBC аналог @ManyToMany? Если нет, как правильно реализовать связь «многие ко многим» в реактивном стиле, чтобы избежать лишних запросов?

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
Корректно работающий реактивный CRUD (все 5 эндпоинтов)	3
Все операции выполняются без блокировок (подтверждено логами потоков или нагрузочным тестом)	2
Включено логирование SQL, в отчёте приведены скриншоты с анализом количества запросов	2
Реализован эндпоинт /products/slow и проведён сравнительный нагрузочный тест неблокируемости	2
Ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- Для проверки, что обновление выполняется одним запросом, избегайте паттерна «найти, потом обновить». Используйте UPDATE … SET … WHERE id = :id RETURNING * или проверяйте количество изменённых строк.
- При нагрузочном тесте с медленными запросами обратите внимание, что у PostgreSQL есть собственный пул соединений; если он исчерпается, то даже реактивный клиент будет ждать. Поэтому выбирайте разумное число соединений (до 100 обычно хватает).
- Для убедительности можно добавить в контроллер логирование имени потока: log.info("Handling request in thread: {}", Thread.currentThread().getName()) и убедиться, что оно одно и то же для всех запросов (в пределах одного event loop).
- Студентам на Python/Go придётся реализовывать «репозиторий» вручную; это допустимо, главное — синхронные драйверы не применять.

Лабораторная работа №3

gRPC-сервер и клиент на Protocol Buffers

1. Цель работы

Сформировать у студентов практические навыки построения высокопроизводительного межсервисного взаимодействия с использованием gRPC и Protocol Buffers. Студенты научатся описывать контракты сервисов в .proto-файлах, генерировать клиентский и серверный код, реализовывать gRPC-сервер и консольный клиент, а также экспериментально сравнить эффективность бинарного протокола gRPC с традиционным JSON/REST.

Знать:

- Архитектуру gRPC: Protocol Buffers как язык описания контрактов, HTTP/2 как транспорт, multiplexing, server push.
- Типы gRPC-методов: унарный (unary), серверный поток, клиентский поток, двунаправленный поток.
- Механизм сериализации protobuf и его отличия от JSON.

Уметь:

- Описывать сервис в .proto-файле, задавать сообщения и RPC.
- Генерировать код для выбранного языка с помощью protoc и плагинов.
- Реализовывать gRPC-сервер (сервисную логику).
- Создавать консольный gRPC-клиент, выполняющий вызовы.
- Проводить сравнительное нагрузочное тестирование и измерять размер сообщений.

Владеть:

- Методикой выбора протокола взаимодействия на основе количественных метрик (задержка, пропускная способность, размер wire-формата).
- Базовыми приёмами отладки gRPC с помощью инструментов типа grpcurl или grpc_cli.

2. Теоретическая часть

2.1. gRPC и Protocol Buffers

gRPC — высокопроизводительный RPC-фреймворк с открытым исходным кодом, разработанный Google. Он использует Protocol Buffers (protobuf) как язык описания интерфейсов (IDL) и бинарный формат сериализации, HTTP/2 в качестве транспорта.

Преимущества перед REST/JSON:

- **Бинарная сериализация:** protobuf кодирует данные компактнее текстового JSON (в 3–10 раз), что снижает нагрузку на сеть и процессор.
- **Строгая типизация и контракт:** .proto-файл служит единым источником истины для сервера и клиента, исключая неоднозначности.
- **Мультиплексирование запросов по одному TCP-соединению** (HTTP/2), что уменьшает накладные расходы на установление соединений и задержки.
- **Встроенная поддержка потоковой передачи** (стриминг) в обоих направлениях.
- **Генерация клиентского и серверного кода** для десятков языков из одного .proto, что ускоряет разработку.

В рамках лабораторной работы мы будем использовать **унарный RPC** (один запрос – один ответ), как наиболее близкий к REST, но позже (ЛР7) изучим потоковые варианты.

2.2. Метрики для сравнения

- **Размер сообщения** (payload size) — количество байт, передаваемых по сети. Измеряется для protobuf-сериализованного объекта и его JSON-представления (с включёнными именами полей или без них). В реальности REST использует JSON, а gRPC — protobuf, но для чистоты эксперимента можно сравнить сериализацию одного и того же объекта в protobuf и JSON.
- **Пропускная способность (RPS)** — количество запросов, обрабатываемых сервером за секунду при заданной нагрузке.
- **Задержка (Latency)** — время от отправки запроса до получения ответа.

2.3. Инструменты для тестирования и отладки

- **grpcurl** — аналог curl для gRPC, позволяет отправлять запросы в командной строке без написания клиента.
- **BloomRPC / Postman** (с поддержкой gRPC) — GUI-клиенты.
- **protoc** — компилятор Protocol Buffers для генерации кода.
- Для нагрузочного тестирования gRPC: ghz (https://ghz.sh) — специализированный инструмент, аналогичный wrk/bombardier, но понимающий protobuf.

3. Задание на лабораторную работу

Выполните следующие этапы:

1. **Выберите предметную область.**
   Рекомендуется использовать ту же сущность «Продукт», что и в ЛР2 (id, name, price), но можно взять «Пользователь» (id, name, email). Договоритесь с преподавателем.
2. **Опишите сервис в** .proto**-файле.**
   Создайте файл, например product_service.proto, содержащий:
   - Сообщение Product с полями int32 id, string name, double price.
   - Сообщение ProductRequest с полем int32 id.
   - Сообщение ProductListResponse с полем repeated Product products.
   - Сервис ProductService с унарными RPC:
     - GetProduct(ProductRequest) returns (Product)
     - ListProducts(google.protobuf.Empty) returns (ProductListResponse)
     - CreateProduct(Product) returns (Product)
     - UpdateProduct(Product) returns (Product)
     - DeleteProduct(ProductRequest) returns (google.protobuf.Empty)
       Опционально можно добавить простое поле-селектор для демонстрации сериализации.
3. **Сгенерируйте серверный и клиентский код** для вашего языка согласно таблице в разделе 4.
4. **Реализуйте gRPC-сервер (Сервер gRPC).**
   Сервер должен хранить продукты в оперативной памяти (например, ConcurrentHashMap в Java, dict в Python, sync.Map в Go) и реализовывать все RPC. Подключение к базе данных в этой работе не требуется; акцент на протоколе и сериализации.
5. **Реализуйте консольный gRPC-клиент.**
   Клиент должен предоставлять меню (или аргументы командной строки) для вызова каждого RPC и выводить ответы. Также клиент должен измерить время выполнения каждого вызова (round-trip time) при многократном повторении.
6. **Создайте REST-аналог (Сервер REST)** с идентичной функциональностью:
   - CRUD для продуктов, JSON-формат.
   - На том же языке и фреймворке (можно использовать реактивный подход из ЛР2, можно синхронный для простоты, но для чистоты сравнения желательно, чтобы обе версии были сходны по окружению).
   - API должно быть максимально похоже на gRPC-контракт: поля называются так же, структура ответов та же.

7. **Сравнительный анализ:**
    - **Размер сообщений:** сериализуйте один объект Product (id=1, name="Laptop", price=1299.99) в protobuf и в JSON (с отступами и без). Запишите размеры в байтах. Рассчитайте сжатие (protobuf bytes vs JSON bytes).
    - **Задержка и RPS:** Проведите нагрузочное тестирование обоих серверов.
        - Для gRPC: используйте ghz с тем же .proto:

bash

ghz --insecure --proto product_service.proto --call ProductService/GetProduct -d '{"id":1}' -c 100 -n 10000 localhost:50051

(для создания — через ghz с CreateProduct и т.п.)

        - Для REST: используйте bombardier или wrk, как в ЛР1–ЛР2.
        - Протестируйте операции GetProduct (по id) и ListProducts (без параметра, возвращает все, скажем, 10 предзагруженных продуктов). Одинаковые условия: 100 соединений, 30 секунд, если возможно.
    - Сравните RPS и среднюю задержку. Объясните разницу.
8. **Подготовьте отчёт**, содержащий:
    - Текст .proto-файла.
    - Ключевые фрагменты кода сервера и клиента.
    - Скриншоты работы клиента (вывод в консоль).
    - Таблицу размеров сообщений (protobuf vs JSON).
    - Таблицу результатов нагрузочного тестирования (RPS, Latency) для gRPC и REST по каждой операции.
    - Выводы о преимуществах и возможных недостатках gRPC.

4. Вариативность по языкам и фреймворкам

Язык	Рекомендуемый стек	Особенности
Java	gRPC-Java, protobuf-gradle-plugin или protoc, Netty-транспорт	Использовать protobuf-maven-plugin или gradle. Сервер — ServerBuilder.forPort(50051).addService(new

Язык	Рекомендуемый стек	Особенности
		ProductServiceImpl()).build(). Клиент — ManagedChannelBuilder.forAddress("localhost", 50051).usePlaintext().build() и сгенерированные стабы.
Kotlin	То же, что Java + kotlinx.coroutines	gRPC-Kotlin предоставляет корутин-обёртки для стабов: ProductServiceGrpcKt.ProductServiceCoroutineStub. Более идиоматичный код.
Python	grpcio, grpcio-tools (для генерации из .proto)	Генерация через python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. product_service.proto. Сервер на grpc.aio для асинхронности. Клиент может быть синхронным или async.
Go	google.golang.org/grpc, protoc-gen-go, protoc-gen-go-grpc	Генерация: protoc --go_out=. --go-grpc_out=. product_service.proto. Сервер: grpc.NewServer(), регистрация pb.RegisterProductServiceServer. Клиент: grpc.Dial.
C#	Grpc.AspNetCore (сервер), Grpc.Net.Client (клиент)	.proto компилируется через Google.Protobuf и Grpc.Tools. Сервер — встроен в ASP.NET Core. Клиент — GrpcChannel.ForAddress("http://localhost:50051").

Общие требования:

- gRPC-сервер должен слушать порт 50051 (или 50052, если рядом REST).
- клиент должен работать отдельно, без веб-интерфейса, демонстрируя вызовы.

5. Порядок выполнения

1. **Установите инструменты:**
    - Компилятор protoc (https://github.com/protocolbuffers/protobuf/releases) и плагины для вашего языка (см. официальную документацию gRPC).

- Для нагрузочного тестирования gRPC:
  установите ghz (https://ghz.sh/docs/install).
- Для отладки: grpcurl (https://github.com/fullstorydev/grpcurl) опционально.

2. **Создайте каталог проекта.** Разместите .proto-файл в подкаталоге proto/ (или в корне). Продублируйте структуру из примеров для вашего языка.
3. **Сгенерируйте код.**
   Для Java (Gradle): добавить плагин com.google.protobuf, выполнить gradle generateProto. Для Python: запустить protoc. Результаты генерации появятся в целевой папке. Убедитесь, что созданы серверные и клиентские классы.
4. **Реализуйте gRPC-сервер.**
   - Создайте класс, наследующий сгенерированный базовый класс сервиса (например, ProductServiceGrpc.ProductServiceImplBase в Java).
   - Переопределите методы. Для хранения используйте потокобезопасную структуру (ConcurrentHashMap, sync.Map, asyncio.Lock-protected dict).
   - Запустите сервер и проверьте через grpcurl:

bash

grpcurl -plaintext -d '{"id":1}' localhost:50051 ProductService/GetProduct

5. **Реализуйте REST-сервер** на порту 8080 (или другом) с эквивалентными эндпоинтами:
   - GET /products/{id}
   - GET /products
   - POST /products
   - PUT /products/{id}
   - DELETE /products/{id}
     Хранилище может быть тем же, что и для gRPC (в едином процессе можно объединить для удобства, но для нагрузочного тестирования лучше раздельные процессы, чтобы не мешали друг другу). Достаточно in-memory.
6. **Реализуйте консольный клиент.** Пример меню (консольный ввод):

text

1. Get product by id

2. List all products

3. Create product

4. Update product

5. Delete product

0. Exit

После каждого вызова выведите ответ и затраченное время. Для замеров используйте системный таймер (System.nanoTime, time.perf_counter, time.Now).

7. **Измерение размера сообщений.**
   Создайте в клиенте или отдельном скрипте сериализацию экземпляра Product (id=1, name="Laptop", price=1299.99) в protobuf (вызовите toByteArray()) и в JSON (любой библиотекой). Запишите длину полученных массивов. Сравните. Желательно также сравнить JSON с минимальным форматированием (без пробелов).
8. **Нагрузочное тестирование.**
   - Подготовьте данные: 10 продуктов в памяти.
   - Для gRPC: ghz с опцией --insecure, указать .proto, метод, данные. Для ListProducts (Empty) данные {}. Выполните по 30-секундному тесту с 100 соединениями.
   - Для REST: bombardier с теми же параметрами,
     URL: http://localhost:8080/products/1 и http://localhost:8080/products.
   - Запишите результаты в таблицу.
9. **Анализ и отчёт.**
   Объясните, почему gRPC демонстрирует бóльшую пропускную способность и меньшую задержку. Укажите, что причина не только в бинарном формате, но и в HTTP/2 мультиплексировании, отсутствии разбора текстовых заголовков и т.д. Но для нашего простого сценария основное влияние оказывает именно сериализация и размер передаваемых данных.

6. Контрольные вопросы

1. Какие типы RPC поддерживает gRPC? Приведите примеры из реальной архитектуры, где применим каждый из них.
2. Почему gRPC работает быстрее REST/JSON при равных условиях? Назовите не менее трёх причин.
3. Чем отличается генерация кода для сервера и клиента? Для чего нужны стабы и скелеты?
4. Каковы ограничения gRPC? В каких сценариях REST может быть предпочтительнее?
5. Что произойдёт, если изменить .proto-файл (добавить поле) и не перегенерировать клиентский код? Как protobuf обеспечивает обратную совместимость?

6. Как бы вы реализовали аутентификацию в gRPC-сервисе? Какие механизмы предоставляет фреймворк?
7. В чём разница между grpcurl и ghz? Для каких целей используется каждый из них?

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
.proto-файл корректен, генерируется код без ошибок	1
gRPC-сервер реализован, все RPC работают (проверено grpcurl/клиентом)	3
Консольный клиент выполняет все вызовы и измеряет время	2
REST-аналог реализован и работоспособен	1
Проведено сравнение размеров сообщений (protobuf vs JSON) с выводами	1
Проведено нагрузочное тестирование обоих серверов, представлены сопоставимые таблицы	1
Отчёт и ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- Если ghz недоступен для вашей ОС, можно использовать написанный на коленке нагрузочный генератор на том же языке с многопоточными/асинхронными запросами к gRPC, но ghz предпочтительнее, так как даёт структурированный вывод.
- При сравнении размеров сообщений обязательно укажите, с форматированием JSON или без. Для чистоты сравнения используйте JSON без пробелов (как это обычно бывает в продакшене).
- Желательно, чтобы gRPC и REST серверы запускались на одном компьютере, но на разных портах, и тесты проводились по очереди для исключения влияния фоновых процессов.

- При реализации сервера на Python обязательно используйте асинхронный вариант grpc.aio, чтобы сохранить реактивность (если выбран Python). Для синхронного сравнения это не критично, но хорошая практика.
- Для Go каналы не обязательны для унарных RPC, достаточно мьютекса для защиты мапы. Клиент может быть простым синхронным.
- В отчёте обязательно приведите скриншот успешного вызова через grpcurl или консольный клиент с выводом времени.

Лабораторная работа №4

Реализация CQRS: разделение Command и Query

1. Цель работы

Сформировать у студентов практические навыки применения паттерна CQRS (Command Query Responsibility Segregation) на уровне одного микросервиса. Студенты научатся разделять модели данных для операций изменения состояния (Command) и операций чтения (Query), а также реализовывать синхронную проекцию для немедленного поддержания read-модели в актуальном состоянии при выполнении команд.

Знать:

- Суть паттерна CQRS, его мотивацию и применимость.
- Различия между моделью команд (нормализованная, оптимизированная под запись) и моделью запросов (денормализованная, оптимизированная под чтение).
- Понятие проекции и способы её синхронизации: синхронное обновление в рамках единой транзакции.

Уметь:

- Проектировать две модели данных: Command Model и Query Model.
- Реализовывать Command-обработчики, осуществляющие запись в нормализованное хранилище и немедленное обновление read-модели в той же транзакции.
- Реализовывать Query-обработчики, читающие данные из денормализованного представления без прямого обращения к Command-таблицам.

Владеть:

- Навыками разделения ответственности команд и запросов на уровне кода и структуры проекта.
- Подходом к оценке преимуществ и недостатков синхронной проекции в сравнении с асинхронными решениями (которые будут изучены далее).

2. Теоретическая часть

2.1. Паттерн CQRS

CQRS (Command Query Responsibility Segregation) — архитектурный паттерн, предписывающий разделение моделей для обработки операций, изменяющих состояние (Commands), и операций, возвращающих данные (Queries). В классической архитектуре обе задачи решаются через единую модель предметной области, что приводит к

компромиссам: модель должна одновременно удовлетворять и требованиям целостности при записи, и требованиям удобства выборок и агрегации при чтении. CQRS снимает это противоречие.

Ключевые принципы:

- **Command сторона** отвечает за бизнес-логику, валидацию и сохранение состояния. Данные хранятся в высоконормализованном виде (реляционные таблицы), оптимизированном для обеспечения непротиворечивости и атомарных изменений.
- **Query сторона** обслуживает запросы на чтение без какой-либо бизнес-логики, используя денормализованное представление (read-модель), спроектированное под конкретные пользовательские интерфейсы или отчёты. Запросы могут выполняться предельно быстро, без сложных JOIN-ов и агрегаций.

В простейшей реализации (синхронная проекция) при выполнении команды атомарно обновляется и Command-хранилище, и Query-представление — обычно в рамках одной транзакции базы данных.

2.2. Модели данных и синхронная проекция

Command Model: таблицы в третьей нормальной форме, без избыточности. Например, для складского учёта — products(id, name), inventory(product_id, quantity). Здесь легко поддерживать целостность (один товар — одна запись количества).

Query Model (Read Model): денормализованная таблица (или материализованное представление), которая содержит все необходимые для UI поля в одной строке: product_summary(product_id, name, quantity). При запросе списка товаров с остатками достаточно простого SELECT * FROM product_summary.

При выполнении команды «Оприходовать товар» Command-обработчик в одной транзакции:

- обновляет inventory (или вставляет, если товара нет);
- обновляет product_summary (пересчитывает количество или создаёт новую запись).

Это гарантирует сильную согласованность: read-модель всегда отражает актуальное состояние. Однако синхронная проекция связывает Command и Query, что может стать узким местом при высокой нагрузке на запись. Тем не менее, это отличная отправная точка для понимания CQRS, перед переходом к асинхронным проекциям через очереди сообщений в последующих работах.

2.3. Когда применять CQRS

- Когда требования к чтению и записи сильно различаются (сложные отчёты против простых команд).

- При необходимости масштабировать чтение независимо от записи (но пока у нас один сервис, это демонстрация разделения ответственности).
- Для упрощения модели: Command-сервис не думает о формате выдачи, Query-сервис не содержит бизнес-логики.

3. Задание на лабораторную работу

Вам необходимо реализовать сервис управления запасами на складе (или товарами в интернет-магазине) с использованием паттерна CQRS. Обязательно разделите модели и логику на две части:

Command сторона (write API):

- POST /inventory/receive — оприходование товара: принимает productId, quantity. Увеличивает остаток на складе.
- POST /inventory/ship — отгрузка товара: принимает productId, quantity. Уменьшает остаток, если достаточно (иначе ошибка).
- Внутреннее хранилище: нормализованная таблица inventory с полями product_id, quantity.

Query сторона (read API):

- GET /inventory?productId=... — получить текущий остаток по одному товару.
- GET /inventory/all — получить список всех товаров с остатками (денормализованное представление).
- Read-модель: таблица inventory_summary с полями product_id, name, quantity. Наполняется синхронно при каждой команде.

Требования:

- Использовать **реактивный** стек и R2DBC (или асинхронный драйвер для выбранного языка) для взаимодействия с БД, как в ЛР2. Все операции неблокирующие.
- Command-обработчики должны в одной транзакции обновлять обе таблицы (либо обе обновляются, либо ни одна).
- Query-обработчики читают **только** из inventory_summary, не обращаясь к основной таблице inventory.
- Код должен быть чётко структурирован: отдельные пакеты/модули для command и query, разные сервисы или хотя бы разные методы, явно обозначающие принадлежность.

- Необходимо включить логирование SQL, чтобы подтвердить, что при команде выполняются два INSERT/UPDATE (или один MERGE), а при запросе — простой SELECT.

Опционально: Добавить эндпоинт GET /inventory/history/{productId}, который бы показывал журнал операций (таблица inventory_log). Команды при выполнении также добавляют запись в лог. Это демонстрирует, что Command-сторона может иметь дополнительные таблицы, невидимые для Query.

4. Вариативность по языкам и фреймворкам

Для всех языков обязателен асинхронный доступ к БД и транзакции.

Язык	Command / Query реализация	Особенности работы с транзакциями
Java	Spring WebFlux + Spring Data R2DBC. Создать отдельные классы: InventoryCommandService, InventoryQueryService. Command-сервис использует @Transactional (R2DBC transactions) или TransactionOperator.	TransactionOperator предоставляет реактивную транзакцию: operator.transactional(status -> inventoryRepo.update(...).then(inventorySummaryRepo.update(...))).
Kotlin	Аналогично Java, с корутинами. Можно использовать coTransaction из spring-tx.	Корутинная транзакция: transactional { updateInventory(); updateSummary() }.
Python	FastAPI + asyncpg или SQLAlchemy async. Command-функции вызывают async with connection.transaction(): await connection.execute(...). Query-функции читают напрямую.	Транзакции через connection.transaction(). Рекомендуется выделить классы InventoryCommandRepo, InventoryQueryRepo.
Go	net/http + pgxpool. В обработчиках command выполняется pool.Begin(ctx) и два запроса внутри defer tx.Rollback(). Query обработчики делают просто pool.QueryRow.	Транзакция — стандартный pgx.Tx.
C#	ASP.NET Core Minimal API + Npgsql. Command-эндпоинты используют await using var transaction = await connection.BeginTransactionAsync(), выполняют два запроса, затем CommitAsync(). Query — одиночный запрос.	Транзакция через DbTransaction.

Общее требование: Использовать ту же базу данных (PostgreSQL), что и в ЛР2. Допускается использовать in-memory хранилище (например, ConcurrentMap) для имитации, но **желательно** делать с БД, чтобы наглядно продемонстрировать синхронную

проекцию через транзакции. Если in-memory — требуется имитация транзакционности (синхронизация блокировками).

5. Порядок выполнения

1. **Подготовка БД (если используется PostgreSQL):**

sql

```
CREATE TABLE inventory (
    product_id INTEGER PRIMARY KEY,
    quantity INTEGER NOT NULL CHECK (quantity >= 0)
);


CREATE TABLE inventory_summary (
    product_id INTEGER PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    quantity INTEGER NOT NULL
);
```

Опционально:

sql

```
CREATE TABLE inventory_log (
    id SERIAL PRIMARY KEY,
    product_id INTEGER NOT NULL,
    change INTEGER NOT NULL,
    reason VARCHAR(50) NOT NULL,
    changed_at TIMESTAMP DEFAULT NOW()
);
```

2. **Создайте проект реактивного сервера.** Подключите зависимости для асинхронной работы с PostgreSQL (как в ЛР2). Настройте подключение.
3. **Определите структуру Command и Query моделей.**
    - Command-репозиторий: методы updateQuantity(productId, delta), insertRecord(...).

- Query-репозиторий: методы findByProductId, findAll.
- В Java/Kotlin: можно использовать отдельные интерфейсы ReactiveCrudRepository для каждой таблицы.

4. **Реализуйте Command-сервис:**
   - receive(productId, name, quantity): в транзакции:
     1. Проверить, есть ли товар в inventory. Если нет — добавить с quantity=0.
     2. Увеличить quantity на полученное значение.
     3. Обновить/вставить inventory_summary: установить name (может обновиться) и quantity новое.
   - ship(productId, quantity):
     1. Проверить достаточно ли остатка. Если нет — ошибка.
     2. Уменьшить quantity в inventory.
     3. Обновить inventory_summary.quantity.
   - Убедитесь, что все операции внутри транзакции.
5. **Реализуйте Query-сервис:**
   - getProduct(productId): простой запрос к inventory_summary.
   - getAll: SELECT * FROM inventory_summary.
     Никаких соединений с inventory.
6. **Реализуйте REST-контроллеры** (можно в рамках одного приложения, но с чётким разделением эндпоинтов).
   - POST /inventory/receive
   - POST /inventory/ship
   - GET /inventory?productId=... (или /inventory/{productId})
   - GET /inventory/all
7. **Тестирование:**
   - Выполните несколько команд прихода и расхода.
   - Получите список всех товаров через query-эндпоинт. Убедитесь, что количество корректное.
   - Просмотрите логи SQL (включите DEBUG как в ЛР2). Подтвердите, что чтение идёт из одной таблицы, запись — в две.

8. **Нагрузочное тестирование (по желанию):** Сравните производительность чтения из денормализованной summary с JOIN-запросом к нормализованным таблицам (можно отдельным эндпоинтом). Оцените разницу в скорости.
9. **Подготовьте отчёт:**
    - Диаграмма или описание архитектуры CQRS вашего сервиса.
    - Ключевые фрагменты кода команды (с транзакцией) и запроса.
    - Скриншоты логов SQL, демонстрирующие операции.
    - Результаты проверки консистентности.

6. Контрольные вопросы

1. В чём заключается принцип разделения ответственности команд и запросов в CQRS?
2. Почему Command-сторона использует нормализованную модель, а Query-сторона денормализованную?
3. Что произойдёт, если при синхронной проекции обновление read-модели выполнится без транзакции? Приведите пример нарушения консистентности.
4. В чём недостатки синхронной проекции по сравнению с асинхронной (через очередь сообщений)? Когда синхронный подход оправдан?
5. Как бы вы масштабировали Query-сторону независимо от Command-стороны?
6. Может ли Query-сторона использовать другую СУБД (например, MongoDB)? Какие требования это предъявляет к проекции?
7. Нужно ли дублировать бизнес-логику валидации в Query-модели? Обоснуйте.

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
Разделены Command и Query модели, эндпоинты работают	3
Command-операции выполняются в транзакции и атомарно обновляют обе таблицы	2
Query-эндпоинты читают исключительно read-модель, без JOIN с Command-	2

Критерий	Баллы
таблицами	
Логи SQL подтверждают корректность (отдельные выборки, одна транзакция при изменении)	1
Чистота кода, чёткое разделение пакетов/модулей	1
Ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- Если используется in-memory (ConcurrentHashMap) вместо БД, необходимо реализовать транзакционность вручную: при команде сначала залочить объект, изменить обе map, затем разблокировать. Это менее наглядно, допускается только в исключительных случаях.
- Для демонстрации преимуществ CQRS можно сравнить скорость «тяжёлого» JOIN-запроса к нормализованным таблицам (если бы Query использовало их) и простого SELECT из inventory_summary. Сравнение улучшит отчёт.
- При работе с Java/Kotlin аккуратно используйте TransactionOperator, чтобы транзакция охватывала все реактивные цепочки. Для Python connection.transaction() обязательно открывать через async with.
- Обратите внимание студентов, что в реальном проекте read-модель может обновляться асинхронно через брокеры сообщений, но это тема следующих ЛР (Event Sourcing, очереди). Здесь мы сознательно используем синхронную проекцию для простоты и надёжности.

Лабораторная работа №5

Event Sourcing: хранение событий и восстановление состояния

1. Цель работы

Освоить принципиально иной подход к хранению состояния — Event Sourcing (ES). Научиться вместо сохранения текущего состояния фиксировать поток событий (изменений), а состояние восстанавливать путём их последовательного воспроизведения (replay). Студенты поймут фундаментальные отличия ES от традиционного CRUD, преимущества (аудит, временные запросы, отладка) и недостатки (объём данных, необходимость снапшотов).

Знать:

- Суть Event Sourcing: хранение всех событий, а не текущего состояния.
- Понятие агрегата, событий предметной области, потока событий (event stream).
- Механизм replay и его вычислительные особенности.
- Понятие снапшота (snapshot) как оптимизации (будет рассмотрено в будущих работах).

Уметь:

- Проектировать события, описывающие бизнес-изменения.
- Реализовывать Event Store (в оперативной памяти или с персистентностью в Redis).
- Разрабатывать команды, порождающие события, и применять их к агрегату.
- Восстанавливать текущее состояние агрегата из цепочки событий.
- Строить REST API, которое при запросе выполняет replay для получения баланса.

Владеть:

- Методикой выбора между классическим хранением состояния и Event Sourcing на основе требований к аудиту, временным запросам и сложности.

2. Теоретическая часть

2.1. Event Sourcing: идея и преимущества

В традиционных CRUD-системах мы храним текущее состояние. Например, баланс счёта — это просто цифра в таблице accounts. При изменении баланса мы перезаписываем значение, теряя информацию о том, как оно было достигнуто. Event Sourcing предлагает сохранять все произошедшие с объектом события (Domain Events). Текущее состояние вычисляется на лету путём применения всех событий в хронологическом порядке к начальному состоянию.

Ключевые достоинства:

- **Полная история изменений** — аудит, расследования, отладка становятся простыми.
- **Возможность временных запросов:** восстановить состояние системы на любой момент времени.
- **Причинно-следственный анализ:** легко понять, почему система пришла к данному состоянию.
- **Гибкость построения проекций:** можно создать любые read-модели (как в CQRS) из общего потока событий, даже спустя время, «переиграв» историю.

Недостатки:

- Объём хранимых событий может значительно превышать размер текущего состояния.
- Длительное время replay при большом накоплении событий (решается снапшотами — периодическим сохранением состояния агрегата и replay с последнего снапшота).
- Необходимость механизма оптимистического контроля версий при конкурентных изменениях.

2.2. Агрегат, события и поток

В Event Sourcing центральное понятие — **агрегат** (Aggregate), кластер объектов, состояние которого меняется через команды и фиксируется событиями. Каждый агрегат идентифицируется уникальным ID (например, номером счёта) и имеет поток событий (event stream).

События — неизменяемые объекты, описывающие факт свершившегося действия. Пример для банковского счёта:

- AccountCreated (initialBalance)
- MoneyDeposited (amount, afterDepositBalance) — можно хранить результирующий баланс для удобства, но это опционально.
- MoneyWithdrawn (amount, afterWithdrawalBalance)

Команда (Command) — запрос на изменение. Обработчик команды:

1. Загружает текущее состояние (replay всех событий агрегата).
2. Проверяет бизнес-правила (достаточно ли средств для снятия).
3. Создаёт одно или несколько новых событий.
4. Сохраняет события в Event Store.

Event Store управляет потоками событий. Обычно это append-only хранилище: новые события дописываются в конец, изменение прошлых событий невозможно.

2.3. Replay — восстановление состояния

При старте системы или обработке команды состояние агрегата восстанавливается функцией apply(state, event) -> state. Начальное состояние известно. Далее одно за другим применяются все события из потока агрегата.

Например:

python

```
def replay(events):
  state = None
  for event in events:
    if event.type == 'AccountCreated':
      state = Account(event.accountId, event.initialBalance)
    elif event.type == 'MoneyDeposited':
      state.balance += event.amount
    elif event.type == 'MoneyWithdrawn':
      state.balance -= event.amount
  return state
```

В ходе лабораторной работы мы используем in-memory Event Store (ConcurrentHashMap по id счёта → список событий) или Redis Streams / Lists для персистентности. Redis обеспечивает долговременное хранение и удобен для будущей интеграции с очередями.

3. Задание на лабораторную работу

Реализуйте банковский сервис (управление счётом) на основе Event Sourcing. Требования:

1. **Агрегат «Счёт» (Account)** — имеет идентификатор (UUID или номер), имя владельца, текущий баланс (вычисляется).
2. **События:**
    - AccountCreated — создание счёта с начальным балансом (может быть 0).
    - MoneyDeposited — пополнение счёта.
    - MoneyWithdrawn — списание денег.
      В событиях достаточно хранить сумму изменения и, возможно, итоговый баланс после применения (для контроля), но итоговый баланс не обязан

быть частью события — он вычисляется при replay. Мы рекомендуем не хранить баланс в событиях, чтобы продемонстрировать полный пересчёт. Однако можно добавить и баланс для перекрёстной проверки.

3. **Event Store:**
    - Хранит для каждого счёта упорядоченный список событий.
    - Поддерживает операцию append(accountId, events) и loadEvents(accountId).
    - **Базовый вариант:** in-memory (потокобезопасная структура, например, ConcurrentHashMap<String, CopyOnWriteArrayList<Event>> в Java, sync.Map в Go, asyncio.Lock вокруг словаря в Python).
    - **Продвинутый вариант (рекомендуется):** Redis — использовать структуру List (LPUSH/RPUSH/LRANGE) или Streams. Redis Streams особенно интересны, так как станут основой для ЛР6–ЛР7.
4. **Обработчики команд:**
    - createAccount(accountId, owner, initialBalance) → событие AccountCreated + сохранение в Event Store.
    - deposit(accountId, amount) → replay, проверка, новое событие MoneyDeposited, сохранение.
    - withdraw(accountId, amount) → replay, проверка достаточности баланса, событие MoneyWithdrawn, сохранение.
      Все операции должны быть атомарны на уровне одного счёта (но в in-memory версии достаточно синхронизации).
5. **REST API:**
    - POST /accounts — создать счёт (тело: {"accountId":"...","owner":"...","initialBalance":0}).
    - POST /accounts/{id}/deposit (тело: {"amount":100}).
    - POST /accounts/{id}/withdraw (тело: {"amount":50}).
    - GET /accounts/{id} — возвращает текущий баланс и владельца, **вычисляемые через replay событий**. При каждом запросе загружается полный список событий и применяется. (В учебных целях — ок, в реальности добавили бы кэш или снапшот).
    - (Опционально) GET /accounts/{id}/events — возвращает журнал событий для аудита.
6. **Демонстрация:**
    - Создайте счёт, проведите несколько операций, затем запросите баланс — balance должен быть корректен.

- Выведите в лог последовательность событий и промежуточные состояния при каждом запросе (для отладки).

7. **Отчёт:**
    - Схема событий (можно в таблице).
    - Ключевые фрагменты кода: Event Store, replay-функция, обработчики команд.
    - Скриншоты запросов и ответов, показывающие изменение баланса.
    - Сравнение с традиционным CRUD: что было бы иначе, если бы мы просто хранили balance в базе и меняли его.
    - Выводы: когда целесообразно применять Event Sourcing.

4. Вариативность по языкам и фреймворкам

Язык	Хранение событий (Event Store)	Особенности реализации
Java	ConcurrentHashMap<String, List<Event>> или Redis (Spring Data Redis Reactive)	Replay через Flux.fromIterable(events).reduce(initialState, ...). REST — Spring WebFlux.
Kotlin	Аналогично Java, возможно использование Mutex для защиты	coroutines, Flow для replay. Redis — lettuce реактивный клиент.
Python	asyncio.Lock + dict[str, list[Event]] или Redis через aioredis / redis.asyncio	Асинхронный replay через цикл for event in events: state = apply(state, event). FastAPI для API.
Go	sync.Map со слайсом событий или Redis (go-redis)	Replay — простая итерация. REST — стандартный net/http. Горутины для фоновых операций не требуются.
C#	ConcurrentDictionary<string, List<Event>> или Redis (StackExchange.Redis)	Replay — Aggregate LINQ. ASP.NET Core Minimal API.

Требование совместимости с реактивным стеком: Несмотря на то, что Event Store может быть in-memory, все операции должны быть неблокирующими (для языков, где это применимо — Java, Kotlin, Python async, C# async). Go может использовать синхронный доступ, т.к. горутины дешёвые, но защита общей структуры обязательна.

Redis как Event Store (рекомендованный продвинутый вариант):

- Использовать структуру LIST: ключ events:{accountId}, команды LPUSH events:{id} '{json_event}' (новые события в начало) и LRANGE events:{id} 0 -1 для загрузки всех событий. Можно и RPUSH, главное сохранять порядок.
- Альтернатива: Redis Streams (XADD и XRANGE), что ближе к реальной промышленной практике. Поток позволяет подписываться (consumer groups) — это будет полезно в ЛР6 при построении проекций.

5. Порядок выполнения

1. **Определите структуру событий** (общую для всех языков). Пример для JSON (в Redis):

json

{"type": "AccountCreated", "accountId": "a1", "owner": "Alice", "initialBalance": 0, "timestamp": "..."}

{"type": "MoneyDeposited", "accountId": "a1", "amount": 100, "timestamp": "..."}

{"type": "MoneyWithdrawn", "accountId": "a1", "amount": 30, "timestamp": "..."}

При in-memory хранении можно использовать объекты языка.

2. **Реализуйте Event Store.**
    - **In-memory:** создайте потокобезопасный словарь Map<String, List<Event>> (или синхронизированный доступ через блокировки). Операции добавления и чтения должны быть защищены от конкурентных изменений. Для replay — возврат копии списка (чтобы гарантировать неизменность).
    - **Redis:** настройте подключение, выберите LIST или Streams. Для replay загружайте все записи.
3. **Реализуйте функцию** apply(state, event) (для replay), которая создаёт начальное состояние из AccountCreated и обновляет баланс на последующих событиях.
4. **Создайте сервисный слой с командами:**

java

public Mono<Account> createAccount(String id, String owner, double initialBalance) {

```
    Event event = new AccountCreated(id, owner, initialBalance);
    return eventStore.append(id, event).thenReturn(apply(null, event));
}
public Mono<Account> deposit(String id, double amount) {
    return eventStore.loadEvents(id)
        .reduce(initialState(), (acc, evt) -> apply(acc, evt))
        .flatMap(current -> {
            Event event = new MoneyDeposited(amount);
            return eventStore.append(id, event)
                .thenReturn(apply(current, event));
        });
}
```

Аналогично для withdraw с проверкой current.balance >= amount.

5. **Реализуйте REST контроллер**, маршруты как в задании.
6. **Демонстрация работы:**
    - Создайте счёт.
    - Внесите 100, затем 50.
    - Снимите 30.
    - Получите баланс — должно быть 120.
    - Получите список событий (опциональный эндпоинт) и убедитесь, что все три события сохранены.
7. **Включите логирование** (или выведите в консоль) процесса replay при каждом обращении к балансу, чтобы видеть, что состояние действительно вычисляется каждый раз.
8. **Напишите отчёт.**

6. Контрольные вопросы

1. Чем принципиально отличается Event Sourcing от традиционного CRUD-подхода? Приведите примеры ситуаций, где ES даёт преимущества.
2. Почему при восстановлении состояния (replay) не следует изменять уже сохранённые события?

3. Каковы основные проблемы производительности Event Sourcing? Как их решить (снапшоты, материализованные представления)?
4. Как при Event Sourcing обеспечить атомарность обработки конкурентных запросов к одному агрегату?
5. Может ли поток событий неограниченно расти? Как изменится время восстановления счёта с миллионом транзакций? Предложите пути оптимизации.
6. Каким образом Event Sourcing упрощает внедрение CQRS? (Подсказка: единый источник событий для создания любых read-моделей)
7. Можно ли удалить событие из Event Store, например, по требованию GDPR? Обсудите последствия.
8. Что произойдёт, если изменилась бизнес-логика команды, а события были записаны со старой логикой? Как восстановить правильное состояние?

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
Корректно реализован Event Store (append + load)	2
Реализована функция replay, правильно восстанавливающая баланс	2
Обработчики команд используют replay, проверяют бизнес-правила	2
REST API работоспособно, баланс вычисляется через replay	2
Отчёт содержит демонстрацию и обоснованные выводы	1
Ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- Для простоты в in-memory Event Store не требуется сохранять события на диск, но подчеркните студентам, что в реальности хранилище должно быть персистентным.

- При использовании Redis поясните, что в ЛР5 можно обойтись List, а в ЛР6 Redis Streams позволит организовать публикацию событий для асинхронных проекций. Таким образом, эта работа готовит почву для последующих.
- Для наглядности выводите в консоль промежуточные состояния при replay.
- При проверке атомарности в in-memory версии используйте блокировки (ReentrantLock, asyncio.Lock, sync.Mutex).
- Обратите внимание студентов на важность порядка событий: он должен строго сохраняться.
- Не используйте термин «снапшот» как обязательный элемент, но упомяните, что это тема будущих оптимизаций.

Лабораторная работа №6

Проекция (Projection) для Event Sourcing + асинхронная очередь

1. Цель работы

Сформировать у студентов практические навыки построения асинхронных проекций в системах, использующих Event Sourcing. В ходе работы будет освоена публикация событий в очередь сообщений (Redis Streams или RabbitMQ) и реализован отдельный потребитель (consumer), обновляющий read-модель вне критического пути обработки команды. Студенты проведут сравнительный анализ синхронной (ЛР4) и асинхронной проекций, поймут компромисс между консистентностью и производительностью.

Знать:

- Принципы асинхронных проекций в CQRS/Event Sourcing.
- Архитектурные различия синхронного и асинхронного обновления read-модели.
- Понятие eventual consistency (конечная согласованность) и его влияние на пользовательский опыт.
- Базовые концепции брокеров сообщений: производитель (producer), потребитель (consumer), поток (stream), очередь (queue), подтверждение (acknowledgement).

Уметь:

- Настраивать Redis Streams или RabbitMQ для передачи событий.
- Декомпозировать сервис на Command-обработчик (публикующий события) и проекционный consumer.
- Реализовывать идемпотентную обработку событий при обновлении read-модели.
- Измерять задержку eventual consistency и сравнивать пропускную способность командного API при синхронной и асинхронной проекциях.

Владеть:

- Методикой выбора режима обновления проекций на основе нефункциональных требований.
- Навыками построения event-driven микросервисов с гарантией доставки сообщений.

2. Теоретическая часть

2.1. От синхронной проекции к асинхронной

В лабораторной работе №4 мы реализовали CQRS с синхронной проекцией: команда (оприходование/отгрузка) атомарно изменяла нормализованную таблицу inventory и денормализованную inventory_summary в рамках одной транзакции. Это обеспечило строгую консистентность (read-модель всегда актуальна), но связало производительность записи с обновлением read-модели. При росте сложности проекций или их количества время выполнения команды увеличивается, что может стать узким местом.

Асинхронная проекция разрывает эту жёсткую связь:

- Command-обработчик (после успешного сохранения событий в Event Store) лишь публикует событие в очередь.
- Обработка команды возвращает ответ клиенту немедленно, не дожидаясь обновления read-модели.
- Отдельный процесс (или поток) — consumer — подписан на очередь, получает события и применяет их к read-моделям. Обновление происходит с некоторой задержкой (порядка миллисекунд-секунд), что означает **конечную согласованность (eventual consistency)**.

Это даёт преимущества:

- **Разделение записи и чтения:** Command API может быть очень быстрым и масштабироваться независимо.
- **Устойчивость к сбоям:** Сбой consumer не влияет на приём команд; при восстановлении consumer обработает накопленные события.
- **Множественные проекции:** Одно событие может обновлять несколько read-моделей (например, для разных представлений), просто добавив дополнительных потребителей.
- **Технологическая гибкость:** Read-модель может храниться в другой СУБД, оптимизированной под запросы.

Плата за это — временная рассогласованность: пользователь, прочитавший данные сразу после команды, может увидеть устаревшее значение. Поэтому асинхронные проекции применяются там, где допустима задержка в секунду-две.

2.2. Очереди сообщений: Redis Streams и RabbitMQ

Для доставки событий от Command к Consumer мы используем посредника — очередь сообщений. В рамках курса предложены два варианта.

Redis Streams (рекомендуется как продолжение ЛР5, если там использовался Redis):

- Легковесный, не требует отдельной инсталляции (Redis уже используется как Event Store / кэш).
- Работает как append-only лог. Команды: XADD — добавить событие, XREAD / XREADGROUP — читать.
- Поддержка групп потребителей (consumer groups) гарантирует, что каждое событие будет обработано ровно один раз в группе (при правильном подтверждении).
- Прост в настройке, высокая производительность.

RabbitMQ:

- Полнофункциональный брокер сообщений, работающий по протоколу AMQP 0-9-1.
- Обменники (exchange), очереди (queue), привязки (binding).
- Надёжная доставка с подтверждениями, гибкая маршрутизация.
- Требует отдельного развёртывания (можно в Docker), но даёт мощные возможности.

В лабораторной работе мы сосредоточимся на Redis Streams как естественном развитии ЛР5. При желании студент может использовать RabbitMQ.

2.3. Идемпотентность consumer и порядок обработки

При асинхронной обработке возможны повторные доставки одного и того же сообщения (из-за сбоев, перезапусков). Поэтому consumer должен быть **идемпотентным**: повторное применение того же события не должно приводить к некорректному состоянию read-модели. Обычно это достигается хранением в consumer последнего обработанного идентификатора события и пропуском дубликатов.

Кроме того, события одного агрегата должны обрабатываться строго последовательно, чтобы сохранить причинно-следственный порядок. В Redis Streams для этого можно использовать ключ * при XADD (автоинкрементный ID) и XREADGROUP с правильным consumer-именем, а в RabbitMQ — ключ маршрутизации по ID агрегата.

3. Задание на лабораторную работу

Вам необходимо расширить систему из ЛР5 (банковский счёт) или ЛР4 (склад), переведя обновление read-модели на асинхронный режим с использованием очереди сообщений.

3.1. Базовые требования

1. **Command-сервис (API записи):**
    - Оставляет неизменной логику команд (создание счёта, пополнение, снятие) с сохранением событий в Event Store (in-memory или Redis).
    - После успешного добавления события(й) в Event Store **публикует** событие в очередь (Redis Stream или RabbitMQ) и сразу возвращает успешный ответ клиенту.
    - **Не** выполняет синхронного обновления read-модели.
2. **Event Bus (очередь):**
    - Должна быть развёрнута либо Redis (рекомендуется), либо RabbitMQ (в Docker).
    - Для Redis Streams: поток events:{aggregateId} или общий поток events с ключом агрегата в поле.

3. **Projection Consumer (фоновый сервис):**
    - Постоянно (или по мере появления) читает события из очереди.
    - По каждому полученному событию обновляет read-модель (материализованное представление). Например, для счёта — таблица account_summary(accountId, owner, balance). Для склада — inventory_summary(productId, name, quantity).
    - Consumer должен быть идемпотентным (проверка дубликатов по eventId).
    - Для простоты можно хранить read-модель в оперативной памяти (ConcurrentHashMap) или в той же БД, но в отдельной таблице. Обязательно продемонстрировать, что она обновляется асинхронно.
4. **Query API (чтение):**
    - Предоставляет актуальное (с точки зрения read-модели) состояние. Например, GET /accounts/{id}/balance или GET /inventory/{productId}.
    - Должен обращаться **исключительно** к read-модели, без прямого вызова Event Store или replay.
5. **Сравнение с синхронной проекцией:**
    - Проведите нагрузочное тестирование командного эндпоинта (deposit или withdraw) и зафиксируйте RPS и задержку ответа.
    - Сравните с показателями синхронной проекции (если есть ЛР4) или с гипотетическим синхронным вариантом (когда обновление read-модели выполняется в той же транзакции). Ожидается, что асинхронный вариант даст бóльшую пропускную способность, но при этом read-модель будет отставать.
    - Измерьте задержку распространения (propagation latency): время между получением ответа команды и моментом, когда обновлённое значение становится видимым через Query API. Усредните по нескольким попыткам.

3.2. Сценарий демонстрации

1. Запустить Command-сервер, Consumer и Query-сервер (можно в одном процессе, но consumer должен быть отдельным потоком/асинхронным циклом, не блокирующим запрос).
2. Создать счёт (или продукт) с начальным балансом 0.
3. Выполнить команду deposit 100.
4. Немедленно (в цикле с малым интервалом) опрашивать Query API, пока баланс не станет равным 100. Записать время, затраченное на появление актуального значения.

5. Повторить для нескольких операций, вычислить среднюю задержку.
6. Выполнить нагрузочный тест командного API (100 соединений, 30 секунд) и зафиксировать RPS. Повторить тест для синхронной версии (если реализована), сравнив результаты.

3.3. Отчёт

- Архитектурная схема (producer → broker → consumer → read model → query API).
- Ключевые фрагменты кода: публикация события, consumer (запуск, чтение, обновление), идемпотентность.
- Скриншоты или логи, подтверждающие задержку eventual consistency.
- Таблица с результатами нагрузочного тестирования и сравнение RPS.
- Выводы о преимуществах и недостатках асинхронной проекции.

4. Вариативность по языкам и фреймворкам

Язык	Инструменты для очереди	Рекомендации
Java	Redis: spring-boot-starter-data-redis-reactive + ReactiveRedisTemplate для Streams. RabbitMQ: spring-cloud-starter-stream-rabbit или reactor-rabbitmq	Consumer можно реализовать через @Scheduled или реактивный слушатель с XREADGROUP. Для идемпотентности хранить Set<String> processedEventIds в Redis или ConcurrentHashMap.
Kotlin	То же, что Java, либо lettuce с корутинами.	Аналогично, можно использовать корутины для фонового consumer (например, launch{ while(true){ ... } }).
Python	Redis: redis.asyncio (aioredis) для Streams. RabbitMQ: aio-pika.	Consumer — асинхронная задача, запускаемая через asyncio.create_task. Для FastAPI можно использовать @app.on_event("startup").

Язык	Инструменты для очереди	Рекомендации
Go	Redis: go-redis с поддержкой Streams. RabbitMQ: github.com/rabbitmq/amqp091-go.	Consumer — горутина с бесконечным циклом XREADGROUP. Защита от дублей через sync.Map.
C#	Redis: StackExchange.Redis (Streams). RabbitMQ: RabbitMQ.Client или MassTransit.	Consumer — IHostedService (BackgroundService) в ASP.NET Core. Идемпотентность через ConcurrentDictionary.

Общие указания:

- Redis Streams: Producer использует XADD streamName "*" field value. Consumer — XREADGROUP GROUP mygroup consumer1 BLOCK 1000 STREAMS streamName > для новых сообщений и 0 для ожидающих.
- RabbitMQ: Объявить exchange типа fanout или topic, очередь, привязанную к exchange. Producer публикует в exchange. Consumer слушает очередь с autoAck=false, подтверждая после обработки.

5. Порядок выполнения

1. **Подготовьте инфраструктуру:**
    - Redis или RabbitMQ (в Docker). Например, для Redis:

bash

```
docker run -d --name redis -p 6379:6379 redis:7-alpine
```

    - Базовый проект с Event Sourcing из ЛР5 (или реализовать заново счёт/склад).
2. **Модифицируйте Command-обработчик:**
    - После eventStore.append() (успешного сохранения событий) выполните публикацию в очередь. Например, в Redis Streams:

java

```
reactiveRedisTemplate.opsForStream()
   .add(StreamRecords.newRecord().ofObject(event).withStreamKey(streamKey));
```

    - Убедитесь, что команда не ждёт завершения обновления read-модели.

3. **Создайте Consumer:**
    - Для Java: асинхронный бесконечный цикл, читающий из Stream с XREADGROUP. При получении события десериализовать, обновить read-модель (например, accountSummary.update(event)), затем подтвердить (XACK).
    - Для Python: запустить asyncio.create_task с аналогичной логикой.
    - Реализуйте пропуск дубликатов: сохраняйте ID обработанного события и проверяйте перед применением.
4. **Постройте read-модель:** in-memory ConcurrentHashMap<String, AccountSummary> или таблицу БД. Для упрощения можно in-memory.
5. **Реализуйте Query API:** эндпоинт, возвращающий состояние из read-модели.
6. **Эксперимент по задержке:**
    - Запустите систему.
    - Отправьте команду депозита.
    - В цикле с интервалом 10 мс опрашивайте Query-эндпоинт, засекайте время до появления нового баланса. Запишите результат.
7. **Нагрузочное тестирование команд:**
    - Используйте bombardier или ghz для gRPC. Для простоты REST команд.
    - Протестируйте POST /accounts/{id}/deposit с пулом 100 соединений в течение 30 секунд. Запишите RPS.
    - Если есть синхронный аналог (из ЛР4 или специально сделанный), протестируйте его и сравните RPS. Ожидается рост пропускной способности.
8. **Оформите отчёт.**

6. Контрольные вопросы

1. В чём ключевое отличие асинхронной проекции от синхронной? Какие метрики улучшаются, какие ухудшаются?
2. Что такое «конечная согласованность» (eventual consistency)? Приведите пример ситуации, когда она неприемлема.
3. Как обеспечить идемпотентность consumer при возможной повторной доставке сообщений?
4. Почему важен порядок обработки событий одного агрегата? Как этого добиться в Redis Streams или RabbitMQ?

5. Как повлияет на пропускную способность командного API добавление нескольких потребителей разных проекций?
6. Что произойдёт, если consumer упал? Будут ли потеряны события? Как долго будет накапливаться отставание?
7. Сравните Redis Streams и RabbitMQ для задачи асинхронной проекции. В каких случаях вы выберете каждый из них?
8. Как бы вы реализовали снапшоты (snapshot) в Event Sourcing, чтобы ускорить восстановление состояния, и как это связано с проекциями?

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
Command API публикует события в очередь, асинхронная проекция работает	3
Consumer корректно обновляет read-модель, идемпотентность	2
Query API читает исключительно read-модель	1
Проведены измерения задержки eventual consistency	1
Проведено нагрузочное тестирование, сравнение с синхронным вариантом	2
Ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- Для наглядности можно выводить логи: «Event published to stream», «Consumer received event, updated read model», «Read model now: …».
- При использовании Redis Streams важно правильно создать группу потребителей (команда XGROUP CREATE) при старте consumer, чтобы избежать дублирования при перезапуске.
- Для демонстрации eventual consistency можно написать небольшой скрипт или использовать цикл в клиенте.

- Если студент выполняет работу на Go, consumer может быть отдельной горутиной, не забывайте про синхронизацию доступа к read-модели.
- Подчеркните, что в реальных проектах read-модель обычно хранится в отдельной базе данных (например, MongoDB или отдельном PostgreSQL), а consumer является отдельным развёртываемым сервисом.
- Несмотря на то, что мы используем in-memory read-модель, важно сохранять идемпотентность, так как при перезапуске consumer может повторно получить события из потока. Поэтому храните список обработанных ID событий (в Redis или локально).

Лабораторная работа №7

GraphQL-сервер на основе существующего REST API

1. Цель работы

Сформировать у студентов навыки проектирования и реализации гибких API с помощью GraphQL как альтернативы жёстко фиксированным REST-эндпоинтам. Студенты научатся создавать GraphQL-схему, оборачивающую существующий REST-сервис, писать резолверы

полей и применять DataLoader для эффективной пакетной загрузки связанных данных, решая классическую проблему N+1 запросов.

Знать:

- Архитектуру GraphQL: схема, типы, запросы, мутации, резолверы.
- Отличия GraphQL от REST: гибкость выборки полей, единая точка входа, строгая типизация.
- Проблему N+1 запросов и механизм её решения через DataLoader (пакетная загрузка и кэширование в рамках запроса).

Уметь:

- Проектировать GraphQL-схему (SDL — Schema Definition Language) на основе предметной области.
- Реализовывать резолверы, делегирующие вызовы существующему REST API.
- Интегрировать DataLoader для объединения множественных запросов к REST API в один пакетный запрос.
- Тестировать GraphQL-эндпоинт через встроенный GraphiQL или Apollo Sandbox.

Владеть:

- Методикой выбора между REST-подходом и GraphQL в зависимости от требований клиента (мобильное приложение, сложные связи, необходимость точной выборки данных).
- Приёмами оптимизации сетевого взаимодействия с помощью пакетирования.

2. Теоретическая часть

2.1. GraphQL как альтернатива REST

В REST API структура ответа жёстко задана сервером. Клиент получает все поля ресурса, даже если ему нужна только часть, а для получения связанных данных приходится выполнять дополнительные HTTP-запросы к другим эндпоинтам. Это приводит к избыточной передаче данных (over-fetching) и каскадным запросам (under-fetching).

GraphQL решает эти проблемы, передавая контроль над выборкой данных клиенту. Клиент описывает в запросе, какие поля и связи ему нужны, и получает ровно запрошенную структуру в ответе. Сервер предоставляет единую точку входа (чаще всего /graphql), принимает запросы на языке запросов GraphQL, валидирует их по строго типизированной схеме и возвращает JSON, форма которого в точности повторяет запрос.

Ключевые элементы GraphQL:

- **Схема (Schema)** — центральный артефакт, описывающий типы данных, запросы (Query), мутации (Mutation), подписки (Subscription). Схема строго типизирует все доступные данные и операции.
- **Типы (Object Types)** — определения объектов с полями, каждое из которых имеет тип и может принимать аргументы.
- **Запросы (Queries)** — операции только на чтение, точка входа для получения данных.
- **Мутации (Mutations)** — операции изменения данных, аналогичные POST/PUT/PATCH в REST.
- **Резолверы (Resolvers)** — функции, которые разрешают (получают) значение конкретного поля для конкретного объекта. Резолвер верхнего уровня (Query) обычно обращается к сервису или внешнему API.

2.2. Проблема N+1 и DataLoader

Предположим, у нас есть GraphQL-запрос, который получает список продуктов и для каждого продукта — его категорию. Если резолвер поля category для каждого продукта делает отдельный HTTP-запрос к REST API категорий, то для N продуктов произойдёт N дополнительных запросов (плюс 1 запрос за списком продуктов) — это проблема N+1.

DataLoader — стандартное решение. Это библиотека (изначально из экосистемы JavaScript, но теперь реализованная для всех основных языков), которая:

- Собирает ключи (идентификаторы) всех запрошенных объектов в рамках одного GraphQL-запроса.
- Выполняет **один** пакетный запрос для загрузки всех объектов сразу.
- Кэширует результаты в рамках текущего запроса, чтобы при повторном запросе того же ключа не выполнять лишних загрузок.

DataLoader гарантирует, что каждый уникальный ключ в рамках одного запроса будет загружен не более одного раза, а все загрузки одного типа будут автоматически сгруппированы.

3. Задание на лабораторную работу

Вам необходимо создать GraphQL-сервер, который является обёрткой над одним из ранее созданных REST API. Можно использовать любой из ранее реализованных сервисов:

- CRUD продуктов из ЛР2 (с R2DBC/реактивной БД),
- или складской сервис CQRS из ЛР4,
- или банковский сервис из ЛР5 (Event Sourcing),

- или даже сторонний публичный REST API (JSONPlaceholder, например).

Рекомендуется использовать сервис продуктов с категориями: продукты и категории связаны отношением «многие-к-одному» (у продукта есть categoryId). Это создаст наглядную проблему N+1 для демонстрации DataLoader. Если используете другой сервис, обеспечьте наличие связи между сущностями, загружаемой отдельным запросом.

3.1. Требования

1. **Спроектируйте GraphQL-схему**, описывающую основные сущности и связи. Для продуктов и категорий:
    - Тип Product: поля id, name, price, category (тип Category).
    - Тип Category: поля id, name, products (список продуктов этой категории — опционально).
    - Корневой Query: product(id: ID!), products, category(id: ID!), categories.
    - Опционально Mutation: createProduct, updateProduct, deleteProduct, мутирующие через REST API.
2. **Реализуйте резолверы**, делегирующие вызовы существующему REST API.
    - Поле category для Product должно загружаться через REST-запрос вида GET /categories/{categoryId}.
    - **Без DataLoader** каждый продукт будет порождать отдельный HTTP-запрос — это нужно продемонстрировать.
3. **Интегрируйте DataLoader** для пакетной загрузки категорий.
    - DataLoader собирает все categoryId из запрошенных продуктов и делает **один** REST-запрос, например GET /categories?ids=1,2,3 (если REST API поддерживает фильтрацию по списку ID) либо получает все категории и фильтрует в памяти. Если REST API не поддерживает множественный фильтр, можно загрузить все категории и закэшировать.
    - Примените DataLoader также для других потенциальных N+1 ситуаций (например, загрузка продуктов категории).
4. **Демонстрация N+1 и эффекта DataLoader**:
    - Включите логирование HTTP-запросов от GraphQL-сервера к REST API (на уровне клиента).
    - Выполните GraphQL-запрос, запрашивающий список продуктов с категориями:

graphql

```
{ products { id name category { name } } }
```

- Сначала без DataLoader (или с DataLoader в выключенном состоянии) покажите, что количество запросов к /categories равно числу продуктов.
- Затем с DataLoader покажите, что запрос к категориям выполняется один раз (пакетно).
- Сохраните скриншоты логов.

5. **GraphQL-эндпоинт** должен быть доступен через веб-интерфейс (GraphiQL, если предоставляется фреймворком, или Sandbox). Продемонстрируйте выполнение нескольких запросов с разными наборами полей.

3.2. Отчёт

- Схема GraphQL (SDL).
- Ключевые фрагменты кода: определение схемы (если кодом), резолверы, DataLoader.
- Скриншоты GraphQL-запросов и ответов.
- Скриншоты логов, демонстрирующие количество REST-вызовов без и с DataLoader.
- Краткое сравнение GraphQL и REST для данного кейса.

4. Вариативность по языкам и фреймворкам

Язык	Рекомендуемый GraphQL-фреймворк	DataLoader
Java	spring-boot-starter-graphql (Spring for GraphQL)	org.dataloader.DataLoader и DataLoaderRegistry
Kotlin	Spring for GraphQL (совместим с Kotlin) + graphql-kotlin (опционально)	Тот же DataLoader, что и в Java
Python	Strawberry (рекомендуется) или Graphene + starlette/FastAPI	strawberry.dataloader.DataLoader или aiodataloader
Go	gqlgen (генерация из схемы)	graph-gophers/dataloader
C#	Hot Chocolate (рекомендуется) или GraphQL.NET	Встроенный IDataLoader<TKey, TValue> в Hot Chocolate

Общие требования:

- GraphQL-сервер должен работать на отдельном порту (например, 4000) и обращаться к REST API на другом порту.
- Для HTTP-клиента к REST API используйте асинхронные клиенты (WebClient в Spring, aiohttp в Python, http.Client в Go, HttpClient в C#).
- DataLoader обязательно должен быть продемонстрирован в действии.

5. Порядок выполнения

1. **Запустите существующий REST-сервис** (продукты/категории или другой). Убедитесь, что он работает и доступен.
2. **Создайте новый проект GraphQL-сервера.** Добавьте зависимости для выбранного GraphQL-фреймворка.
3. **Спроектируйте и опишите схему** (schema-first или code-first). Для Spring for GraphQL можно использовать schema-first: файл schema.graphqls в ресурсах. Для gqlgen — аналогично. Strawberry и Hot Chocolate поддерживают оба подхода, но code-first более естествен.
   Пример SDL schema-first:

```
graphql

type Product {

  id: ID!

  name: String!

  price: Float!

  category: Category

}

type Category {

  id: ID!

  name: String!

  products: [Product!]!

}

type Query {

  product(id: ID!): Product

  products: [Product!]!
```

```
  category(id: ID!): Category
  categories: [Category!]!
}
```

4. **Реализуйте резолверы.** Для этого создайте REST-клиент (асинхронный) к вашему REST API. В резолвере Query.products вызовите GET /products. В резолвере поля Product.category — вызов GET /categories/{categoryId}.
5. **Добавьте DataLoader для загрузки категорий.**
    - Создайте DataLoader<Long, Category>, который принимает список ids и делает один GET /categories?ids=1,2,3 (или загружает все категории и выбирает нужные).
    - В резолвере поля Product.category вместо прямого вызова REST-клиента используйте dataLoader.load(categoryId).
    - В Java: зарегистрируйте DataLoaderRegistry в контексте GraphQL-запроса. В Spring for GraphQL можно получить DataLoader через @BatchMapping или создать вручную при каждом запросе.
    - В Python (Strawberry): используйте @strawberry.dataloader и аннотированный загрузчик.
    - В Go (gqlgen): реализуйте интерфейс dataloader.BatchFunc.
6. **Настройте логирование HTTP-запросов** к REST API. Для Spring WebClient можно настроить logging.level.reactor.netty.http.client=DEBUG и фильтровать строки. Для других языков — аналогично.
7. **Проведите эксперимент:**
    - Выполните GraphQL-запрос списка продуктов с категориями **без DataLoader**: либо временно закомментировав DataLoader и используя прямой вызов в резолвере. Подсчитайте количество запросов к /categories в логах.
    - Включите DataLoader, повторите запрос, убедитесь, что вызов к /categories происходит один раз.
8. **Протестируйте запросы с разной глубиной и набором полей**, демонстрируя гибкость GraphQL.
9. **Оформите отчёт.**

6. Контрольные вопросы

1. Как GraphQL решает проблемы over-fetching и under-fetching? Приведите примеры.
2. Что такое проблема N+1 в контексте GraphQL и как DataLoader её решает?

3. Каким образом DataLoader группирует запросы, если они поступают от разных резолверов в рамках одного GraphQL-запроса?
4. Нужно ли реализовывать DataLoader для корневых полей (например, загрузка списка продуктов)? Почему?
5. Как GraphQL-схема обеспечивает строгую типизацию и валидацию запросов до выполнения резолверов?
6. В чём преимущества и недостатки GraphQL по сравнению с REST?
7. Как бы вы организовали мутации (создание, обновление, удаление) через GraphQL поверх существующего REST API?
8. Можно ли совместить GraphQL и gRPC? Как в таком случае организовать резолверы и DataLoader?

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
GraphQL-схема адекватно отражает предметную область	2
Резолверы корректно вызывают существующий REST API	2
Реализован DataLoader для пакетной загрузки связанных данных	2
Эксперимент демонстрирует N+1 без DataLoader и решение с DataLoader (подтверждено логами)	2
GraphQL-эндпоинт работает, запросы возвращают ожидаемые данные	1
Ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- Если ваш REST API не поддерживает фильтрацию по нескольким ID, в DataLoader можно загрузить все категории (GET /categories) и выбрать нужные в коде. Это тоже допустимо, но важно отметить, что это менее эффективно; лучше модифицировать REST API для поддержки пакетной выборки.

- Для Spring for GraphQL удобно использовать аннотацию @BatchMapping, которая автоматически создаёт DataLoader. Однако для полного понимания механизма студентам полезно реализовать DataLoader вручную, используя DataLoaderRegistry.
- При настройке логирования HTTP-клиента убедитесь, что в логах видны тела запросов, а не только факт обращения. Для Python aiohttp можно включить logging.getLogger('aiohttp.client').setLevel(logging.DEBUG).
- Студенты могут столкнуться с тем, что GraphQL-запросы не содержат .graphql расширения при тестировании через curl. Рекомендуется использовать GraphiQL или Apollo Sandbox для интерактивного исследования схемы.
- В отчёте обязательно покажите скриншоты GraphQL-запроса с категориями и логов REST-клиента, где видно количество вызовов.
- Напомните, что DataLoader работает на уровне одного запроса и не кэширует данные между запросами; для глобального кэша нужны другие механизмы.

Лабораторная работа №8

Нагрузочное тестирование: блокирующий vs реактивный стек

1. Цель работы

Сформировать у студентов практические навыки планирования и проведения нагрузочного тестирования серверных приложений, интерпретации ключевых метрик производительности (RPS, перцентили задержки) и обоснованного выбора архитектурного стиля (блокирующий или реактивный) в зависимости от профиля нагрузки.

Знать:

- Основные метрики производительности: пропускная способность (RPS), средняя задержка, медианная задержка (p50), хвостовые задержки (p95, p99), максимальная задержка.
- Принципиальные различия в поведении блокирующей многопоточной модели и реактивной неблокирующей модели под нагрузкой.
- Методику нагрузочного тестирования: warm-up, стационарный режим, повторяемость, обработка выбросов.

Уметь:

- Реализовывать функционально идентичные CRUD-сервисы на блокирующем и реактивном стеках (на примере выбранного языка).
- Настраивать и запускать нагрузочные тесты с помощью bombardier, wrk или k6.
- Собирать и анализировать RPS и задержки, строить сравнительные таблицы и графики.
- Делать выводы о целесообразности применения реактивного подхода в зависимости от характера нагрузки (вычислительная, I/O-зависимая, смешанная).

Владеть:

- Методикой доказательного выбора архитектуры на основе измерений.
- Приёмами выявления узких мест (пул потоков, пул соединений БД) через анализ метрик.

2. Теоретическая часть

2.1. Блокирующая модель: потоки и пул соединений

В традиционных блокирующих фреймворках (Spring MVC, Flask, синхронный Go net/http с database/sql, синхронный ASP.NET Core) на каждый входящий HTTP-запрос выделяется отдельный поток ОС (или горутина, но с блокирующим I/O). Когда обработчик выполняет операцию ввода-вывода (запрос к базе данных, вызов другого сервиса), поток блокируется на всё время ожидания ответа. Пока он заблокирован, он не может обслуживать другие запросы, а операционная система вынуждена переключать контексты между множеством потоков, что создаёт накладные расходы.

При увеличении числа одновременных соединений (concurrency) сверх определённого предела (обычно 200–500 потоков) пропускная способность резко падает, а задержки экспоненциально растут. Это связано с исчерпанием пула потоков сервера и пула соединений к базе данных: новые запросы становятся в очередь, ожидая освобождения ресурса.

2.2. Реактивная модель: Event Loop и неблокирующий I/O

Реактивные серверы (Spring WebFlux, FastAPI с async/await, Go с асинхронными драйверами БД в отдельных горутинах, ASP.NET Core async) используют малое число рабочих потоков (обычно по числу ядер CPU) и событийно-ориентированный цикл (Event Loop). Операции ввода-вывода не блокируют поток: запрос регистрируется, и поток переходит к обработке следующего события. Когда ответ от БД готов, цикл получает уведомление и продолжает обработку с того места, где остановился.

Это позволяет эффективно обслуживать тысячи одновременных соединений без переключения контекста и без выделения дополнительных потоков. Потребление памяти

и процессора остаётся низким. Пропускная способность растёт почти линейно с нагрузкой до упора в ресурсы самой базы данных или сети.

2.3. Ключевые метрики

- **RPS (Requests Per Second)** — количество успешных запросов в секунду. Показывает пропускную способность системы.
- **Latency (задержка)** — время от отправки запроса до получения ответа.
    - **Avg (средняя)** — чувствительна к выбросам.
    - **p50 (медиана)** — 50% запросов выполнены быстрее этого значения.
    - **p95** — 95% запросов выполнены быстрее; важный показатель «хвостовой» задержки.
    - **p99** — 99%; характеризует наихудшие случаи для большинства пользователей.
- **Максимальная задержка (Max)** — наихудшее время, может быть вызвано GC, сбросом буферов и т.д.

Для высоконагруженных систем важно, чтобы хвостовые задержки (p95, p99) были приемлемыми; средняя может скрывать проблемы.

2.4. Методика нагрузочного тестирования

- **Warm-up** — прогрев JIT-компилятора и кэшей: перед основным тестом делают несколько тысяч «разогревочных» запросов, метрики которых не учитывают.
- **Стационарный режим** — тест должен длиться достаточно долго (не менее 30 секунд), чтобы усреднить колебания.
- **Повторяемость** — выполнить не менее трёх замеров, результаты усреднить.
- **Изоляция** — тестируемые серверы запущены на одной машине, но по очереди, чтобы не влиять друг на друга. Нагрузочный генератор — на отдельной машине или хотя бы на отдельном ядре (можно на той же, но следить, чтобы CPU не был перегружен).
- **Уровни параллелизма** — варьировать количество одновременных соединений (10, 100, 500, 1000) для выявления порога деградации блокирующего сервера.

3. Задание на лабораторную работу

Реализуйте два **функционально идентичных** CRUD-сервера для управления сущностью «Продукт» (id, name, price). Один сервер построен на блокирующем стеке, второй — на реактивном. Затем проведите их нагрузочное тестирование и сравнительный анализ.

3.1. Блокирующий сервер (Сервер А)

- Для Java: Spring Boot MVC + Spring Data JPA (Hibernate) + PostgreSQL (JDBC-драйвер).
- Используйте контроллеры, возвращающие синхронные объекты (Product, List<Product>).
- Эндпоинты:
    - GET /products — получить все продукты.
    - GET /products/{id} — получить продукт по id.
    - POST /products — создать продукт.
    - PUT /products/{id} — обновить продукт.
    - DELETE /products/{id} — удалить продукт.
- База данных PostgreSQL, доступ через JPA-репозиторий (или синхронный JDBC Template).
- В обработчиках возможны блокирующие задержки (например, Thread.sleep(5) для имитации бизнес-логики).

3.2. Реактивный сервер (Сервер Б)

- Для Java: Spring Boot WebFlux + Spring Data R2DBC (R2DBC PostgreSQL).
- Полный асинхронный CRUD, все методы возвращают Mono/Flux.
- Та же схема БД, но доступ через R2DBC.
- Искусственная задержка (если добавляется) должна быть неблокирующей: Mono.delay(Duration.ofMillis(5)) или аналог.

Примечание: Для чистоты сравнения оба сервера должны использовать одно и то же окружение (одна БД, одинаковые сетевые условия, одинаковый объём предзагруженных данных, например 100 продуктов). Искусственная задержка 5 мс добавляется в каждый обработчик запроса (кроме массового списка), чтобы эмулировать реальную бизнес-логику и дать преимущество реактивному стеку.

3.3. Нагрузочное тестирование

1. **Подготовьте тестовые данные:** 100 продуктов в БД.
2. **Прогрев:** перед каждым тестом выполните 1000 запросов к GET /products/{id} со случайным id, чтобы прогреть JIT и кэши.
3. **Тестовые сценарии:**
    - **Сценарий 1: Простой запрос (I/O bound).** GET /products/{id} со случайным id от 1 до 100. Здесь доминирует ожидание ответа от БД.

- **Сценарий 2: Смешанная нагрузка.** 50% запросов GET /products/{id}, 50% POST /products (создание нового продукта). Для POST тело {"name": "test", "price": 10.0}.
- **Сценарий 3 (опционально): Чтение списка.** GET /products — нагрузка на сериализацию JSON и передачу данных.

4. **Параметры тестов:** Для каждого сценария выполнить замеры при трёх уровнях параллелизма: -c 10, -c 100, -c 500. Длительность каждого теста 30 секунд. Инструмент — bombardier (или wrk) с выводом latency distribution. Пример команды:

bash

bombardier -c 100 -d 30s -l http://localhost:8080/products/1

(ключ -l выводит детальную статистику, включая процентили).

5. **Сбор метрик:** Зафиксируйте RPS, p50, p95, p99, max latency для каждого теста. Повторите каждый тест трижды, вычислите средние значения.

3.4. Анализ результатов

- Постройте таблицы: для каждого сценария и уровня параллелизма укажите RPS и p95 latency для блокирующего и реактивного серверов.
- Постройте графики (можно в Excel/Google Sheets): зависимость RPS от количества соединений для обоих стеков. Отобразите точку «перелома», где блокирующий сервер начинает деградировать.
- Напишите выводы: при каком уровне параллелизма реактивный стек даёт значительный выигрыш? Как меняются хвостовые задержки (p99) при росте нагрузки? В каком сценарии (чтение, запись) разница наиболее заметна? Какие ограничения могут быть у реактивного стека? Обоснуйте, в каких проектах вы бы выбрали реактивный подход, а в каких — традиционный.

3.5. Отчёт

- Код (ключевые классы) для обоих серверов.
- Скриншоты запуска серверов и нагрузочных тестов.
- Таблицы с результатами (усреднённые по трём запускам).
- Графики.
- Аргументированные выводы.

4. Вариативность по языкам и фреймворкам

	Блокирующий стек	Реактивный стек
Java	Spring MVC + JPA (Hibernate) + PostgreSQL JDBC	Spring WebFlux + R2DBC (r2dbc-postgresql)
Kotlin	Spring MVC + JPA (можно с корутинами, но JDBC блокирующий)	Spring WebFlux + coroutines + R2DBC
Python	Flask + SQLAlchemy (синхронный) + psycopg2	FastAPI + SQLAlchemy async (asyncpg) / asyncpg напрямую
Go	net/http + database/sql (pgx/stdlib или pq)	net/http + pgxpool (асинхронный)
C#	ASP.NET Core с синхронными контроллерами + Entity Framework Core (синхронные методы)	ASP.NET Core Minimal API (полностью async) + Npgsql (асинхронный)

Для всех стеков: обязательно использовать одну и ту же СУБД (PostgreSQL), одинаковую схему и тестовые данные. Искусственная задержка реализуется средствами языка/фреймворка: для синхронных — блокирующий sleep, для асинхронных — неблокирующая задержка (Task.Delay, asyncio.sleep, time.After в Go возвращает канал, который не блокирует горутину). В Go асинхронный драйвер pgx уже использует горутины, но для имитации задержки можно использовать time.Sleep в отдельной горутине? Лучше использовать неблокирующую задержку: time.AfterFunc или time.NewTimer. В Python — await asyncio.sleep(0.005). В C# — await Task.Delay(5).

5. Порядок выполнения

1. **Подготовьте инфраструктуру:**
    - PostgreSQL (локально или Docker).
    - Инструмент нагрузочного тестирования: bombardier (https://github.com/codesenberg/bombardier) — он выводит перцентили и удобен под Windows/Linux/macOS. Или wrk.
    - Среда разработки.
2. **Создайте проект блокирующего сервера.**
    - Зависимости согласно выбранному языку.

- Настройте подключение к БД (пул соединений по умолчанию, например, HikariCP для Java, размер пула ~50).
- Добавьте сущность Product, репозиторий (JpaRepository, синхронный SQLAlchemy Session, и т.д.).
- В каждом методе контроллера добавьте Thread.sleep(5), time.sleep(0.005), и т.д. **перед** обращением к БД или после, но до возврата.
- Запустите сервер на порту 8080.

3. **Создайте проект реактивного сервера.**
   - Зависимости согласно выбранному языку.
   - Настройте асинхронное подключение (R2DBC, asyncpg, pgxpool).
   - Используйте реактивный репозиторий (ReactiveCrudRepository, асинхронные методы).
   - В каждом методе добавьте неблокирующую задержку на 5 мс (Mono.delay, asyncio.sleep, Task.Delay, time.After). Убедитесь, что она не блокирует Event Loop.
   - Запустите на порту 8081.
4. **Заполните БД тестовыми данными.** Напишите скрипт или выполните серию POST-запросов для создания 100 продуктов.
5. **Прогрев.** Для каждого сервера выполните быстрый тест с -c 10 -n 1000 к GET /products/{id}.
6. **Проведите серию тестов для каждого сценария и уровня параллелизма.** Строго соблюдайте последовательность: сначала тестируем один сервер, останавливаем, потом второй, чтобы ресурсы не пересекались. Записывайте результаты.
7. **Постройте таблицы и графики.**
8. **Оформите отчёт, сформулируйте выводы.**

6. Контрольные вопросы

1. Почему при увеличении числа одновременных соединений блокирующий сервер начинает резко терять в пропускной способности, а реактивный — нет?
2. Что означает p99 задержка 200 мс? Почему важно смотреть на хвостовые перцентили, а не только на среднюю?
3. В каком сценарии разница между блокирующим и реактивным стеком минимальна? Почему?

4. Как повлияет на результаты теста ограничение пула соединений к БД? Сравните поведение двух стеков при пуле размером 5.
5. Какие дополнительные накладные расходы может создавать реактивный стек (например, накладные на операторы Reactor)? В каких случаях эти расходы перевешивают выигрыш?
6. Можно ли масштабировать блокирующее приложение, увеличив количество экземпляров? В чём тогда остаются преимущества реактивного подхода?
7. Какой инструмент нагрузочного тестирования вы бы выбрали для gRPC и почему?
8. Какие ещё метрики (помимо RPS и latency) стоит мониторить при нагрузочном тестировании реальной системы (потребление CPU, памяти, сборщик мусора)?

7. Критерии оценки (максимум 10 баллов)

Критерий	Баллы
Реализованы и запущены оба сервера с идентичным функционалом	2
Проведены нагрузочные тесты для всех указанных сценариев и уровней параллелизма	3
Собраны и представлены метрики RPS, p50, p95, p99 (таблицы)	2
Построены графики и сделан обоснованный анализ результатов	2
Ответы на контрольные вопросы	1

Минимальный порог — 6 баллов.

Методические рекомендации

- При тестировании на локальной машине убедитесь, что CPU не перегружен нагрузочным генератором. Запускайте генератор на другом ядре/машине, либо используйте bombardier с ограничением использования CPU (есть опция).
- Для Java важно указать одинаковые настройки JVM (Xmx, GC) для обоих приложений, чтобы сравнение было честным.

- Студентам на Python стоит учесть, что uvicorn с несколькими воркерами использует процессы, а не потоки; для чистоты эксперимента сравнение должно проводиться с одним воркером на реактивном и синхронном сервере.
- В отчёте обязательно укажите версию инструмента тестирования, параметры ОС, характеристики машины (CPU, RAM).
- При анализе результатов не просто констатируйте «реактивный быстрее», а объясните, почему при низкой нагрузке разница может быть незначительной, а при высокой — огромной.
- Если один из стеков показывает неожиданно низкие результаты, проверьте настройки пула соединений к БД — возможно, он является узким местом.

Лабораторная работа №9

Итоговый проект: реактивный бэкенд с CQRS + gRPC

1. Цель работы

Продемонстрировать комплексное владение технологиями и архитектурными паттернами, изученными в течение семестра, путём самостоятельного проектирования и реализации законченного прототипа высокопроизводительной back-end системы. Студенты интегрируют реактивный фреймворк, gRPC и/или GraphQL, CQRS, Event Sourcing (опционально), очереди сообщений, контейнеризацию и нагрузочное тестирование, а также защищают принятые архитектурные решения перед комиссией.

Знать:

- Критерии выбора архитектурного стиля под заданные нефункциональные требования.
- Методику сквозного проектирования от предметной области до развёртывания.

Уметь:

- Проектировать систему на основе требований, выбирая релевантные паттерны (CQRS, Event Sourcing, реактивный доступ к данным).
- Реализовывать HTTP/gRPC/GraphQL API с полным покрытием бизнес-логики.
- Настраивать асинхронные проекции через Redis Streams / RabbitMQ.
- Упаковывать все компоненты в Docker-контейнеры и описывать инфраструктуру через Docker Compose.
- Проводить нагрузочное тестирование и интерпретировать метрики для обоснования архитектурного выбора.
- Готовить техническую документацию и публичную защиту.

Владеть:

- Инженерным подходом к созданию современных highload-систем с использованием реактивных и событийно-ориентированных практик.

2. Теоретическая часть

Итоговый проект синтезирует все пройденные темы в целостную архитектуру. Ключевые элементы, которые должны быть продемонстрированы:

- **Реактивный стек:** неблокирующий HTTP-сервер, асинхронный доступ к БД (R2DBC, asyncpg, pgxpool, Npgsql async) — основа для высокой конкурентности.
- **gRPC или GraphQL** — эффективное и гибкое межсервисное взаимодействие (gRPC для высокоскоростных внутренних вызовов, GraphQL для клиентских запросов с точной выборкой данных). Допускается реализация одного из протоколов, либо комбинация (gRPC между внутренними сервисами, GraphQL как внешний API).
- **CQRS (обязательно):** разделение операций изменения состояния (Commands) и операций чтения (Queries) на уровне моделей, а в идеале — и сервисов. Read-модель(и) обновляются через асинхронные проекции.
- **Event Sourcing (опционально, рекомендуется):** вместо прямого хранения текущего состояния сохраняется журнал событий. Агрегаты восстанавливаются через replay. Позволяет естественно интегрироваться с очередями для проекций.
- **Очереди сообщений (Redis Streams или RabbitMQ):** обеспечивают асинхронное обновление read-моделей, реализуют конечную согласованность и развязывают Command и Query компоненты.
- **Docker и Docker Compose:** все компоненты (основной сервис, БД, брокер, consumer проекции) контейнеризированы и запускаются единой командой.

- **Нагрузочное тестирование:** количественное обоснование принимаемых решений — измерение RPS и хвостовых задержек (p95, p99) для критических сценариев.

3. Задание на лабораторную работу

3.1. Выбор предметной области

Студент (или команда до 2 человек) выбирает одну из предложенных или согласовывает собственную тему:

- **Бронирование билетов / номеров** (события, места).
- **Система заказов** (корзина, оформление, статусы).
- **Платформа аукциона** (ставки, таймеры).
- **Банковские счета** (расширение ЛР5-6).
- **Складской учёт** (расширение ЛР4-6).

3.2. Обязательные технологические компоненты

Проект должен включать:

1. **Реактивное REST или gRPC API для команд (Command API):**
    - Минимум 2–3 бизнес-операции, изменяющих состояние (создание сущности, обновление, отмена и т.п.).
    - Сохранение событий в Event Store (если ES) или запись в нормализованные таблицы (CQRS без ES) с транзакционной поддержкой.
    - После успешной фиксации изменений события/команды публикуются в очередь.
2. **Асинхронный consumer проекций:**
    - Подписывается на очередь событий.
    - Обновляет одну или несколько денормализованных read-моделей (например, для экранов «Мои заказы», «Свободные места», «Баланс счёта»).
    - Идемпотентная обработка.
3. **API для запросов (Query API):**
    - Может быть реализовано как GraphQL (предпочтительно, если внешнее) или как gRPC/REST, но обязательно читает **только** из read-модели.
    - Поддерживает фильтрацию и получение связанных данных без N+1 (если GraphQL — DataLoader).

4. **Реактивный доступ к БД:**
    - Использование R2DBC/asyncpg/pgxpool/Npgsql async для всех обращений к PostgreSQL.
    - Никаких блокирующих JDBC/ORM.
5. **Контейнеризация:**
    - Dockerfile для каждого сервиса (command, query, consumer — могут быть в одном проекте, но должны быть разделены хотя бы на уровне модулей; consumer может быть отдельным процессом).
    - docker-compose.yml, поднимающий PostgreSQL, Redis/RabbitMQ, и все сервисы.
6. **Нагрузочное тестирование:**
    - Сценарий: проверка пропускной способности Command API (например, создание заказа) и Query API (получение списка/детали).
    - Использование bombardier, wrk или k6. Для gRPC — ghz.
    - Результаты: RPS, p50, p95, p99 latency. Сравнение с эталонными замерами блокирующего аналога (можно гипотетическое объяснение, если не реализован).
7. **Документация и презентация:**
    - **README.md:** описание архитектуры, схема развёртывания, инструкция по запуску (docker-compose up), описание API (gRPC proto или GraphQL schema), результаты нагрузочного тестирования.
    - **Презентация (5–7 слайдов):** цель, архитектурная диаграмма, ключевые технологии, метрики производительности, обоснование выбора реактивного стека и CQRS.

3.3. Критерии успешности минимума

- Все компоненты запускаются через docker-compose up.
- Command API корректно выполняет бизнес-операции.
- Query API возвращает актуальные (с задержкой ≤ 5 сек) данные из read-модели.
- Нагрузочный тест демонстрирует хотя бы 500 RPS на чтение и 200 RPS на запись при 100 одновременных соединениях (для локальной среды; цифры ориентировочные, зависят от железа, но нужно показать стабильность).

4. Вариативность по языкам и фреймворкам

Язык	Рекомендуемый стек
Java	Spring WebFlux + Spring Data R2DBC + gRPC-java / GraphQL Java + Redis Streams (Spring Data Redis Reactive) + Docker
Kotlin	Spring WebFlux + coroutines + R2DBC + gRPC-Kotlin + Redis
Python	FastAPI + asyncpg + grpcio (или Strawberry для GraphQL) + aioredis + Docker
Go	net/http + pgxpool + gRPC + gqlgen (опционально) + go-redis + Docker
C#	ASP.NET Core Minimal API (async) + Npgsql + Grpc.Net / Hot Chocolate + StackExchange.Redis + Docker

Очереди: предпочтительно Redis Streams (проще, уже использовался), допустимо RabbitMQ.

5. Порядок выполнения (рекомендуемый план)

1. **Неделя 1: Проектирование**
    - Определить предметную область, выделить агрегаты, события, команды.
    - Спроектировать нормализованные таблицы и read-модели.
    - Описать proto-файл (если gRPC) или GraphQL-схему.
    - Нарисовать диаграмму потоков данных.
2. **Неделя 2: Разработка ядра**
    - Реализовать Command API с реактивным доступом к БД, внедрить Event Sourcing (если используется).
    - Подключить брокер, реализовать публикацию событий.
    - Реализовать consumer, обновляющий read-модель.
    - Написать основные тесты (интеграционные, проверяющие идемпотентность).
3. **Неделя 3: Query API и контейнеризация**
    - Реализовать Query API (GraphQL или REST/gRPC).
    - Убедиться в работе сквозного сценария.

- Написать Dockerfile, docker-compose.yml.
- Проверить запуск всей системы в контейнерах.

4. **Неделя 4: Нагрузочное тестирование и документация**
   - Провести серию нагрузочных тестов, записать метрики.
   - Подготовить README.md и презентацию.
   - Репетиция защиты.

6. Контрольные вопросы (примерный перечень для защиты)

1. Почему в вашем проекте выбрана реактивная модель? Приведите количественные аргументы.
2. Как вы обеспечили идемпотентность consumer проекций?
3. Что произойдёт, если consumer проекции упадёт на длительное время? Как система восстановит консистентность?
4. Почему вы выбрали gRPC/GraphQL? Какие альтернативы рассматривались?
5. Как в вашей архитектуре решается проблема конкурентного доступа к агрегатам?
6. Какие метрики нагрузочного тестирования вы считаете ключевыми и почему?
7. Если бы требовалось обеспечить строгую консистентность между командой и запросом, как бы вы изменили архитектуру?
8. Как вы мониторили бы такую систему в production?

7. Критерии оценки (максимум 30 баллов)

Критерий	Баллы
Архитектура и дизайн (CQRS, обоснование выбора технологий)	6
Реализация Command API (реактивный CRUD, Event Sourcing/транзакции)	6
Асинхронная проекция (consumer, очередь, идемпотентность)	5
Query API (GraphQL/gRPC/REST без N+1, использование read-модели)	4

Критерий	Баллы
Docker-упаковка и воспроизводимость	3
Нагрузочное тестирование и анализ результатов	3
Документация и презентация	2
Защита (ответы на вопросы)	1

Минимальный порог — 18 баллов (60%).

Методические рекомендации

- При проектировании следуйте принципу «минимально жизнеспособный продукт»: выделите ровно столько бизнес-кейсов, сколько сможете качественно реализовать. Лучше меньше, но глубже.
- Обязательно продемонстрируйте асинхронный consumer как отдельный процесс (горутина, фоновая задача, отдельный контейнер).
- Для нагрузочного тестирования используйте скрипт, который сначала создаёт тестовые данные.
- В презентации обязательно приведите диаграмму развёртывания и график результатов нагрузочного тестирования.
- При защите будьте готовы обосновать каждый архитектурный выбор, ссылаясь на изученные принципы и измерения.
- Поощряется использование CI/CD (например, GitHub Actions для сборки и тестов), но не обязательно.
- Оценивается не только работающий код, но и инженерная культура: читаемый код, адекватные комментарии, чистота коммитов.