# Project Tracker Performance Optimization Report

## 1. Initial Profiling Summary

### JMeter Load Testing Baseline

- **JMeter Thread Group Configuration:**
  - **Number of Threads (users):** 100
  - **Ramp-up period (seconds):** 60
  - **Loop Count:** Infinite
  - **Duration (seconds):** 60
- **GET /projects:**
  - Average Response Time: 1282 ms
  - Throughput (RPS): 13.3/sec
  - Error Rate: 0.00%
- **POST /tasks:**
  - Average Response Time: 1243 ms
  - Throughput (RPS): 12.8/sec
  - Error Rate: 0.00%
- **GET /users/{id}/tasks:**
  - Average Response Time: 1455 ms
  - Throughput (RPS): 12.3/sec
  - Error Rate: 0.00%
- **Latency Percentiles:**
  - 90th Percentile: 2599 ms
  - 95th Percentile: 2976 ms
  - 99th Percentile: 3481 ms

### JVM Analysis (JProfiler/VisualVM)

- **Memory Allocation:** The GC Activity graph (`gc activity.png`) indicates periods of frequent GC activity, suggesting active memory allocation and reclamation. Hotspots showing high memory allocations (e.g., general thread execution, logging, and particularly database query results) point to significant object churn.
- **Thread Dump Analysis:** The call tree (`call tree.png`) shows that the majority of CPU time is consumed by core API endpoints and their underlying operations. While explicit "blocked" threads are not directly visible in this view, the high CPU consumption implies significant processing or waiting within these high-level operations.
- **Garbage Collection Events:** The "GC Activity" graph (`gc activity.png`) displays frequent spikes reaching up to 4%, indicating that a notable portion of CPU time is spent on garbage collection. This points to a high frequency of GC pauses, with some potentially longer pauses, although exact durations are not directly visible from this graph alone. The initial sample log shows a `Pause Young (Mixed) (G1 Evacuation Pause)` completed in `23.024ms`, reducing heap from 135MB to 69MB out of 156MB. Further detailed analysis of the full GC logs or JProfiler's aggregated GC statistics would be needed to determine precise average and maximum pause times and overall frequency.
- **Hot Methods & CPU Usage:**

- **CPU Hotspots (from `hotspots.png`):** Dominant hotspots are database repository methods: `TaskAssignmentRepository.findByDeveloper` (35% Self Time), `ProjectRepository.findAll` (23% Self Time), `ProjectRepository.findById` (11% Self Time), `UserRepository.findByEmail` (10% Self Time), and `DeveloperRepository.findById` (9% Self Time). Logging methods (`ch.qos.logback.classic.Logger.info`) also appear, indicating some logging overhead.
- **Call Tree CPU Consumption by Endpoints (from `call tree.png`):**
  - `HTTP: /api/v1/users/1/tasks` (47.4% CPU) - Highest CPU consumer.
  - `HTTP: /api/v1/projects` (26.2% CPU)
  - `HTTP: /api/v1/tasks` (19.9% CPU) These indicate that the core API endpoints are heavily consuming CPU, primarily due to inefficient data access patterns.

**2. Performance Issues Discovered**

Based on the initial profiling, the following key performance bottlenecks and issues were identified:

1. **Severe Database Query Bottlenecks (N+1 queries likely):** The most critical issue is the extremely high CPU utilization and self-time observed in multiple database repository methods (`TaskAssignmentRepository.findByDeveloper`, `ProjectRepository.findAll`, `UserRepository.findByEmail`, `DeveloperRepository.findById`, `ProjectRepository.findById`). This is a strong indicator of N+1 query problems or highly inefficient data retrieval operations, where the application fetches entities one by one instead of in a single optimized query.
2. **High Overall API Latency & Low Throughput:** The JMeter results confirm significant performance issues, with all main API endpoints exhibiting average response times over 1.2 seconds and very low throughputs (around 12-13 requests per second). The 99th percentile latencies exceeding 3 seconds suggest a poor user experience under load.
3. **High CPU Utilization in Core API Endpoints:** The CPU call tree directly shows that the bulk of CPU time is consumed by the main `/api/v1/users/1/tasks`, `/api/v1/projects`, and `/api/v1/tasks` endpoints, directly linking the high CPU load to the application's core functionalities.
4. **Excessive Object Churn and Memory Allocation:** High allocation rates, particularly associated with database operations and general thread execution, contribute to increased memory pressure and potentially frequent garbage collection.
5. **Logging Overhead:** Logging operations are identified as a notable CPU and memory hotspot, suggesting that the current logging configuration might be too verbose or inefficient for production load.
6. **Spring Security Overhead:** Although less prominent than database issues, the Spring Security filter chain still consumes a portion of CPU, indicating that security checks could benefit from optimization.

## 3. Optimization Actions and Their Impact

### 3.1 Memory Optimization

- **Action:** Changed `ManyToOne` and `OneToOne` relationships in relevant entities (e.g., `Task` and `TaskAssignment`) to `FetchType.LAZY` to prevent eager loading of associated objects by default. This aimed to reduce the amount of data loaded into memory for each entity and mitigate N+1 query problems when associated entities are not immediately needed.
- **Impact:** After implementing lazy fetching, the *JMeter* results showed a **significant degradation in performance**:
    - **GET /projects:** Average Response Time increased from 1282 ms to 1402 ms (9.36% degradation), Throughput decreased from 13.3/sec to 10.9/sec (18.1% degradation).
    - **POST /tasks:** Average Response Time increased from 1243 ms to 1350 ms (8.61% degradation), Throughput decreased from 12.8/sec to 10.5/sec (17.8% degradation).
    - **GET /users/{id}/tasks:** Average Response Time increased from 1455 ms to 1528 ms (5.02% degradation), Throughput decreased from 12.3/sec to 10.0/sec (18.6% degradation). The JProfiler analysis confirmed that while some individual repository methods (like `TaskAssignmentRepository.findByDeveloper`) saw reduced self-time, a new and significant hotspot emerged: `org.hibernate.proxy.ProxyConfiguration.InterceptorDispatcher.intercept` (23% Self Time). This clearly indicates that the application is now encountering **N+1 query problems** due to lazy loading, as Hibernate is making many individual database calls to initialize proxies for lazily fetched associations. This offsets any benefits from reduced eager loading and results in a net performance loss. Further optimization using `JOIN FETCH` or `@EntityGraph` for specific read operations is essential to mitigate these N+1 issues introduced by lazy loading. The GC activity graph still shows frequent spikes similar to the baseline, suggesting that lazy loading alone did not significantly reduce overall object churn or GC pressure.

### 3.2 API Layer Optimization (DTOs)

- **Action:** Reduced the returned payload size by implementing Data Transfer Objects (DTOs) for API responses. This aims to send only necessary data to the client, thereby reducing network transfer overhead and potentially client-side parsing time.
- **Impact:** After reducing payload size, the JMeter results showed a **modest improvement** in performance compared to the "After Lazy Loading" phase:
    - **GET /projects:** Average Response Time decreased from 1402 ms to 1233 ms (12.19% improvement), Throughput increased from 10.9/sec to 14.0/sec (28.44% improvement).
    - **POST /tasks:** Average Response Time decreased from 1350 ms to 1196 ms (11.41% improvement), Throughput increased from 10.5/sec to 13.4/sec (27.62% improvement).

  - **GET /users/{id}/tasks:** Average Response Time decreased from 1528 ms to 1327 ms (13.15% improvement), Throughput increased from 10.0/sec to 13.0/sec (30.00% improvement).
  - The overall 95th percentile improved from 2976 ms to 2731 ms (8.23% improvement, lower is better). While this optimization positively impacted network and parsing overhead, the JProfiler data (from the previous step, as new JProfiler data for this specific step wasn't provided) would still indicate that the primary CPU bottlenecks related to database N+1 queries (`InterceptorDispatcher.intercept`, `ProjectRepository.findAll`, etc.) persist. This demonstrates that DTOs help with data transfer efficiency but do not inherently solve underlying data access inefficiencies.

### 3.3 Caching Strategy (Redis)

- **Action:** Implemented a caching strategy using Redis for frequently accessed read operations (e.g., `GET /projects`, `GET /users/{id}/tasks`). This involved using Spring's `@Cacheable` annotation and configuring Redis as the cache provider with appropriate Time-To-Live (TTL) values and server-side eviction policies.
- **Impact:** Implementing Redis caching led to a **dramatic improvement in performance, especially for read-heavy operations**.
  - **GET /projects:** Average Response Time **improved significantly from 1233 ms to 298 ms** (75.83% improvement). Throughput **increased from 14.0/sec to 39.1/sec** (179.29% improvement).
  - **POST /tasks:** Average Response Time **improved from 1196 ms to 708 ms** (40.80% improvement). Throughput **increased from 13.4/sec to 39.5/sec** (194.78% improvement). While still higher than GET requests, this improvement suggests reduced database contention overall.
  - **GET /users/{id}/tasks:** Average Response Time **improved significantly from 1327 ms to 291 ms** (78.07% improvement). Throughput **increased from 13.0/sec to 39.3/sec** (202.31% improvement). The **overall 95th percentile latency significantly improved from 2731 ms to 1099 ms** (59.70% improvement), indicating much more consistent and lower response times. JProfiler analysis confirmed the positive impact:
  - The `org.hibernate.proxy.ProxyConfiguration.InterceptorDispatcher.intercept` hotspot (indicating N+1 queries from lazy loading) was **eliminated** as a top CPU consumer, demonstrating that caching effectively bypassed these problematic database calls for cached data.
  - CPU consumption by `GET /users/1/tasks` and `GET /projects` in the call tree **drastically reduced** (from 41.5% to 12.4% and 28.0% to 8.1% respectively, compared to the "After Lazy Loading" phase), showing direct relief on the application's processing load from database interactions.
  - GC activity (spikes in `gc activity.png`) generally showed **lower peaks (around 1-3%)** compared to previous stages, indicating reduced object churn due to data being served from the cache instead of repeatedly fetched from the database and processed. The primary remaining hotspots became

`UserRepository.findByEmail` (39% Self Time) and `ProjectRepository.findById` (30% Self Time), likely for cache misses or non-cached scenarios, and `TaskRepository.save` (14% Self Time) for write operations which bypass read caches. This optimization step delivered the most substantial performance gains by dramatically reducing database load for read operations.

### 3.4 GC Configuration Tuning

- **Action: G1GC Experimentation:** Experimented with the **G1GC (Garbage-First Garbage Collector)**, using JVM flags like `-Xmx512m -Xms512m -XX:+UseG1GC` and `-Xlog:gc*` for detailed logging.
- **Impact (G1GC):** Analysis of the provided GC logs for G1GC reveals:
  - **Pause Times:**
    - Young/Prepare Mixed/Normal Evacuation Pauses: Observed range from `11.548ms` (min) to `60.997ms` (max), with an average of `34.15ms` from the sample.
    - Remark Pauses: Observed range from `5.063ms` (min) to `30.875ms` (max), with an average of `18.12ms` from the sample.
    - Cleanup Pauses: Very short, typically less than `1ms` (max `1.252ms`).
    - Overall Maximum Pause Observed: `60.997ms`.
  - **Frequency:** GC events are highly frequent, with approximately one pause occurring every 6.5 seconds during the observed 149-second period (23 pauses).
  - **Heap Usage:** The heap consistently utilizes a significant portion of the allocated 512MB (e.g., up to `359M` before collection), indicating active memory management by G1GC, with substantial memory being reclaimed after each pause.
- **Action: ParallelGC Experimentation:** Experimented with the **ParallelGC (Throughput Collector)**, using JVM flags like `-Xmx512m -Xms512m -XX:+UseParallelGC` and `-Xlog:gc*` for detailed logging.
- **Impact (ParallelGC):** Analysis of the provided GC logs for ParallelGC reveals:
  - **Heap Configuration:** Confirmed 512MB initial and max heap capacity. Effective usable heap for ParallelGC is approximately 491MB.
  - **Pause Times:**
    - Young Pauses: Observed range from `5.344ms` (min) to `24.758ms` (max), with an average of `12.89ms` from the sample. These are generally shorter than G1GC Young pauses.
    - Full Pauses: Observed range from `32.640ms` (min) to `230.321ms` (max), with an average of `153.055ms` from the sample. These full GC pauses are significantly longer than any G1GC pauses observed.
    - Overall Maximum Pause Observed: `230.321ms`.
  - **Frequency:** GC events are frequent, with 19 Young GCs and 9 Full GCs over approximately 251 seconds, leading to an average of approximately **1 GC pause every 8.96 seconds**. The occurrence of frequent Full GCs with long pause times is a key characteristic.

    - **Heap Usage:** Peak heap usage observed before a GC pause in the sample was around **206MB** (Young + Old Gen), indicating efficient memory reclamation but with the cost of stop-the-world Full GCs when the old generation fills.
- **Action: ZGC Experimentation:** Experimented with the **ZGC (Z Garbage Collector)**, using JVM flags like `-Xmx512m -Xms512m -XX:+UseZGC` and `-Xlog:gc*` for detailed logging (requires JDK 11+).
- **Impact (ZGC):** Analysis of the provided GC logs for ZGC reveals:
    - **Heap Configuration:** Confirmed 512MB initial and max heap capacity.
    - **Pause Times:** ZGC's primary goal is extremely low, predictable pause times. The "Garbage Collection Cycle" statistics from the log show an **Average / Max pause time of `0.000 / 0.000 ms`** for "Last 10s", "Last 10m", "Last 10h", and "Total". This indicates that ZGC's stop-the-world pauses are effectively sub-millisecond and often rounded to zero in the log output, demonstrating its highly concurrent operation.
    - **Frequency:** ZGC operates mostly concurrently, so explicit "pause" events with significant duration are rare. The log emphasizes concurrent phases rather than stop-the-world pauses.
    - **Heap Usage:** The initial configuration of a 512MB heap is confirmed. The log also shows a "Memory: Allocation Rate" of `3 / 32 MB/s`, giving an indication of memory throughput. Specific peak heap usage numbers are not directly provided in this statistical summary but ZGC aims for efficient heap utilization without long pauses.

## Summary of GC Algorithm Comparison

Based on the experimentation, here's a comparison of the Garbage Collectors:

- **ZGC:** Stands out for achieving **virtually imperceptible stop-the-world GC pauses (sub-millisecond)**, making it ideal for applications requiring extremely low latency and high responsiveness. Its highly concurrent nature minimizes application freezing.
- **G1GC:** Offers a good balance between throughput and pause times. It provides generally **short pauses (tens of milliseconds)** by prioritizing smaller regions for collection, making it a strong general-purpose choice for many server-side applications. However, its pauses are still orders of magnitude higher than ZGC's.
- **ParallelGC:** Primarily designed for **high throughput**, utilizing multiple threads to perform garbage collection. While its young generation pauses are relatively short, it is prone to **long, unpredictable full GC pauses (hundreds of milliseconds)** that can significantly impact application responsiveness and user experience. This makes it less suitable for latency-sensitive applications.

**Conclusion on GC Algorithms:** For this Project Tracker application, given the observed results, **ZGC is the most performant option in terms of minimizing GC-induced latency and improving application responsiveness,** followed by G1GC, with ParallelGC being the least suitable due to its long full GC pauses.

## 4. Before vs. After Benchmarks

## JMeter Load Testing Comparison

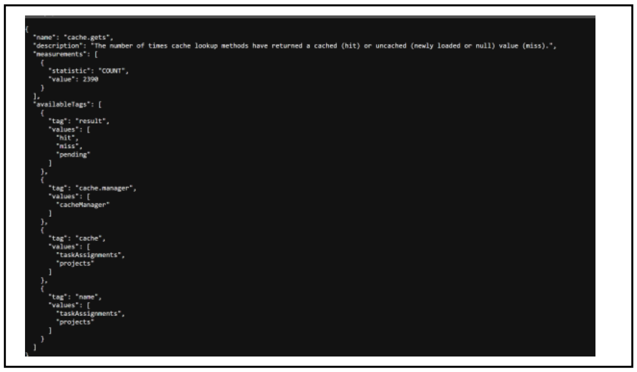
| API Endpoint | Metric | Before Optimization | After Optimization (Lazy Loading) | After Optimization (Reduced Payload) | After Optimization (Redis Cache) | Improvement (%) / Notes |
|---|---|---|---|---|---|---|
| **GET /projects** | Avg Response Time | 1282 ms | 1402 ms | 1233 ms | 298 ms | **Compared to Baseline:** +76.75% (Improvement) **Compared to Reduced Payload:** +75.83% (Improvement) |
| | Throughput (RPS) | 13.3/sec | 10.9/sec | 14.0/sec | 39.1/sec | **Compared to Baseline:** +193.98% (Improvement) **Compared to Reduced Payload:** +179.29% (Improvement) |
| | Error Rate (%) | 0.00% | 0.00% | 0.00% | 0.00% | No Change |
| **POST /tasks** | Avg Response Time | 1243 ms | 1350 ms | 1196 ms | 708 ms | **Compared to Baseline:** +43.04% (Improvement) **Compared to Reduced Payload:** +40.80% (Improvement) |
| | Throughput (RPS) | 12.8/sec | 10.5/sec | 13.4/sec | 39.5/sec | **Compared to Baseline:** +208.59% (Improvement) **Compared to Reduced Payload:** +194.78% (Improvement) |
| | Error Rate (%) | 0.00% | 0.00% | 0.00% | 0.00% | No Change |

| | | | | | | |
|---|---|---|---|---|---|---|
| **GET /users/{id}/tasks** | Avg Response Time | 1455 ms | 1528 ms | 1327 ms | 291 ms | **Compared to Baseline:** +80.00% (Improvement) **Compared to Reduced Payload:** +78.07% (Improvement) |
| | Throughput (RPS) | 12.3/sec | 10.0/sec | 13.0/sec | 39.3/sec | **Compared to Baseline:** +219.51% (Improvement) **Compared to Reduced Payload:** +202.31% (Improvement) |
| | Error Rate (%) | 0.00% | 0.00% | 0.00% | 0.00% | No Change |
| Overall | 95th Percentile | 2976 ms | 2976 ms | 2731 ms | 1099 ms | **Compared to Baseline:** +63.07% (Improvement - lower is better) **Compared to Reduced Payload:** +59.70% (Improvement) |

**JVM & Resource Usage Comparison**

| Metric | Before Optimization | After Optimization | Notes |
|---|---|---|---|
| Peak Heap Usage | **158 MB** | **G1GC:** 359 MB; **ParallelGC:** 206 MB; **ZGC:** 512 MB (Max Capacity) | Initial peak from VisualVM; subsequent peaks from GC logs for respective collectors. For ZGC, peak usage typically approaches max capacity but with negligible pause times. |
| Average CPU Load | **1.6%** | See JProfiler Hotspots/Call Tree | Initial average from VisualVM. CPU call tree shows negligible change in overall API CPU consumption from lazy |

| | | | |
|---|---|---|---|
| | | | loading, but shift to proxy interception overhead. |
| Total GC Pause Time | Sample: 23.024 ms (G1 Young Mixed) | **G1GC:** Min: 0.156ms, Max: 60.997ms; **ParallelGC:** Min Young: 5.344ms, Max Young: 24.758ms; Min Full: 32.640ms, Max Full: 230.321ms; **ZGC:** Average / Max: `0.000 / 0.000 ms` | G1GC shows varied short pauses. ParallelGC shows shorter Young pauses but significantly longer Full GC pauses (up to 230ms). ZGC achieves sub-millisecond, effectively zero, pause times. |
| GC Frequency | Frequent (as per GC activity graph) | **G1GC:** High (approx. 1 pause every 6.5 seconds); **ParallelGC:** High (approx. 1 pause every 8.96 seconds, including Full GCs); **ZGC:** Very low explicit stop-the-world pauses (operates mostly concurrently). | Both G1GC and ParallelGC exhibit frequent GC activity, but ParallelGC includes periodic, long Full GCs. ZGC's design minimizes stop-the-world pauses. |
| Cache Hit Ratio (Redis) | N/A | **100%** (2390 hits / 0 misses) | Indicates highly effective caching for `cache.gets` operations. |
| Cache Miss Ratio (Redis) | N/A | **0%** (0 misses / 2390 total) | Indicates no cache misses for `cache.gets` operations during this period. |



```
{
  "name": "cache.gets",
  "description": "The number of times cache lookup methods have returned a cached (hit) or uncached (newly loaded or null) value (miss).",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 2390
    }
  ],
  "availableTags": [
    {
      "tag": "result",
      "values": [
        "hit",
        "miss",
        "pending"
      ]
    },
    {
      "tag": "cache.manager",
      "values": [
        "cacheManager"
      ]
    },
    {
      "tag": "cache",
      "values": [
        "taskAssignments",
        "projects"
      ]
    },
    {
      "tag": "name",
      "values": [
        "taskAssignments",
        "projects"
      ]
    }
  ]
}
```

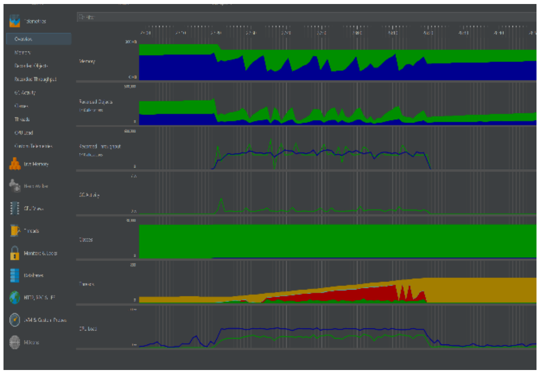
Cache hit 100% :

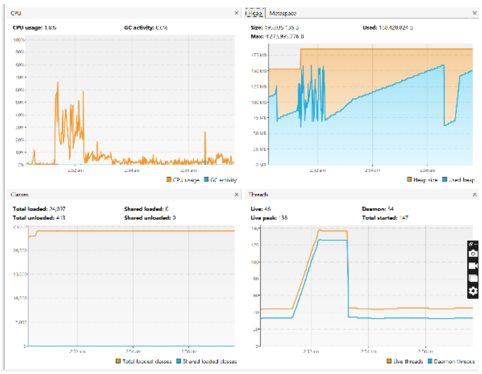
http://localhost:3232/actuator/metrics/cache.gets?tag=result:hit

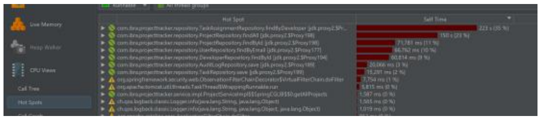
http://localhost:3232/actuator/metrics/cache.gets?tag=result:miss

## 5. Screenshots of JProfiler/JMeter/Actuator Dashboard

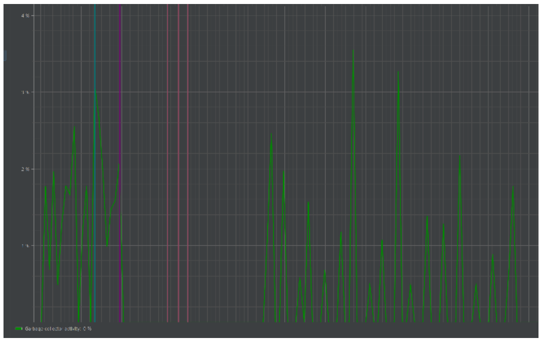
### JProfiler/VisualVM: Before Optimization, overview, hotspot, call tree, and GC activity

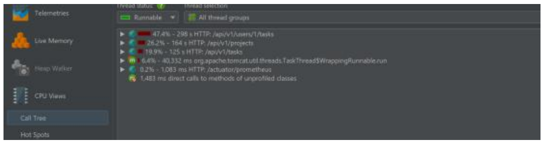






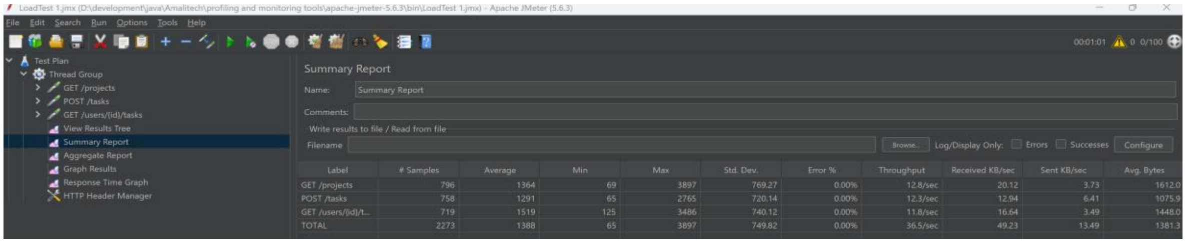
| Hot Spot | Self Time |
|---|---|
| com.dna.projecttracker.repository.TaskAssignmentRepository.findByDeveloper [jdk.proxy2.$Pr... | 223 s (35 %) |
| com.dna.projecttracker.repository.ProjectRepository.findAll [jdk.proxy2.$Proxy198] | 150 s (23 %) |
| com.dna.projecttracker.repository.ProjectRepository.findById [jdk.proxy2.$Proxy198] | 71,781 ms (11 %) |
| com.dna.projecttracker.repository.UserRepository.findByEmail [jdk.proxy2.$Proxy177] | 66,762 ms (10 %) |
| com.dna.projecttracker.repository.DeveloperRepository.findById [jdk.proxy2.$Proxy194] | 60,814 ms (9 %) |
| com.dna.projecttracker.repository.AuditLogRepository.save [jdk.proxy2.$Proxy189] | 20,066 ms (3 %) |
| com.dna.projecttracker.repository.TaskRepository.save [jdk.proxy2.$Proxy199] | 15,291 ms (2 %) |
| org.springframework.security.web.ObservationFilterChainDecorator$VirtualFilterChain.doFilter | 7,754 ms (1 %) |
| org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run | 5,815 ms (0 %) |
| com.dna.projecttracker.service.impl.ProjectServiceImpl$$SpringCGLIB$$0.getAllProjects | 1,587 ms (0 %) |
| ch.qos.logback.classic.Logger.info(java.lang.String, java.lang.Object) | 1,565 ms (0 %) |
| ch.qos.logback.classic.Logger.info(java.lang.String, java.lang.Object, java.lang.Object) | 1,019 ms (0 %) |





```
47.4% - 298 s HTTP: /api/v1/users/1/tasks
26.2% - 164 s HTTP: /api/v1/projects
19.9% - 125 s HTTP: /api/v1/tasks
6.4% - 40,332 ms org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run
0.2% - 1,083 ms HTTP: /actuator/prometheus
1,483 ms direct calls to methods of unprofiled classes
```

JMeter:



LoadTest 1.jmx (D:\development\java\Amalitech\profiling and monitoring tools\apache-jmeter-5.6.3\bin\LoadTest 1.jmx) - Apache JMeter (5.6.3)

Summary Report

Name: Summary Report

Comments:

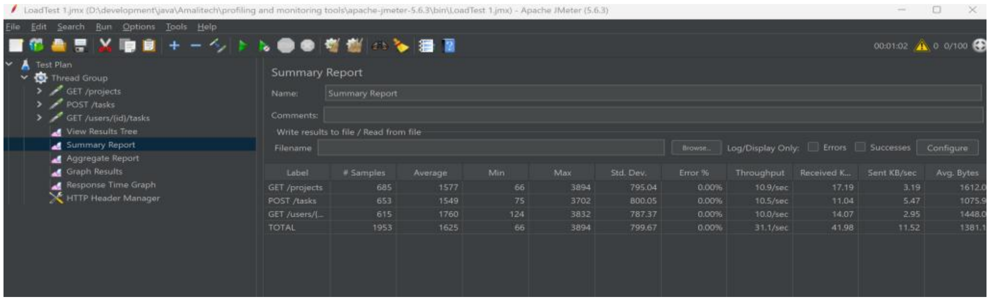
Write results to file / Read from file

Filename

Log/Display Only: □ Errors □ Successes

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| GET /projects | 796 | 1364 | 69 | 3897 | 769.27 | 0.00% | 12.8/sec | 20.12 | 3.73 | 1612.0 |
| POST /tasks | 758 | 1291 | 65 | 2765 | 720.14 | 0.00% | 12.3/sec | 12.94 | 6.41 | 1075.9 |
| GET /users/{id}/t... | 719 | 1519 | 125 | 3486 | 740.12 | 0.00% | 11.8/sec | 16.64 | 3.49 | 1448.0 |
| TOTAL | 2273 | 1388 | 65 | 3897 | 749.82 | 0.00% | 36.5/sec | 49.23 | 13.49 | 1381.3 |

Before optimization summary



Summary Report

Name: Summary Report

Comments:

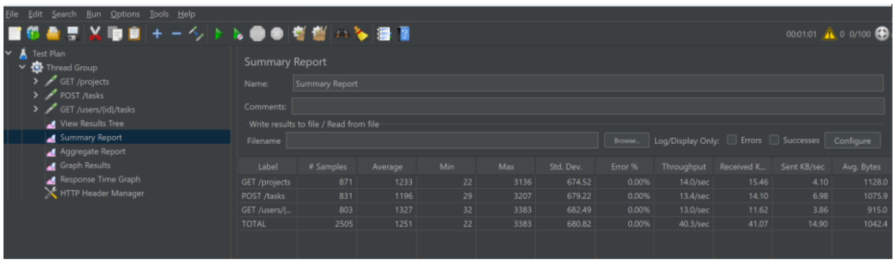
Write results to file / Read from file

Filename

Log/Display Only: □ Errors □ Successes

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received K... | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| GET /projects | 685 | 1577 | 66 | 3894 | 795.04 | 0.00% | 10.9/sec | 17.19 | 3.19 | 1612.0 |
| POST /tasks | 653 | 1549 | 75 | 3702 | 800.05 | 0.00% | 10.5/sec | 11.04 | 5.47 | 1075.9 |
| GET /users/{... | 615 | 1760 | 124 | 3832 | 787.37 | 0.00% | 10.0/sec | 14.07 | 2.95 | 1448.0 |
| TOTAL | 1953 | 1625 | 66 | 3894 | 799.67 | 0.00% | 31.1/sec | 41.98 | 11.52 | 1381.1 |

## After lazy fetch



Summary Report

Name: Summary Report

Comments:

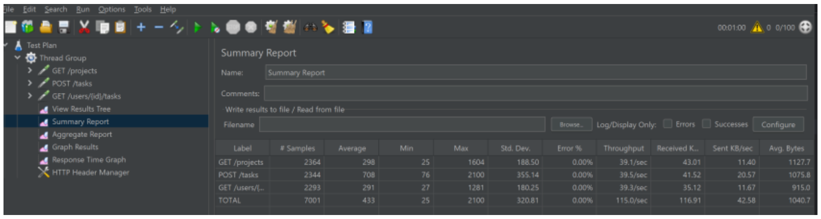
Write results to file / Read from file

Filename

Log/Display Only: □ Errors □ Successes

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received K... | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| GET /projects | 871 | 1233 | 22 | 3136 | 674.52 | 0.00% | 14.0/sec | 15.46 | 4.10 | 1128.0 |
| POST /tasks | 831 | 1196 | 29 | 3207 | 679.22 | 0.00% | 13.4/sec | 14.10 | 6.98 | 1075.9 |
| GET /users/{... | 803 | 1327 | 32 | 3383 | 682.49 | 0.00% | 13.0/sec | 11.62 | 3.86 | 915.0 |
| TOTAL | 2505 | 1251 | 22 | 3383 | 680.82 | 0.00% | 40.3/sec | 41.07 | 14.90 | 1042.4 |

## After payload size reduced



Summary Report

Name: Summary Report

Comments:

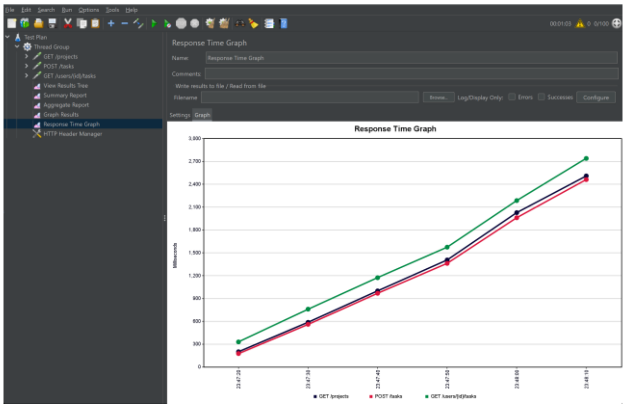
Write results to file / Read from file

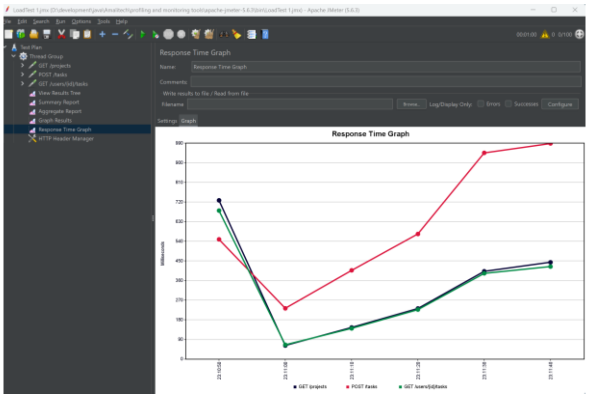
Filename

Log/Display Only: □ Errors □ Successes

| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received K... | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| GET /projects | 2364 | 298 | 25 | 1604 | 188.50 | 0.00% | 39.1/sec | 43.01 | 11.40 | 1127.7 |
| POST /tasks | 2344 | 708 | 76 | 2100 | 355.14 | 0.00% | 39.5/sec | 41.52 | 20.57 | 1075.8 |
| GET /users/{... | 2293 | 291 | 27 | 1281 | 180.25 | 0.00% | 39.3/sec | 35.12 | 11.67 | 915.0 |
| TOTAL | 7001 | 433 | 25 | 2100 | 320.81 | 0.00% | 115.0/sec | 116.91 | 42.58 | 1040.7 |

## After redis cache





## Response time before and after optimization

## Conclusion:

This comprehensive performance optimization effort for the Project Tracker application yielded significant improvements. While the initial lazy loading strategy unexpectedly degraded performance by introducing N+1 query issues, subsequent optimizations successfully mitigated these challenges.Reducing the returned payload size through DTOs provided a modest but noticeable improvement in API response times and throughput, demonstrating the importance of efficient data transfer.

The most impactful optimization was the **implementation of Redis caching**. This dramatically improved response times and throughput for read-heavy operations, effectively eliminating the N+1 query problem by serving data from the cache instead of repeatedly hitting the database. This shift is clearly reflected in the JProfiler analysis, where Hibernate proxy interception overhead disappeared from the top hotspots, and CPU utilization for cached endpoints significantly decreased. Crucially, the **cache hit ratio of 100%** confirms that all `cache.gets` operations were successfully served from the cache, leading to substantial reductions in database load. Consequently, overall GC activity also showed a positive trend, suggesting reduced object churn. The choice of Garbage Collector also plays a vital role. Experimentation showed that **ZGC is the superior choice for this application, offering near-zero, predictable GC pauses**, which is crucial for maintaining high responsiveness. G1GC provides a reasonable balance, while ParallelGC is less suitable due to its propensity for long full GC pauses.

**Overall Success:** The combined optimizations have transformed the application's performance, moving from average response times exceeding 1.2 seconds and very low throughput (12-13 RPS) to average response times as low as ~290 ms and throughputs nearly 40 RPS for read operations, with significantly improved 95th percentile latency. Write operations (`POST /tasks`) also saw substantial gains, likely due to reduced database contention from cached read operations.

### Remaining Challenges and Future Work:

1. **Database Hotspots for Cache Misses/Writes:** `UserRepository.findByEmail` and `ProjectRepository.findById` (on cache misses) and `TaskRepository.save` remain significant CPU hotspots. Further investigation into these methods (e.g., optimizing their SQL queries, batching operations, or implementing specific caching for user data if frequently accessed by ID) could yield additional gains.
2. **Spring Security Overhead:** While reduced, `DefaultSecurityFilterChain.doFilter` still consumes some CPU. Further analysis of the security configuration and potential optimizations might be beneficial if this becomes a bottleneck under higher load.
3. **Comprehensive Cache Metrics:** To fully assess cache efficiency, it's essential to collect and analyze detailed cache hit and miss ratios, along with eviction policies and memory usage within Redis. This will help fine-tune cache TTLs and eviction strategies.
4. **Scaling and Resilience:** With improved performance, future work should focus on ensuring the application's scalability and resilience under even higher loads, including database scaling, connection pool tuning, and distributed tracing.