Лабораторная работа №3 — Профилирование и оптимизация микросервисов (на основе ЛР2)

Цель: научиться выявлять и устранять узкие места в Java-приложениях, используя современные инструменты профилирования, и оптимизировать реализацию их серверного микросервиса из ЛР2, который был намеренно реализован «неэффективно».

Задачи:

1. Запустить свой сервис B (сервер) в режиме, позволяющем профилирование.
2. Провести анализ работы приложения с помощью нескольких инструментов:
   - Java Flight Recorder (JFR)
   - Java Mission Control (JMC)
   - VisualVM
   - Async Profiler
   - (опционально) IntelliJ Profiler
3. Найти в сервисе:
   - CPU hot spots
   - allocation hot spots
   - неэффективные алгоритмы
   - чрезмерное создание объектов
   - неоптимальные структуры данных
4. Описать проблему.
5. Оптимизировать код.
6. Повторно провести профилирование и сравнить результаты.

Теормин

1 Понятие профилирования

Профилирование — это процесс анализа поведения программы во время выполнения. Оно отвечает на вопросы:

- Где тратится процессорное время?
- Что создаёт слишком много объектов?
- Какие методы вызываются чаще всего?
- Где возникают блокировки потоков?
- Что вызывает GC и насколько он загружен?

Профилирование позволяет обнаружить реальные, а не предполагаемые узкие места. Причины использования профилирования:

- Java-приложения имеют скрытые издержки абстракций,
- часто пишутся с избыточным выделением памяти,
- могут использовать неоптимальные алгоритмы,
- могут некорректно работать с потоками,
- могут неправильно использовать коллекции,
- и потому в реальных проектах оптимизация невозможна без профилирования.

2 Разновидности профилирования:

Тип профилирования	Что анализирует
CPU profiling	что нагружает процессор
Memory profiling	что создаёт объекты и вызывает GC
Allocation profiling	какие участки создают больше всего мусора
Thread profiling	блокировки, ожидания, contention
I/O profiling	медленные сети/файлы
Event profiling (JFR)	системные события JVM

3 Обзор инструментов:

3.1 JDK Flight Recorder (JFR) — MUST HAVE

JFR — встроенный инструмент профилирования, не влияющий сильно на производительность.

Как включить на сервере:

java -XX:StartFlightRecording=filename=profile.jfr,dumponexit=true -jar service-b.jar

Что смотреть:

- CPU usage → Hot Methods
- Allocation → Memory leaks
- GC → pause times
- Threads → blocked threads
- Exceptions → frequent exceptions

Куда открывать:

В Java Mission Control.

3.2 Java Mission Control (JMC)

GUI для анализа файлов .jfr.

Что нужно уметь:

- открыть файл .jfr
- смотреть разделы:
    1. Method Profiling
    2. Memory Leak
    3. CPU & Allocation
    4. GC Activity
    5. Threads View

3.3 VisualVM

Простой и наглядный инструмент.

Основные вкладки:

- CPU Sampler
- Memory Sampler
- Heap Dump Analyzer
- Threads

3.4 Async Profiler

Лучший инструмент для CPU и allocations.

Запуск CPU:

./profiler.sh -e cpu -f cpu.html <pid>

Запуск памяти:

./profiler.sh -e alloc -f alloc.html <pid>

Получаем:

- flame graph (CPU)
- allocation flame graph (мусор)

3.5 IntelliJ Profiler

Простой для начинающих.

Обычно используют в режиме:

- Sampling
- Allocation

Универсальный пошаговый план выполнения

1. Запустить сервис B с профилированием JFR

java -XX:StartFlightRecording=filename=profile.jfr -jar service-b.jar

2. Сервис A генерирует нагрузку

Отправлять 100–10000 запросов к сервису B.

Можно через Postman, curl или собственный клиент.

3. Собрать профиль JFR

Файл profile.jfr появится после остановки процесса. (Стопаете приклад)

4. Открыть JFR в Java Mission Control

Посмотреть:

- CPU hot methods
- Allocations
- GC
- threads
- exceptions

5. Дополнительно профилировать Async Profiler

Получить flame graph.

6. Найти узкое место

Типичные проблемы:

- двойные сортировки
- лишние преобразования строк
- регулярные выражения
- вложенные циклы
- создание объектов внутри циклов
- дорогие мапперы
- неиспользование стримов там, где они оптимальны

- использование стримов там, где они не оптимальны
- повторное чтение файлов
- BigDecimal в больших циклах

7. Оптимизировать

Варианты оптимизаций:

- заменить алгоритм
- убрать лишние преобразования
- кэшировать вычисления
- заменить структуры данных
- использовать предрасчитанные результаты
- перенести часть логики в базу
- уменьшить создание объектов

8. Повторно провести профилирование

И доказать, что:

- latency уменьшилась
- CPU time снизился
- потоков стало меньше
- мусора меньше

Вариант 1 — Прогноз погоды

Идея:

A → отправляет название города.

B → ищет прогноз по городу в большой базе (генерирует огромный список погодных данных, фильтрует в памяти, сортирует два раза)

Инструменты:

- JFR + JMC
- Async Profiler (CPU + alloc)

Что искать:

- Hotspot: сортировка (TimSort)
- Аллокации списков/строк
- Двойная фильтрация
- GC activity

Ожидаемые выводы:

- сортировка выполняется несколько раз
- создаётся слишком много объектов

Что оптимизировать в ЛР3:

- удалить двойную сортировку
- сделать пагинацию вместо полного набора
- использовать более компактные структуры
- кешировать результаты

Вариант 2 — Курсы валют

Идея:

A → отправляет желаемые валюты.

B → считает курс по тикеру на основе исторических данных.

Инструменты:

- VisualVM
- JFR

Что искать:

- BigDecimal hot spots

- слишком много конвертаций типов

Оптимизация:

- заменить BigDecimal на double, где точность не важна
- кешировать частые конверсии

Вариант 3 — Расчёт расстояний между точками

Идея:

A → отправляет координаты двух точек.

B → возвращает расстояние между двумя координатами.

Инструменты:

- Async Profiler (CPU)

Hot spots:

- Math.sqrt
- Math.pow
- двойные циклы

Оптимизация:

- заменить pow(x,2) на x*x (ну тут опирайтесь уже на конкретно вашу реализацию)
- изменить алгоритм N^2 → grid-based или k-d tree

Вариант 4 — Рекомендации фильмов

Идея:

A → отправляет жанр.

B → отдаёт top-N фильмов по жанру.

ЛР3 — профилирование

Инструменты:

- JMC (Memory)
- Async Profiler (alloc)

Hot spots:

- создание массивов
- повторная сортировка

- избыточные DTO

Оптимизация:

- использовать stream().limit()
- сортировать 1 раз
- кешировать рейтинги

Вариант 5 — Поиск книг по автору

Идея:

A → отправляет автора.

B → возвращает книги по автору. ЛР3

Инструменты:

Hot spots:

- регулярные выражения
- String operations
- создание временных коллекций

Оптимизация:

- заменить regex на startsWith/contains
- использовать заранее индексированные данные

Вариант 6 — Котировки акций

Идея:

A → отправляет тикер акций.

B → генерирует котировку «симуляцией» рынка.

Инструменты:

- JFR
- Async Profiler (alloc)

Hot spots:

- double → BigDecimal → double
- сортировка
- создание объектов PriceDTO

Оптимизация:

- убрать лишние преобразования
- использовать собственный POJO без BigDecimal
- кешировать расчёты

Вариант 7 — Аналитика новостей (scoring)

Идея:

A → отправляет запрос с темой(например, музыка).

B → сканирует новости и присваивает скор.

Инструменты:

Hot spots:

- вложенные фильтры
- многократная нормализация текста
- регулярные выражения

Оптимизация:

- прекомпилировать регулярки
- вынести нормализацию текста в предрасчёт
- ограничить размер файла

Вариант 8 — Учет студентов: вычисление среднего балла

Идея:

A → отправляет название предмета.

B → считает средний балл студента по курсам. Загружает все оценки (например, 100k записей) при каждом запросе.Для каждого курса студента последовательно вызывает DAO (N запросов на курс — N+1 проблема).

Инструменты

- Async Profiler (cpu + alloc)
- Java Flight Recorder (allocation + cpu)

Что искать (hot spots)

- Частые вызовы DAO/PreparedStatement в цикле.
- Большие аллокации коллекций и объектов-обёрток (Long, Integer, DTO).

- Многострочные сортировки и повторные проходы по коллекциям.

Что оптимизировать

- Предварительные SQL JOIN/группировка на стороне БД (агрегация по studentId/course).
- Batch-запросы/IN-clause вместо N отдельных вызовов.
- Использовать primitive-friendly структуры, стримы с уменьшенной аллокацией, кэширование результатов по курсу.

Вариант 9 — Генерация UUID (неоптимальный путь)

Идея:

A → запрос на генерацию UUID.

B → выдаёт UUID.

Инструменты:

Hot spots:

- создание новых SecureRandom на каждый запрос
- String.format

Оптимизация:

- реиспользовать генератор
- убрать лишние преобразования

Вариант 10 — Имитация очереди задач

Идея:

A → запрос на получение пользовательских задач.

B → отдаёт набор задач. На каждый запрос генерирует **100k задач** (title, timestamp, priority), запускает последовательные фильтрации,nсортировки. Выполняет многослойную группировку, создавая множество временных коллекций. Генерирует множество объектов-обёрток и DTO для каждого элемента. Выполняет несколько проходов по списку («naive pipeline generation → filter → sort → group»).

Инструменты

- Async Profiler (cpu + alloc)

- Java Flight Recorder (allocation + cpu)

Hot spots:

- Аллокации в Stream API (filter → map → sorted → collect).
- Ленивая инициализация Comparator'ов внутри sort → CPU-пики.
- Многократный обход списка (до 6 проходов).

Что оптимизировать

- Объединить фильтры в один предикат (один проход вместо трёх).
- Минимизировать сортировки → сортировать один раз по ключу.
- Использовать Comparator.compareInt / primitives без боксинга.
- Стримы заменить на ручные циклы для снижения аллокаций.
- Генерацию 100k задач вынести в подготовленный пул/кэш (если допустимо).

Вариант 11 — Перевод слов

Идея:

A → слово на русском.

B → переводит слово с русского на английский. На каждый запрос считывает целиком большой словарь из файла (несколько десятков / сотен МБ). Для поиска строит сложные regex-маппинги с lookaround и запускает Unicode.normalize(NFKC) для каждой строки.

Инструменты

- Async Profiler (alloc + cpu)
- JFR (String allocation + IO waits)

Что искать (hot spots)

- Частые File.readAllBytes / new String(...) аллокации.
- Regex-матчинги, которые занимают CPU (Pattern.matcher, lookaround).
- Частые вызовы Normalizer.normalize и создания новых String.

Что оптимизировать

- Кэшировать словарь в памяти (или использовать mmap / memory-mapped file).

- Предкомпилировать Pattern, избегать lookaround, использовать trie/ HashMap для быстрых lookup.
- Минимизировать Unicode-нормализацию: нормализовать при загрузке и хранить уже нормализованные формы.

Вариант 12 — Проверка орфографии

Идея:

A → отправляет текст.

B → исправляет текст. Для каждого слова генерирует все возможные правки (Levenshtein neighbours): удаление каждой буквы, замена каждой буквы всеми символами, вставка каждой буквы во все позиции. Для каждого кандидата делает lookup в словаре на 100k+ слов. Использует списки, сет структуры, регулярки — большое количество аллокаций.

Инструменты

- Async Profiler (alloc + cpu)
- JFR (String allocation + IO waits)
- VisualVM — heap dump, sampler,

Что искать (hot spots)

- высокая аллокация строк при генерации edits множество короткоживущих стрингов
- функции генерации кандидатов (edits1, edits2)
- частые GC-паузы (Young Gen), всплески мелких объектов
- многократный полный проход по словарю (100k строк) для каждого кандидата
- высокая CPU загрузка от миллиона сравнений строк

Что оптимизировать

- Заменить список словарных слов на hashSet (O(1) lookup).
- Предварительно нормализовать словарь.
- Ограничивать количество уровней правок (только edits1).
- Использовать trie-структуру вместо полного перебора.

- Генерацию кандидатов делать лениво и прерывать после первых совпадений.

Вариант 13 — Управление заказами (корзина)

Идея:

A → отправляется список идентификаторов товаров.

B → возвращает корзину пользователя.На вход получает list, для каждого itemId выполняет отдельный DAO-запрос (N+1). Для расчёта скидок в цикле делает вложенные циклы по правилам скидок и промо-условиям. Много раз преобразует сущности в DTO, проводит 2–3 сортировки и фильтрации.

Инструменты

- Async Profiler (cpu + alloc)
- JFR (lock contention + allocations)

Что искать (hot spots)

- Множество мелких SQL-вызовов (latency).
- Аллокации DTO и промежуточных списков.
- Горячие места в логике скидок — вложенные циклы.

Что оптимизировать

- Запросить все нужные товары одним batch/IN-запросом.
- Применять кеширование товаров/каталогов.
- Выделить вычисление скидок в потоковую/векторную обработку (один проход), уменьшить количество DTO.

Вариант 14 — Обработка платежей (симуляция)

Идея:

A → отправляет данные банковской карты.

B → выполняет валидацию и возвращает статус. Для каждого запроса выполняет тяжёлые крипто-операции: создание MessageDigest, многократные SHA-256/iterations (например, 10k итераций) для симуляции риск-оценки. Повторно сериализует/десериализует данные, держит в памяти список «черных карт» (20000 записей) и выполняет линейный поиск.

Инструменты

- Async Profiler (cpu) — увидеть дорогие хеши
- JFR (safepoint time + allocations)

Что искать (hot spots)

- Создание новых MessageDigest каждый раз.
- Большие CPU-затраты на многократные хеширования.
- Повторные парсинги/серилизации.

Что оптимизировать

- Переиспользовать объекты/реальзовать пул MessageDigest (если безопасно).
- Уменьшить количество итераций или переместить heavy-work в асинхронный бекграунд/очередь (в реалии — агрегировать).
- Хранить blacklist в структуре с быстрым lookup (HashSet, Bloom filter).

Вариант 15 — Расчёт BMI (ИМТ)

Идея:

A → отправляется вес и рост (все что нужно).

B → считает индекс массы тела и рекомендации. Для каждого запроса строит lookup-таблицы градаций заново (множество уровней). Выполняет BigDecimal-вычисления и многоразовые округления, выполняет одну и ту же формулу IMT 1000 раз (симулируя нагрузку). Дополнительно делает множественные проходы по таблицам рекомендаций.

Инструменты

- Async Profiler (alloc + cpu)
- JFR (allocation profiler — BigDecimal allocations)

Что искать (hot spots)

- Частые аллокации BigDecimal и BigInteger.
- Повторное построение таблиц/структур на каждый запрос.
- Множество одинаковых вычислений.

Что оптимизировать

- Использовать double/primitive там, где точность допускает.

- Кэшировать градации/таблицы рекомендаций.
- Уменьшить количество повторений формул; векторизовать расчёты если нужно.

Вариант 16 — Расписание поездов

Идея:

A → отправляется станцию.

B → отдаёт расписание по станции. Загружает все расписания (миллионы записей) при каждом запросе. Выполняет in-memory JOIN между маршрутами и остановками через nested loops (вложенные for), многократно сортирует (по 3 полям подряд) и фильтрует через regex. Собирает большие коллекции поездов и затем делает несколько проходов группировки.

Инструменты

- Async Profiler (cpu + alloc)
- JFR (oldgen/young gen allocations, GC pauses)

Что искать (hot spots)

- Большие выборки из БД и последующая работа в памяти.
- Негарантированные больших объектов/коллекций и частые сортировки.
- Regex-фильтры на больших наборах.

Что оптимизировать

- Фильтрация и join на стороне БД (WHERE station_id = ? + JOIN + LIMIT).
- Пагинация и streaming из БД (ResultSet streaming).
- Предобработанные индексы/материализованные представления для частых запросов.

Вариант 17 — Обработка метаданных изображений

Идея:

A → запрос получение метаданных по ID.

B → извлекает метаданные изображений. Загружает псевдо-изображение полностью в память для каждого запроса (большие байтовые массивы). Парсит вручную (reflection/slow parsing), сериализует метаданные несколько раз (JSON

→ объект → JSON → объект). Дополнительно запускает повторный парсинг/валидацию для каждой версии.

Инструменты

- Async Profiler (alloc + cpu)
- JFR (allocation, IO waits)

Что искать (hot spots)

- Большие byte[] аллокации (изображения).
- Частые сериализации/десериализации.
- Рефлексия в парсере EXIF.

Что оптимизировать

- Загружать только header/metadata (partial read) вместо полного изображения.
- Использовать специализированные EXIF-парсеры и избегать рефлексии.
- Кэшировать метаданные; минимизировать сериализации.

Вариант 18 — Генерация случайных чисел (большой диапазон)

Идея:

A → запрашивает набор случайных чисел.

B → генерирует статистику случайных чисел. Генерирует миллионы случайных значений на каждый запрос, хранит их в списке, затем считает распределение в несколько проходов (несколько O(n) проходов). Создаёт новый Random/ThreadLocalRandom каждый раз. Выполняет повторный прогон алгоритма (re-seed) и дорогостоящие преобразования.

Инструменты

- Async Profiler (cpu + alloc)
- JFR (allocation + thread states)

Что искать (hot spots)

- Аллокации больших списков/ArrayList.
- Стоимость создания Random/ThreadLocalRandom объектов.
- Много проходов по данным для histogram/percentile.

Что оптимизировать

- Генерировать поток данных и агрегировать в одном проходе (single-pass histogram).
- Переиспользовать генератор случайных чисел (ThreadLocalRandom.current()).
- Работать с примитивными массивами/primitive streams, не хранить весь набор целиком.

Вариант 19 — Случайная цитата

Идея:

A → запрос на случайную цитвту.

B → отдаёт случайную цитату. При каждом запросе считывает весь файл с цитатами. Делает Collections.shuffle (O(n)) на списке и затем берёт первый элемент. Нет кэширования; файл читается и парсится полностью каждый раз.

Инструменты:

- Async Profiler (alloc + io)
- JFR (I/O waits + allocations)

Что искать (hot spots)

- File I/O на каждый запрос.
- Аллокации коллекций и shuffle (swap, temporary objects).

Что оптимизировать

- Кэшировать список цитат в памяти и брать случайный индекс.
- Если памяти мало — использовать reservoir sampling при стриме файла.
- Избежать полной перестановки коллекции (shuffle) — выбрать случайный индекс.

Вариант 20 — Менеджер задач (ToDo)

Идея:

A → Отправляют новые задачи или запрашивает существующие.

B → отдаёт список задач пользователя. Загружает все задачи для всех пользователей (миллионы) при каждом запросе. Затем фильтрует по userId в

памяти через nested loops и проводит многослойную группировку. Выполняет тяжелые преобразования DTO и дублирует логику фильтрации.

Инструменты

- Async Profiler (alloc + cpu)
- JFR (GC, allocations)

Что искать (hot spots)

- Полные загрузки таблицы и большие одноразовые аллокации.
- Повторные преобразования DTO.
- Контенция на коллекциях при группировке.

Что оптимизировать

- Фильтрация/пагинация на стороне БД (WHERE user_id = ? LIMIT OFFSET).
- Lazy-mapping в DTO, преобразовывать только нужные поля.
- Использовать stream/paged queries and avoid loading all rows.

Вариант 21 — История погоды (по дате)

Идея:

A → отправляет дату и город.

B → возвращает температуру за прошлый период. При каждом запросе полностью парсит большой набор исторических записей (CSV/JSON) в память и делает линейный поиск по всем записям; выполняет несколько агрегаций (среднее, медиана) отдельными проходами.

Инструменты

- JFR + Java Mission Control (JMC)
- VisualVM (memory/heap, sampler)
- Async Profiler (allocation)

Что искать в профиле

- высокая аллокация при парсинге файла (много short-lived объектов)
- hot methods: парсинг строки, String.split, создание коллекций
- многократные полные проходы (повторные map/reduce)

Что оптимизировать

- загрузить и кэшировать данные при старте (или использовать базу)
- индексировать по дате (Map<LocalDate, List>) для O(1) поиска
- делать одну агрегацию в одном проходе (аккумулятор), заменить BigDecimal на primitive, если допустимо

Вариант 22 — Рекомендации музыки

Идея:

A → отправляет жанр

B → возвращает плейлист по жанру. При каждом запросе перебирает весь каталог (десятки тысяч треков), для каждого трека считает сходство (dense vector операции) и сортирует весь каталог.

Инструменты

- Async Profiler (CPU/flame graph)
- JFR (Method profiling)
- VisualVM (threads, CPU sampler)

Что искать

- горячие места: вычисление сходства (косинус/скалярные произведения), сортировка всего списка
- высокая загрузка CPU при обработке векторов
- частые аллокации временных векторов

Что оптимизировать

- предвычислять embeddings/векторы, использовать sparse представления
- использовать approximate nearest neighbor (ANN) библиотеки или индексы (FAISS/Annoy)
- ограничивать выборку (topK) без полного скана, кешировать популярные запросы

Вариант 23 — Генератор lorem ipsum

Идея:

A → запрос на генерацию длины N.

B → генерирует текст заданной длины. При каждом запросе создаёт сложную модель (псевдо-Марковскую) заново, много конкатенаций строк + в циклах, создаёт промежуточные списки и regex-постобработку.

Инструменты

- VisualVM (allocation)
- JFR (allocation, method profiling)

Что искать

- многочисленные аллокации строк (вызовы new String), StringBuilder создаётся и выбрасывается часто
- hot methods: concat, regex replace

Что оптимизировать

- использовать StringBuilder или StringJoiner для конкатенаций
- создать singleton генератор (переиспользуемое состояние) или кеш шаблонов
- сократить/предкомпилировать регулярки

Вариант 24 — Профиль пользователя

Идея:

A → запрос на профиль пользователя по id.

B → возвращает профиль и активность. Делает несколько последовательных запросов/агрегаций в память (N+1), сериализует/десериализует JSON несколько раз, мержит коллекции вручную.

Инструменты

- JFR (cpu, threads)
- VisualVM (heap / threads)
- Network/DB monitoring (если есть база)

Что искать

- много маленьких вызовов к DAO (N+1)
- hot methods: JSON (ObjectMapper) сериализация/десериализация
- allocation spike от DTO копий

Что оптимизировать

- объединить запросы (JOIN / fetch-join) или использовать projection DTO в SQL
- уменьшить количество сериализаций, использовать view/projection напрямую
- кешировать неизменяемые части профиля

Вариант 25 — Уведомления (flux)

Идея:

A → запрашивает список уведомлений для пользователя.

B → отдаёт поток непрочитанных уведомлений. Собирает уведомления из разных источников, объединяет через nested loops, для каждого уведомления выполняет тяжёлое форматирование (дата/строка) и возвращает Flux.

Инструменты

- Async Profiler (allocation + cpu)
- JFR (events, allocations)

Что искать

- heavy per-item formatting (DateTimeFormatter в синхронизации)
- много небольших объектов (String) → GC pressure
- медленные объединения списков

Что оптимизировать

- объединять источники более эффективно (merge streams), использовать flatMap с ограниченной concurrency
- предварительно форматировать даты в потокобезопасном формате (DateTimeFormatter без синхронизации)
- использовать batching (группировать уведомления)

Вариант 26 — Новости (актуальные)

Идея:

A → запрос на случайную новость.

B → B возвращает список новостей. На каждый запрос скачивает метаданные с внешних источников (HTTP) и парсит HTML DOM.Полностью

загружает изображения в память, делает image-size check (loading into memory). Затем парсит/нормализует контент и делает сортировку/фильтрацию (shuffles, multiple sorts).

Инструменты

- Async Profiler (cpu + alloc)
- JFR (I/O waits + safepoint)
- трассировка внешних HTTP-запросов (latency, retries)

Что искать (hot spots)

- Блокирующие внешние HTTP-вызовы в потоках обработчика.
- Аллокации при загрузке больших изображений.
- Парсинг DOM/HTML на больших строках.

Что оптимизировать

- Кэшировать внешние метаданные; использовать асинхронные вызовы и таймауты.
- Загружать только требуемые метаданные (избегать полной загрузки изображений).
- Offload image analysis в отдельный сервис/worker и использовать thumbnails.

Вариант 27 — Конвертация timestamp/часовых поясов

Идея:

A → отправляет timestamp.

B → переводит метку времени в локальное время. При каждом вызове загружает/воссоздаёт объекты ZoneRulesProvider, выполняет несколько конвертаций через разные форматы, создаёт много временных объектов.

Инструменты

- VisualVM (alloc)
- JFR (method profiling)

Что искать

- повторные создания ZoneRules/ZoneId/DateTimeFormatter
- synchronized вызовы в DateFormat (если используются)

- allocation hot spots на Temporal объекты

Что оптимизировать

- кешировать ZoneId и ZoneRules
- использовать DateTimeFormatter как thread-safe singleton или использовать DateTimeFormatter из java.time
- минимизировать количество форматирований (ленивая/отложенная сериализация)

Вариант 28 — Рейтинг приложений

Идея:

A → отправляет приложение.

B → возвращает рейтинг приложения. Пересчитывает рейтинг из всех отзывов (полный scan), сортирует отзывы и потом считает среднее, делает это синхронно и каждый раз.

Инструменты

- Async Profiler (cpu)
- VisualVM (alloc)

Что искать

- full-scan review list hot spot
- сортировки и агрегации, высокие аллокации
- N+1 при подгрузке отзывов

Что оптимизировать

- поддерживать агрегированное поле (материализованное среднее) при записи отзывов
- выполнять агрегации на БД (SQL AVG)
- использовать топ-N через heap (priority queue) вместо сортировки всего массива

Вариант 29 — Статус серверов (health)

Идея:

A → отправляет запрос на статус серверов.

B → проверяет статус ряда сервисов. Последовательно (синхронно) делает HTTP вызовы/проверки для всех 200 сервисов, парсит ответы и строит отчёт.

Инструменты

- JFR (threads, I/O)
- Async Profiler (cpu)

Что искать

- последовательные сетевые вызовы (waiting/blocking)
- блокировки и рост latency
- большое время ожидания в event-loop

Что оптимизировать

- параллелизовать проверки (flatMap с ограниченной concurrency)
- использовать timeouts и circuit-breaker (Resilience4j)
- кешировать последние статусы и отдавать stale+refresh

Вариант 30 — Математический сервис (операции)

Идея:

A → отправляет операцию для выполнения.

B → выполняет заданную операцию над большими массивами. Выполняет численное интегрирование/приближения через большое число шагов (например, 100k) и использует BigInteger/BigDecimal там, где можно double.

Инструменты

- Async Profiler (cpu)
- JFR (method)

Что искать

- горячие циклы с BigDecimal операции
- heavy math functions (pow, sin, cos)
- allocation spikes

Что оптимизировать

- заменить BigDecimal на double/long при допустимой точности

- уменьшить число шагов или применять адаптивные алгоритмы
- использовать native math libs / vectorized operations

Вариант 31 — Сервис для профилирования запросов

Идея:

A → отправляет операцию для выполнения

B → отвечает и позволяет клиенту измерить время на стороне A. Содержит несколько целенаправленных «узких мест» (вложенные циклы, лишняя сериализация), специально чтобы студенты могли профилировать.

Инструменты

- JFR + JMC (основной)
- Async Profiler (CPU+alloc)
- VisualVM (heap + threads)

Что искать

- hotspot-методы, allocation hot spots, GC pauses, блокировки
- специфические элементы в зависимости от того, что реализовано в B

Что оптимизировать

- заменить алгоритмы,
- уменьшить аллокации,
- вынести тяжелую работу асинхронно,
- кешировать

Вариант 32 — Фильтрация данных (сложные условия)

Идея:

A → запрос на фильтрацию отправляемых данных.

B → фильтрует набор данных по множественным правилам. Применяет последовательность фильтров, каждый делает полный проход по коллекции, иногда использует reflection-based predicates.

Инструменты

- Async Profiler (cpu)
- VisualVM (allocation)

Что искать

- повторные полные scans
- overhead от reflection (invoke)
- heavy predicate evaluation

Что оптимизировать

- объединить предикаты в один проход
- прекомпилировать/создать лямбды вместо reflection
- индексировать данные и применять short-circuit

Вариант 33 — Датчики температуры (поток)

Идея:

A → запрос на данные датчиков.

B → отдаёт Flux с данными датчиков. Генерирует данные в памяти для 1000 датчиков, на каждый элемент выполняет тяжёлую обработку и создаёт большие истории (lists).

Инструменты

- JFR (alloc, threads)
- Async Profiler (alloc)

Что искать

- рост памяти из-за хранения истории
- отсутствие backpressure → очередь растёт
- heavy per-item processing

Что оптимизировать

- использовать bounded buffer и backpressure
- хранить только скользящее окно (sliding window), а не всю историю
- выполнять лёгкую агрегацию на лету, отложенная обработка в batch

Вариант 34 — Комментарии к посту

Идея:

A → запрос на получение комментариев к посту.

B → возвращает комментарии.

(Рекурсивно строит полное дерево комментариев, делает глубокие копии и множество DTO-преобразований (много временных объектов)).

Инструменты

- VisualVM (heap)
- JFR (allocation)

Что искать

- большие объёмы временных объектов при сборке дерева
- рекурсивные вызовы, глубинные стеки
- long GC pauses

Что оптимизировать

- вернуть плоскую структуру с lazy-загрузкой children (пагинация)
- использовать DTO-просмотры / ленивую сериализацию
- избегать глубоких копий — использовать неизменяемые ссылки

Вариант 35 — Генерация фейковых пользователей

Идея:

A → запрос на генерацию пользователей.

B → генерирует N пользователей «на лету». На каждый запрос генерирует 10k пользователей в памяти (имя, адрес, email) с тяжёлыми строковыми операциями и множественными сериализациями.

Инструменты

- Async Profiler (alloc + cpu)
- JFR (allocation)

Что искать

- allocation hot spots при генерации объектов
- String operations, regex, repeated ObjectMapper calls

Что оптимизировать

- предварительно сгенерировать и кэшировать dataset, отдавать по страницам
- использовать primitive-friendly структуры, избегать regex при генерации имён

- уменьшить сериализации (один проход)

Таблица профилирования и оптимизации для ЛР3

Вариант	**Инструмент**	**Что искать**	**Тип горячих точек**	**Советы по оптимизации**
1	JFR, Async Profiler, VisualVM	CPU hot spots, Allocation	Двойная сортировка, фильтрация, временные списки	Убрать лишнюю сортировку, использовать пагинацию, компактные структуры
2	JFR, VisualVM	CPU, Memory	BigDecimal конверсии, повторные вычисления	Заменить BigDecimal на double там, где точность допустима, кешировать частые преобразования
3	Async Profiler, JFR	CPU	Math.sqrt, Math.pow, двойные циклы	Заменить pow(x,2) на x*x, изменить алгоритм N^2 → grid-based/k-d tree
4	JMC, Async Profiler	Memory, CPU	Повторная сортировка, лишние DTO	Сортировать один раз, кешировать результаты, limit()
5	Async Profiler, VisualVM	CPU, Allocation	Regex, String operations, временные коллекции	Regex → startsWith/ contains, индексировать данные
6	JFR, Async Profiler	CPU, Allocation	Double → BigDecimal → double, PriceDTO	Убрать лишние конверсии, использовать POJO без BigDecimal, кешировать расчёты
7	Async Profiler, JMC	CPU	Вложенные фильтры, нормализация текста, regex	Предкомпилировать regex, вынести нормализацию в предрасчёт
8	JFR, Async Profiler	CPU, Memory	Sentiment analysis, создание объектов	Кешировать результаты анализа, заменить алгоритм на линейный
9	JFR, Async Profiler	CPU	Repeated SecureRandom, String.format	Переиспользовать генератор, убрать лишние преобразования
10	JFR, VisualVM, Async Profiler (alloc	CPU, Memory, Allocation	Многократные фильтры, двойная сортировка, создание обёрток	Генерировать меньше объектов, свести фильтры в один проход, заменить groupingBy на ручное накопление

11	JFR, VisualVM, Async Profiler	IO + CPU (regex) + Allocation	Многократный проход по списку погодных данных	File IO, regex matcher, нормализация Unicode\ Повторное чтение файла, heavy regex, String operations Кеш словаря, прекомпилировать regex, нормализовать один раз, хранить Map
12	Async Profiler, JFR	CPU + Allocation	Генерация Levenshtein edits, большие временные строки, словарные lookup	Использовать HashSet, уменьшить количество edits, reuse буферов
13	JFR, VisualVM	IO (DB) + CPU + Allocation	Множество DAO-вызовов, вложенные скидочные циклы	Объединить запросы, убрать лишние сортировки, кешировать данные
14	JFR, Async Profiler	CPU	MessageDigest, hashing loop, сериализация цепочек	Кешировать Digest, уменьшить длину цепочек, буферизовать операции
15	JFR, VisualVM	fCPU + Allocation	BigDecimal операции, пересоздание lookup	Кешировать таблицы, использовать double, сократить количество повторов
16	JFR, VisualVM	CPU + Memory	nested loops $O(n^2)$, многократные сортировки, regex	Прединдексация, уменьшить сортировки, заменить regex на предикаты
17	JFR, VisualVM	CPU + Allocation + Memory	Полная загрузка файла, JSON сериализация, создание больших объектов	Ленивая загрузка, уменьшить количество сериализаций, кешировать парсер

18	Async Profiler	CPU + Allocation	Создание Random, миллионы значений, многопроходная агрегация	Reuse Random, примитивные массивы, один проход для histogram
19	JFR , VisualVM	IO + CPU + Allocation	Полное чтение файла, shuffle O(n), сортировка для выбора (!)	Кешировать файл, выбирать случайную строку по индексу
20	JFR, Async Profiler	Memory + CPU + Allocation	Полная загрузка, nested loops, тяжёлые DTO	Индексировать по userId, упростить DTO, избегать повторной фильтрации
21	JFR, VisualVM, Async Profiler	CPU, Allocation	Full scan CSV/ JSON, multiple aggregations	Индексирование, один проход, primitive arrays
22	Async Profiler, JFR	CPU	Dense vector similarity, full catalog scan	ANN, topK, кеширование, sparse vectors
23	VisualVM, JFR	Allocation, CPU	String concatenation, regex	StringBuilder/StringJoiner, singleton generator, precompiled regex
24	JFR, VisualVM	CPU, Allocation	N+1 JSON serialization, DTO copies	Объединение запросов, уменьшение сериализаций, projection DTO
25	Async Profiler, JFR	CPU, Allocation	Heavy per-item formatting, merge Flux	FlatMap с ограниченной concurrency, batching, thread-safe formatter
26	JFR, Async Profiler	IO + CPU + Memory + Allocation	HTML DOM parse, загрузка изображений, Base64 decode	Лёгкий HTML parser, кеширование метаданных, не грузить изображения полностью
27	Async Profiler, VisualVM	CPU, Allocation	ZoneRulesProvider recreation, multiple conversions	Singleton ZoneRules, thread-safe DateTimeFormatter, минимизировать форматирование
28	Async Profiler, VisualVM	CPU, Allocation	Full scan reviews, sorting, N+1	Материализованное среднее, SQL AVG, top-N heap
29	JFR, Async Profiler	Threads, CPU	Sequential HTTP checks, blocking	Параллелизация, timeouts, circuit breaker, caching

30	Async Profiler, JFR	CPU, Allocation	BigDecimal in loops, heavy math	Replace BigDecimal with double, reduce steps, vectorized math
31	JFR, Async Profiler, VisualVM	CPU, Allocation	Purposeful heavy loops and conversions	Replace algorithms, reduce allocations, async work, caching
32	Async Profiler, VisualVM	CPU, Allocation	Reflection-based predicates, full scans	Compile predicates, merge filters, short-circuit, index data
33	JFR, Async Profiler	CPU, Allocation	Heavy per-item processing, unbounded history	Bounded buffer, sliding window, batch aggregation
34	VisualVM, JFR	Memory, Allocation	Deep recursive tree, DTO copies	Lazy-loading children, immutable references, DTO projections
35	Async Profiler, JFR	CPU, Allocation	Bulk fake user generation, string operations	Pre-generate dataset, pagination, primitive structures, reduce serialization

Шаблон отчёта

Студент: _____

Группа: _____

Вариант: _____

1. Цель работы

Выявить узкие места в сервисе B (ЛР2), измерить производительность, память, CPU, latency и оптимизировать реализацию.

2. Описание сервиса

- Сервис A: _____
- Сервис B (неоптимально): _____
- Описание тяжёлых операций / неоптимальных алгоритмов:

3. Инструменты профилирования

3.1. JDK Flight Recorder (JFR)

-XX:StartFlightRecording=filename=profile.jfr,settings=profile,dumponexit=true -jar service-b.jar

- Анализ в JMC: Method Profiling, Memory, Threads, GC Activity
- Сделать скриншоты hot methods, allocation, threads, GC

3.2. VisualVM

- Подключение к JVM → вкладки: Sampler → CPU/Memory, Heap Dump, Threads
- Скриншоты hot methods, allocation, потоки

3.3. Async Profiler

CPU профилирование:

./profiler.sh -e cpu -f cpu.html <pid>

Allocation профилирование:

./profiler.sh -e alloc -f alloc.html <pid>

- Открыть HTML → flame graph, сделать скриншоты

4. Генерация нагрузки (Сервис A)

- Количество запросов: 100–10 000
- Тип запросов: одиночные / Flux / batch
- Зафиксировать среднее время отклика

5. Результаты профилирования (до оптимизации)

Метрика	Значение	Скрин
CPU usage	_____	
Allocation rate	_____	
Latency	_____	
GC pause time	_____	
Threads blocked	_____	

Горячие методы (CPU): _____

Горячие аллокации: _____

Потенциальные узкие места: _____

6. Оптимизации

- Внесённые изменения:
  - Алгоритм / структура данных
  - Кэширование
  - Переиспользование объектов
  - Параллельная обработка / batching

7. Результаты профилирования (после оптимизации)

Метрика	Значение	Скрин
CPU usage		
Allocation rate		
Latency		
GC pause time		
Threads blocked		

Сравнение hot methods и allocation «до/после».

8. Выводы

- Основные узкие места до оптимизации
- Какие изменения дали наибольший эффект
- Общие рекомендации по улучшению реактивных и потоковых сервисов

ПРИМЕРЫ

Вариант 36 — Генерация отчёта по транзакциям

Сценарий:

- Сервис A отправляет запрос с периодом времени → получает отчёт по транзакциям.
- Сервис B (неоптимально) собирает все транзакции в память, группирует и агрегирует их с использованием наивных алгоритмов.

1. Неоптимальный код сервиса B

```
@RestController
@RequestMapping("/report")
public class TransactionReportController {

    private final TransactionService transactionService;

    /**
     * Конструктор контроллера для внедрения зависимости сервиса транзакций.
     *
     * @param transactionService Сервис транзакций
     */
    public TransactionReportController(TransactionService transactionService) {
        this.transactionService = transactionService;
    }

    /**
     * Генерирует отчёт по транзакциям за указанный период.
     *
     * @param startDate Начальная дата периода отчёта
     * @param endDate Конечная дата периода отчёта
     * @return Список объектов отчёта {@link ReportDTO}, отсортированный по общей сумме транзакций
     */
    @GetMapping
    public List<ReportDTO> generateReport(
        @RequestParam String startDate,
        @RequestParam String endDate
    ) {
        // Шаг 1: загрузка всех транзакций за указанный период
        List<Transaction> transactions = transactionService.getTransactions(startDate, endDate);

        // Шаг 2: группировка транзакций по пользователям
        Map<String, List<Transaction>> grouped = new HashMap<>();
        for (Transaction tx : transactions) {
            grouped.computeIfAbsent(tx.getUserId(), k -> new ArrayList<>())
                .add(tx);
        }

        // Шаг 3: подсчёт суммарных значений по каждой группе пользователей
        List<ReportDTO> report = new ArrayList<>();
        for (Map.Entry<String, List<Transaction>> entry : grouped.entrySet()) {
            double total = 0;
            for (Transaction tx : entry.getValue()) {
                total += tx.getAmount();
            }
            report.add(new ReportDTO(entry.getKey(), total));
        }

        // Шаг 4: сортировка списка отчётов по убыванию сумм
        report.sort(Comparator.comparingDouble(ReportDTO::getTotal).reversed());

        return report;
    }
}
```

Проблемы кода:

- Загружаем все транзакции в память (плохая масштабируемость).
- Группировка через HashMap + второй проход → двойная работа.
- Третий проход — сортировка полного списка.
- Нет ленивой обработки / потоков → весь процесс синхронный.

2. Инструмент профилирования

- CPU + allocation: Async Profiler
- Memory / Heap: VisualVM
- Метрики JVM: JFR + Java Mission Control

3. Как снимаем метрики

3.1 Async Profiler — CPU и Allocation:

CPU

./profiler.sh -e cpu -f cpu_flame.html <PID_SERVICE_B>

Allocation

./profiler.sh -e alloc -f alloc_flame.html <PID_SERVICE_B>

- Открываем *.html в браузере → видим горячие методы и точки аллокации

3.2 VisualVM — память:

1. Подключаемся к JVM (service B)
2. Открываем вкладку Sampler → Memory
3. Запускаем нагрузку через сервис A (HTTP-запросы)
4. Фиксируем рост heap, количество объектов, GC activity

JFR + JMC:

-XX:StartFlightRecording=filename=profile.jfr,settings=profile,dumponexit=true -jar service-b.jar

- Открываем profile.jfr в JMC
- Анализ: Method Profiling, Allocation, GC Activity

4. Оптимизация

Идея:

- Делегировать агрегацию БД, использовать stream API и ленивые вычисления.
- Минимизировать количество проходов по коллекции.

Оптимизированный код:

```
// Обрабатываем GET-запрос на получение отчета
@GetMapping
public List<ReportDTO> generateReportOptimized(
      @RequestParam String startDate,   // Параметр запроса с начальной датой отчета
      @RequestParam String endDate      // Параметр запроса с конечной датой отчета
) {
   // Шаг 1: Получение данных из базы через сервис транзакций
   List<ReportDTO> report = transactionService.getAggregatedReport(startDate, endDate);

   // Шаг 2: Сортировка результатов по сумме (от большего к меньшему)
   // Возвращаем отсортированную коллекцию, ограничивая вывод первыми 100 элементами
   return report.stream()                  // Создаем поток данных
            .sorted(Comparator.comparingDouble(ReportDTO::getTotal).reversed()) // Сортируем по общей сумме в обратном порядке
            .limit(100)                    // Оставляем только верхние 100 записей
            .collect(Collectors.toList()); // Преобразовываем поток обратно в список
}
```

В сервисе TransactionService:

```
// Метод для получения агрегированного отчета по пользователям и суммам транзакций
public List<ReportDTO> getAggregatedReport(String startDate, String endDate) {
   // SQL-запрос для выборки суммы транзакций каждого пользователя за заданный временной диапазон
   String sql = "SELECT user_id, SUM(amount) AS total " +          // Агрегируем сумму транзакций
            "FROM transactions " +                                 // Из таблицы транзакций
            "WHERE date >= :start AND date <= :end " +             // Фильтрация по диапазону дат
            "GROUP BY user_id";                                    // Группировка по каждому пользователю

   // Выполняем запрос с параметрами и маппингом результата в объекты типа ReportDTO
   return jdbcTemplate.query(sql,                                  // Запускаем запрос
                  Map.of("start", startDate,                       // Передаем параметры начала и конца периода
                        "end", endDate),
                  (rs, rowNum) ->                                  // Преобразование строки результата в объект DTO
                     new ReportDTO(rs.getString("user_id"),
                           rs.getDouble("total"))); // Конструируем объект DTO
}
```

Что изменилось:

- Не загружаем все транзакции в память.
- Считаем суммы в БД, один проход → меньше аллокаций, меньше CPU.

- Ограничение топ-N → меньшая сортировка.

5. Пошаговое объяснение

Шаг	Что делаем	Ожидаемый эффект
1	Запускаем старый код, нагрузка с сервиса A	CPU spike, высокая память, slow response
2	Снимаем профили: Async Profiler, VisualVM, JFR	Видим горячие методы: for циклы + сортировка, Allocation spike
3	Вносим оптимизацию (агрегируем в БД, stream API, топ-N)	Уменьшение heap, меньший CPU, faster response
4	Снова снимаем метрики	Сравниваем «до / после», фиксируем улучшение в отчёте
5	Выводы	принцип: минимизировать проходы, делегировать вычисления, использовать потоковые / ленивые операции

6. Результаты (пример)

Метрика	До оптимизации	После оптимизации
Среднее время отклика	2500 ms	300 ms
CPU usage	80 %	20 %
Allocation rate	200 MB/s	50 MB/s
GC pause time	200 ms	50 ms

Вариант 37 — Потоковая обработка заказов (Flux)

Сценарий:

- Сервис A отправляет поток заказов в сервис B через WebClient / Flux.
- Сервис B наивно обрабатывает каждый заказ синхронно и создаёт временные объекты для расчёта бонусов.

1. Неоптимальный код

```
// Контроллер для обработки заказов и формирования отчета с бонусами
@GetMapping("/processOrders")
public Flux<OrderReportDTO> processOrders() {
   // Получаем полный список всех существующих заказов
   List<Order> orders = orderService.getAllOrders();

   // Формируем промежуточный отчет с расчетом бонусов для каждого заказа
   List<OrderReportDTO> report = new ArrayList<>();
   for (Order order : orders) {
      // Расчет бонуса является ресурсоемким процессом для каждого отдельного заказа
      double bonus = calculateBonus(order);
      report.add(new OrderReportDTO(order.getId(), bonus)); // Добавляем новый элемент
отчета
   }

   // Преобразуем полученный список отчетов в реактивный поток (Flux)
   return Flux.fromIterable(report);
}

// Приватный метод расчета бонуса для конкретного заказа
private double calculateBonus(Order order) {
   // Простая демонстрационная формула для вычисления бонуса
   double result = 0;
   for (int i = 0; i < 10000; i++) {          // Ресурсоемкая операция повторяется много раз
      result += Math.sqrt(order.getAmount() * i); // Вычисление квадратного корня суммы
умноженной на индекс итерации
   }
   return result;
}
```

Проблемы:

- Синхронная обработка → нет параллелизма
- Heavy calculation на каждом объекте → CPU hot spots
- Много временных объектов → высокий memory allocation

2. Инструменты профилирования

- CPU и Allocation: Async Profiler
- Memory: VisualVM

- JVM Metrics: JFR + Java Mission Control

Снятие метрик:

- Async Profiler → ./profiler.sh -e cpu -f cpu.html <pid>
- VisualVM → Sampler → CPU / Memory
- JFR → -XX:StartFlightRecording=filename=profile.jfr,...

3. Оптимизация

- Используем параллельный поток (parallel Flux)
- Минимизируем временные объекты (прямые примитивные вычисления)
- Ограничиваем concurrency

```
// Оптимизированный контроллер для асинхронной обработки заказов и построения отчета
@GetMapping("/processOrdersOptimized")
public Flux<OrderReportDTO> processOrdersOptimized() {
    // Возвращаем поток данных (Flux) для параллельной обработки заказов
    return Flux.fromIterable(orderService.getAllOrders())       // Преобразуем список заказов в поток
            .parallel()                                 // Включаем параллельную обработку потоков
            .runOn(Schedulers.parallel())                   // Распределяем выполнение операций на разные потоки
            .map(order -> {                             // Картируем каждый заказ в объект OrderReportDTO
                double bonus = optimizedCalculateBonus(order); // Рассчитываем бонус
                return new OrderReportDTO(order.getId(), bonus); // Формируем отчет
            })
            .sequential();                              // Объединяем потоки обратно в последовательный порядок
}

// Ускоренный метод расчета бонуса для отдельного заказа
private double optimizedCalculateBonus(Order order) {
    double result = 0;
    double amount = order.getAmount();                      // Сохраняем значение суммы заказа для оптимизации

    // Оптимизированная ресурсоемкая логика расчета бонуса
    for (int i = 0; i < 10000; i++) {
        result += Math.sqrt(amount * i);                    // Суммируем корень произведения суммы и индекса
    }
    return result;
}
```

Эффект:

- Параллельное выполнение → меньше CPU bottleneck
- Memory allocation снижается за счёт потоковой обработки
- Возможность масштабирования на большое количество заказов

4. Пошаговое объяснение

Шаг	**Что делаем**	**Ожидаемый эффект**
1	Запускаем старый код	Высокое CPU, задержка обработки всех заказов
2	Снимаем профили	Hot methods: calculateBonus, высокая аллокация
3	Вносим оптимизацию	Flux.parallel + Schedulers.parallel
4	Снова снимаем метрики	CPU распределяется между потоками, память ниже
5	Выводы	—-

Вариант 38 — Генерация отчетов с повторными преобразованиями даты

Сценарий:

- Сервис A запрашивает отчёт по событиям за период.
- Сервис B создаёт DTO, конвертирует даты через ZonedDateTime и каждый раз создаёт новый ZoneRulesProvider.

1. Неоптимальный код

```
// Обработка GET-запроса для формирования отчета по событиям
@GetMapping("/eventsReport")
public List<EventDTO> getEventsReport() {
    // Получаем список всех событий из сервиса
    List<Event> events = eventService.getAllEvents();

    // Готовим список объектов EventDTO для представления данных
    List<EventDTO> dtos = new ArrayList<>();

    // Перебираем каждое событие и формируем соответствующий объект EventDTO
    for (Event e : events) {
        // Преобразуем дату события в форматом UTC с указанием формата вывода
        ZonedDateTime zdt = e.getDate().atZone(ZoneId.of("UTC")); // Устанавливаем временную
зону UTC
        String formatted = zdt.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")); //
Форматируем дату-время

        // Добавляем новый объект EventDTO в итоговый список
        dtos.add(new EventDTO(e.getId(), formatted));
    }

    // Возвращаем сформированный список объектов EventDTO
    return dtos;
}
```

Проблемы:

- Повторное создание ZoneRulesProvider → высокие аллокации
- Форматирование дат на каждом объекте → CPU hot spots
- Синхронная обработка всего списка → задержка при больших объёмах

2. Инструменты профилирования

- CPU и Allocation: Async Profiler → горячие методы: ZonedDateTime.of, DateTimeFormatter.format
- Memory: VisualVM → аллокации объектов даты и DTO
- JFR: Method Profiling, Allocation

3. Оптимизация

- Создать один DateTimeFormatter и ZoneId

- Использовать stream API для ленивой обработки

private static final DateTimeFormatter FORMATTER =

DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

private static final ZoneId UTC = ZoneId.of("UTC");

```
// Обработка GET-запроса для оптимального формирования отчета по событиям
@GetMapping("/eventsReportOptimized")
public List<EventDTO> getEventsReportOptimized() {
   // Получаем список всех событий из сервиса
   return eventService.getAllEvents().stream()                    // Преобразуем список событий в
поток
         .map(event -> {                                  // Применяем преобразование к каждому
событию
            // Преобразуем дату события в формат UTC и форматируем её согласно
шаблону
            ZonedDateTime zdt = event.getDate().atZone(UTC);      // Определяем временную
зону UTC
            String formatted = zdt.format(FORMATTER);            // Применяем заранее
определённый шаблон форматирования

            // Формируем и возвращаем объект EventDTO
            return new EventDTO(event.getId(), formatted);
         })
         .collect(Collectors.toList());                    // Собираем результаты преобразования
в список
}
```

Эффект:

- Уменьшение аллокаций
- CPU hot spot сокращается, код быстрее работает
- Ленивый Stream API позволяет легко интегрировать Flux для больших объёмов

4. Пошаговое объяснение

Шаг	**Что делаем**	**Ожидаемый эффект**
1	Запускаем старый код	Высокий CPU, много временных объектов
2	Снимаем профили	Async Profiler показывает hot methods в ZonedDateTime и format
3	Оптимизируем	Один DateTimeFormatter + ZoneId, stream API
4	Снова профилируем	Снижение CPU и allocation, меньше GC pauses
5	Выводы	——