Java Performance

Shalini Mittal
Corporate Trainer

Performance Bottleneck

- Scenario A: CPU-Bound Issues
  - Product Recommendation Engine
  - Complex ML algorithms running on main thread
  - Synchronous processing of recommendation matrix
  - Heavy JSON parsing for product attributes
  - Inefficient string concatenation in logging
- Symptoms:
  - High CPU utilization (>90%)
  - Slow response times during peak hours
  - Thread contention in recommendation service

Performance Bottleneck

- Scenario B: I/O-Bound Issues
    - Inventory Management System
    - Database queries without proper indexing
    - Synchronous API calls to supplier systems
    - File-based product image loading
    - Network timeouts in distributed cache access
- Symptoms:
    - Low CPU usage (<30%) but slow responses
    - Database connection pool exhaustion
    - High network latency
    - Blocked threads waiting for I/O operations

Performance Bottleneck

- Scenario C: Memory-Bound Issues
    - Customer Data Service
    - Loading entire customer history into memory
    - Inefficient caching of product catalogs
    - Memory leaks in session management
    - Large object graphs for order processing
- Symptoms:
    - Frequent garbage collection
    - OutOfMemoryError during peak loads
    - Increasing response times over time
    - Memory usage climbing continuously

Profiling

- Profiling is an essential step in performance tuning that involves analysing your application’s runtime behaviour to detect performance issues and bottlenecks.
- Profiling provides detailed insights that can guide your optimization efforts more effectively than mere guesswork
- Types of Profiling
    - **CPU Profiling:** Focuses on identifying functions or methods that consume the most processor time. This type of profiling helps developers understand which parts of their code are the most computationally expensive.
    - **Memory Profiling:** Identifies how your application uses memory and helps track down memory leaks, excessive garbage collection activity, and inefficient memory usage patterns.
    - **Thread Profiling:** Helps analyze issues related to multithreading, such as deadlocks, thread contention, and suboptimal synchronization mechanisms.
    - **Higher Level Profiling:** Higher level profiling is about monitoring the behavior of technical application functions. For example, the execution of queries against a database (JDBC, JPA, ...), web requests (servlets, ...) or the throwing of exceptions.
    - **Application Profiling:** Application profiling is about making the behavior of your own application logic visible. To do this, you often need to implement your own metrics in the application that make sense in context.

Application Variants

- Depending on the situation, profilers can be used in different ways.
- **Attached:** A process is started that is prepared (instrumentalized) with the corresponding profiler.
- **Local:** The profiler establishes a connection to a JVM process that is already running locally.
- **Remote:** The profiler establishes a connection to a remotely running process via a corresponding protocol. To do this, the protocol must be set up accordingly on the server side.
- **Snapshot:** An image of a system is created at a point in time in order to then evaluate the image in detail. As a snapshot only records one point in time, the information it contains can be very detailed. A special snapshot are Java Flight Recorder recordings. The information and metrics are collected by a program over a period of time and stored temporarily in a file. The data can then be analyzed later in a profiler. Especially for resource-intensive surveys, it is often a good option to make targeted and short recordings.

Profiling Tools

- **VisualVM:** A versatile tool that not only provides CPU and memory profiling but also features capabilities for thread analysis and heap dump analysis. After installing VisualVM and launching your application, you can connect VisualVM to your running application to start gathering data. The tool's interface allows for real-time performance monitoring, and its plugins enhance its functionality further.
- **JProfiler:** Offers a rich user interface with dynamic and configurable views that can adapt to the specific details of your application. JProfiler can be used to perform on-demand profiling or continuous monitoring, making it suitable for development as well as production environments.
- **JDK Mission Control:**
  It's a profiling and monitoring tool for the JVM that comes from Oracle, designed to work hand-in-hand with the Java Flight Recorder (JFR).

https://blog.doubleslash.de/en/software-technologien/hands-on-visualvm-java-mission-control-jprofiler-yourkit-java-profiler/

JDK Mission Control

- Let’s you inspect detailed runtime data from a running Java application. And find
    - Performance bottlenecks
    - Excessive GC activity
    - Memory leaks
    - Thread contention & deadlocks
    - Class loading issues
    - Can connect to a running JVM or open a .jfr recording file captured earlier.
- How it works:
    - Java Flight Recorder (JFR) collects extremely detailed telemetry from the JVM with very low overhead.
    - JMC visualizes and analyzes that data in an interactive GUI — think of it like VisualVM but with deeper, more precise JVM internals
- Use Cases:
    - Garbage Collection tuning — see exactly when minor/major GCs happen and how they affect heap.
    - Memory leak detection — identify which objects are retaining memory.
    - Performance profiling — method-level CPU usage with wall-clock and thread states.
    - Thread analysis — see blocking and deadlocks over time.

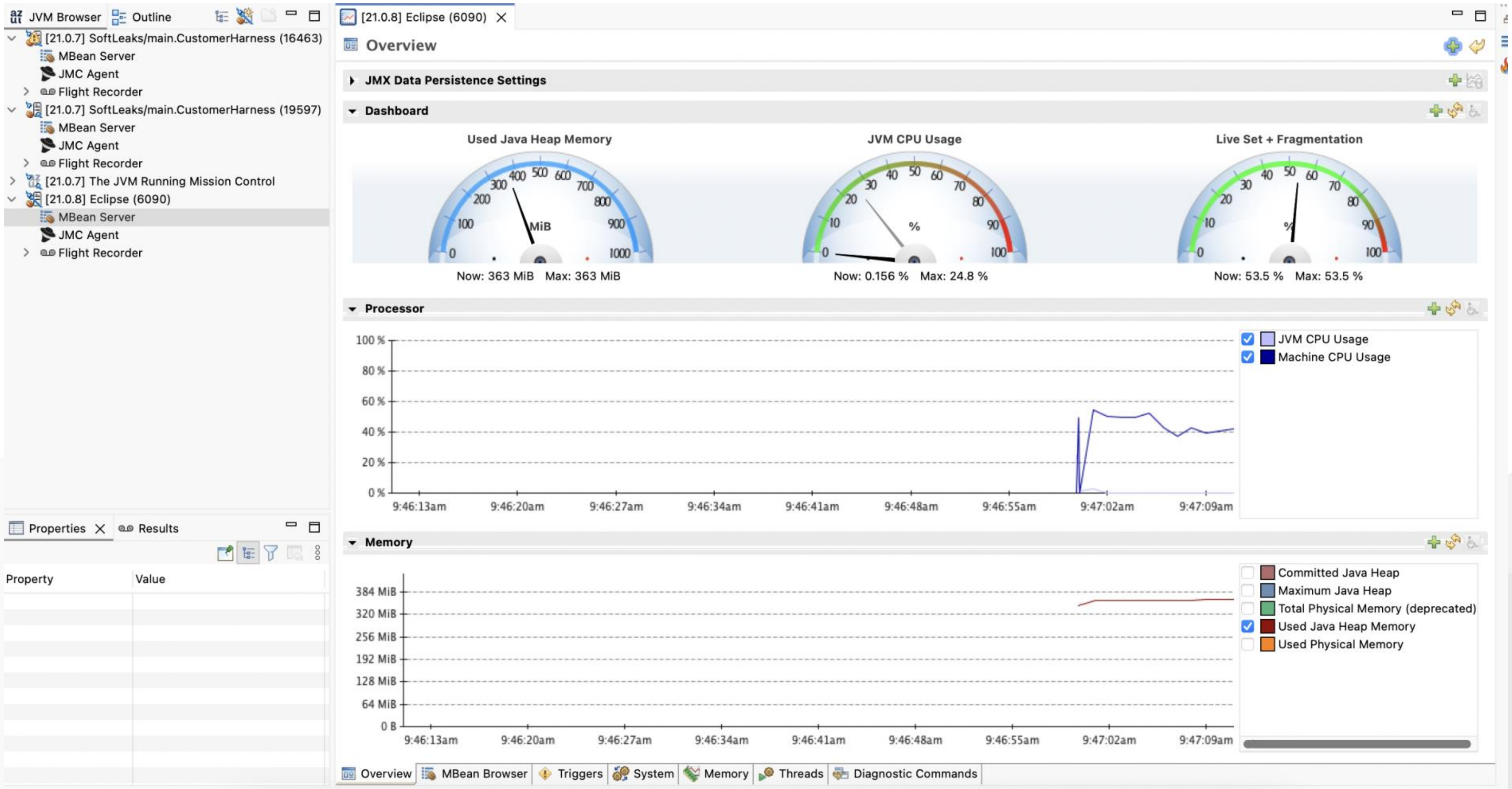
https://github.com/openjdk/jmc

JDK Mission Control GUI

- Open Window menu, select Show View and then Other to open necessary views.
- **JVM Browser:**
  Lists all the JVM instances running locally (on the host) and JVMs discovered on the network. The JVM Browser can be viewed in two different modes: as a flat list, and as a tree (visible by default).
- **Outline:**
  Shows the data collected in a Flight Recording. It organizes and presents flight recording data as pages in a tree for easy navigation (visible by default).
- **Progress View:**
  Displays the progress of running operations, for example, a flight recording.
- **Properties:**
  Lists the properties of items that you select in tables, including hidden properties that are not displayed in the tables (visible by default).
- **Results:**
  Displays a list of rules with their corresponding scores (visible by default). It also shows the results from the automated analysis relevant to the currently opened page in the editor.
- **Stack Trace:**
  Displays stack traces for the recorded events (visible by default).

JMC Dashboard

- By default, the JMX Console displays the **Overview** tab with the **JMX Data Persistence Settings** and **Dashboard** panels. It also displays the **Processor** and **Memory** charts.
- The default dials on the **Dashboard** show information about memory utilization, CPU usage in the JVM, and live set and fragmentation.
- The **Overview** tab allows you to add, remove, and edit dials and charts.



MBean Server

- Java Management console (JMX) connects to a running JVM and collects and displays key characteristics in real time.
- The tool presents live data about memory and CPU usage, garbage collections, thread activity, and so on.
- It also includes a fully featured JMX MBean browser that you can use to monitor and manage MBeans in the JVM and in your Java application.
- The MBean Server serves as the intermediary between the managed resources (represented by MBeans) and the management applications that interact with them, providing a standardized and centralized mechanism for monitoring and controlling Java applications and the Java Virtual Machine itself.

https://docs.oracle.com/en/java/java-components/jdk-mission-control/9/user-guide/real-time-jmx-monitoring.html#GUID-CC337C22-195A-469C-8B8D-D71AEEEEE111ca

FibonacciPrimes
FibonacciPrimesImproved

BenchMarking

- What:
  Systematic measurement of code performance under controlled conditions to measure its efficiency, scalability, and overall performance.
- Crucial for quantifying their performance and comparing the impacts of different code changes.
- Why:
  - Prevent premature optimization
  - Make data-driven tuning decisions
  - Validate performance after code changes
  - Support before/after comparisons
  - Detect regressions in CI/CD pipelines
  - Identify performance bottlenecks
  - Compare alternative implementations
  - Track performance regressions over time

- .

Best Practices

- Use dedicated benchmarking tools (e.g., JMH)
- Run multiple iterations & forks
- Separate production and benchmark code
- Minimize external interference (close apps, fix CPU scaling)
- Always measure before & after tuning

Common Benchmarking Techniques

- Using System time : measure the execution time of a piece of code

```
long startTime = System.nanoTime();
// Your code to benchmark goes here // ...
long endTime = System.nanoTime();
long elapsedTime = endTime - startTime;
System.out.println("Elapsed time: " + elapsedTime + " nanoseconds"); }
```

- This method can be affected by factors such as garbage collection, CPU scheduling, and other system processes, which can lead to inaccurate results.
- Therefore, it is recommended to use more advanced benchmarking techniques for more accurate performance measurements.

- Other methods include:
    - Using JMH
    - Using Frameworks like Google Caliper, EJML Benchmarking, Dropwizard Metrics

JMH Overview

- Java Microbenchmark Harness by OpenJDK team
- Features:
- Control Over JVM Optimizations:
  JMH provides mechanisms to control and account for JVM optimizations like method inlining and loop unrolling, which can skew microbenchmark results.
- Benchmark Modes:
  JMH supports various modes of operation, like measuring average time, throughput, sample time, and more. This flexibility allows developers to focus on the specific performance metrics that matter for their code.
- Multithreaded Benchmarking:
  JMH allows benchmarking in a multithreaded context, essential for modern, concurrent applications.
- Parameterization:
  JMH benchmarks can be parameterized to run with different inputs, enabling more comprehensive performance analysis.
- Result Profiling:
  It integrates with Java profilers, allowing deeper insight into the benchmark behavior beyond simple metrics.

JMH Modes:

- **Throughout(thrpt):** Number of operations per unit of time. (ops/sec, ops/ms, etc.)
    - Focus: Capacity / throughput of your code.
    - Use case: When you want to know how many requests/transactions/operations per second your method can handle.
    - Example: Testing a web service handler → "Can it process 50k requests/sec?"
- **AverageTime** (Mode.AverageTime or avgt): The average time taken by a single benchmark method call.
    - Focus: Latency of individual operations.
    - Use case: When you care about how long a single operation takes on average.
    - Example: Testing HashSet.contains(x) → "On average, how many nanoseconds does it take?"
- **SampleTime** (Mode.SampleTime or sample): Measures execution time of operations by sampling. Produces a distribution (histogram) of times instead of just averages.
    - Focus: Variance and tail latencies.
    - Use case: When you want to see how response time varies (e.g., 95th percentile latency).
    - Example: If some operations take 1μs but occasionally one takes 10ms, sample mode will reveal it.
- **Single Shot Time (ss):** Measures the time for a single operation (useful for cold-star measurements)

JMH Dependency

- mvn archetype:generate
-   -DinteractiveMode=false
-     -DarchetypeGroupId=org.openjdk.jmh
-     -DarchetypeArtifactId=jmh-java-benchmark-archetype
-     -DgroupId=org.sample
-     -DartifactId=benchmarkingdemo
-     -Dversion=1.0
- OR

-
  ```
  <dependency>
  <groupId>org.openjdk.jmh</groupId>
  <artifactId>jmh-core</artifactId>
  <version>1.32</version>
  <scope>test</scope>
  </dependency>
  <dependency>
  <groupId>org.openjdk.jmh</groupId>
  <artifactId>jmh-generator-annprocess</artifactId>
  <version>1.32</version>
  <scope>test</scope>
  </dependency>
  ```

Common Annotations

- @Benchmark: is used to annotate a benchmark method.
- @BenchmarkMode : is used to set the benchmark mode, the possible enum values you can set this to are Throughput, AverageTime, SampleTime, SingleShotTime or All. See JMHSample_02_BenchmarkModes for more details.
- @OutputTimeUnit : is used to set the unit of time to be used in your performance metrics including Nano/ Micro/ Milliseconds, Seconds, Minutes, Hours or Days.
- @State : allows us to define the state of fields with a scope of either per benchmark or per thread. The thread scope can be useful if running multi-threaded benchmarks. If we only need one state object this annotation can be used at the class level, otherwise we can use it on inner classes. See JMHSample_03_States for more details.
- @Param : is used to mark a configurable parameter, @Param fields should be non-final fields and should only reside in @State classes. The annotation value is given in String, and will be coerced as required to match the field type.
  See JMHSample_27_Params for more details.

Common Annotations

- @Setup & @Teardown: are used to annotate fixture methods that act on @State objects, these can be used for setup or housekeeping tasks outside of the measured code section. See JMHSample_05_StateFixtures for more details.
- @Warmup & @Measurement: are similar in that they share common options you can set, these are the number of iterations, the time, the timeUnit or the batchSize. Obviously @Warmup is for the warm up runs, which are not measured, and @Measurement is for the measured runs.
- @Fork: is used to set the number of JVM forks. Forking (running in a separate process) is enabled by default in the JMH. Running benchmarks within the same JVM is wrong because of how the JVM optimises an application by creating a profile of the application's behaviour. The fork is created to reset this profile and ensures that benchmarks are unaffected by other benchmarks or the order they are run in.

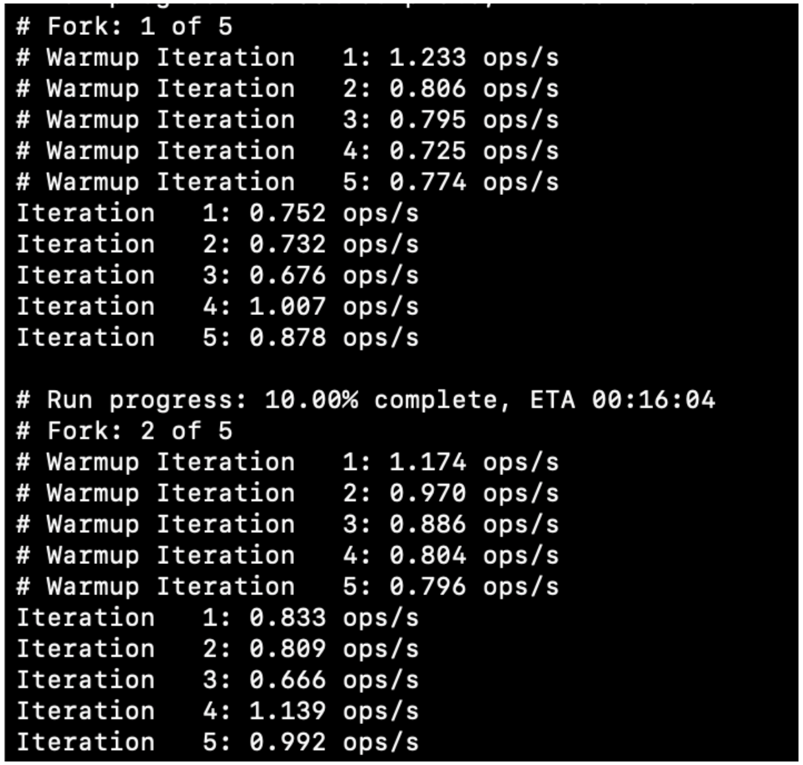
- **WARMUP**
- A key feature of the JMH is how it performs warm up iterations before measuring performance. It takes time for a Java application to reach optimal performance, this is mainly due to the Just-In-Time (JIT) compiler and how it improves performance by compiling bytecode to optimised machine code at runtime. Skipping these early iterations is essential in order to get more accurate results.

Run Benchmarking

- mvn clean install: cd target/: java –jar benchmarks.jar (This takes time)
- JMH repeatedly calls @Benchmark method.
- For some of those calls, it records the start & end time → that’s one sample.
- It doesn’t measure every call, otherwise measurement overhead would distort results.
- At the end, JMH reports statistics over all those samples:
- Mean (average execution time), Min / Max, Standard deviation, Percentiles (like 90%, 95%, 99%)

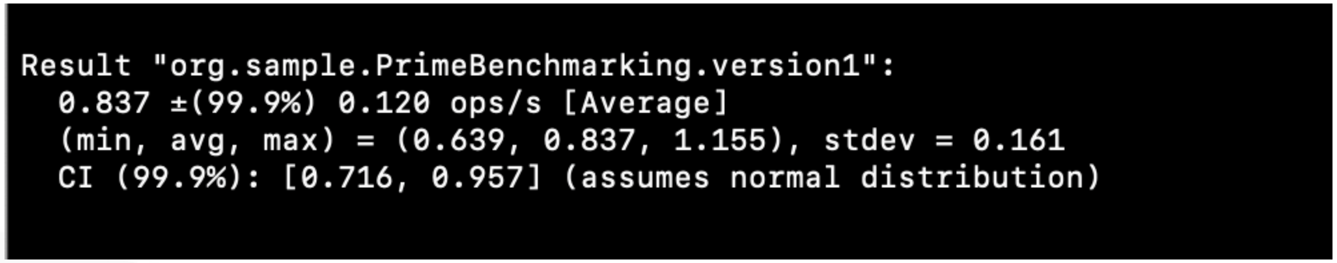
Output 1:

- Warmup: stabilizes JIT & caches
- Iteration: actual measured runs
- ops/ms: number of operations per millisecond

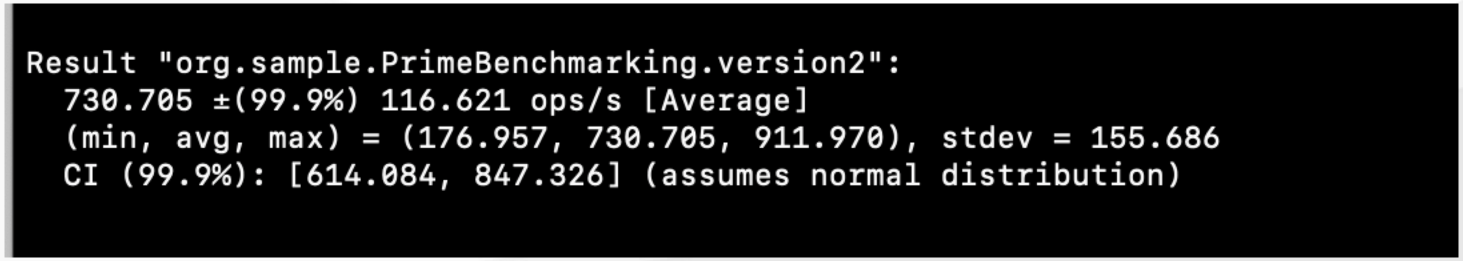


```
# Fork: 1 of 5
# Warmup Iteration   1: 1.233 ops/s
# Warmup Iteration   2: 0.806 ops/s
# Warmup Iteration   3: 0.795 ops/s
# Warmup Iteration   4: 0.725 ops/s
# Warmup Iteration   5: 0.774 ops/s
Iteration   1: 0.752 ops/s
Iteration   2: 0.732 ops/s
Iteration   3: 0.676 ops/s
Iteration   4: 1.007 ops/s
Iteration   5: 0.878 ops/s

# Run progress: 10.00% complete, ETA 00:16:04
# Fork: 2 of 5
# Warmup Iteration   1: 1.174 ops/s
# Warmup Iteration   2: 0.970 ops/s
# Warmup Iteration   3: 0.886 ops/s
# Warmup Iteration   4: 0.804 ops/s
# Warmup Iteration   5: 0.796 ops/s
Iteration   1: 0.833 ops/s
Iteration   2: 0.809 ops/s
Iteration   3: 0.666 ops/s
Iteration   4: 1.139 ops/s
Iteration   5: 0.992 ops/s
```



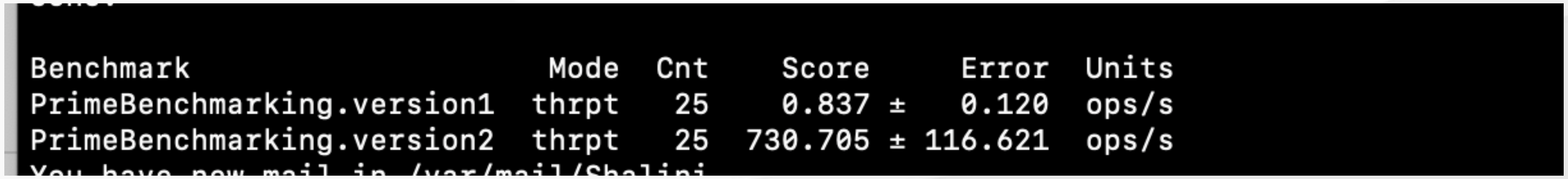
```
Result "org.sample.PrimeBenchmarking.version1":
  0.837 ±(99.9%) 0.120 ops/s [Average]
  (min, avg, max) = (0.639, 0.837, 1.155), stdev = 0.161
  CI (99.9%): [0.716, 0.957] (assumes normal distribution)
```



```
Result "org.sample.PrimeBenchmarking.version2":
  730.705 ±(99.9%) 116.621 ops/s [Average]
  (min, avg, max) = (176.957, 730.705, 911.970), stdev = 155.686
  CI (99.9%): [614.084, 847.326] (assumes normal distribution)
```

Benchmark Report

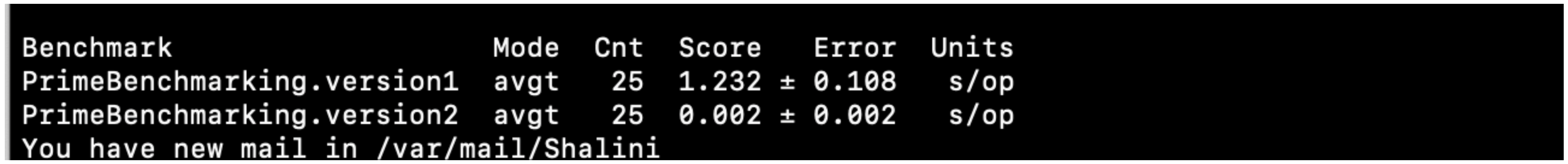
- Benchmark: This column names the benchmarked method.
- Mode: The benchmark mode (avgt in this case stands for average time taken).
- Cnt: Number of benchmark iterations (20 in this example).
- Score: The primary result of the benchmark, here in nanoseconds per operation.
- Error: The error margin of the score, indicating the variability of results.
- Units: The units of the score, such as ns/op (nanoseconds per operation).
- Score: This is the average time taken per operation (when in avgt mode). A lower score generally indicates better performance. However, the context and nature of the benchmark are important for interpretation.
- Error: Also known as the margin of error, this shows the variability in the benchmark results. A smaller error range indicates more consistent performance.
- Units: Common units include ns/op (nanoseconds per operation), us/op (microseconds per operation), ms/op (milliseconds per operation), and ops/s (operations per second). The choice of units depends on the benchmark mode and what you are measuring.



```
Benchmark                      Mode  Cnt    Score     Error  Units
PrimeBenchmarking.version1    thrpt   25    0.837 ±   0.120  ops/s
PrimeBenchmarking.version2    thrpt   25  730.705 ± 116.621  ops/s
```

Output with modes:

- thrpt - throughput
- java –jar benchmarks.jar -h
- Java –jar benchmarks.jar -bm avgt



```
Benchmark                      Mode  Cnt  Score   Error  Units
PrimeBenchmarking.version1     avgt   25  1.232 ± 0.108   s/op
PrimeBenchmarking.version2     avgt   25  0.002 ± 0.002   s/op
You have new mail in /var/mail/Shalini
```

List & Map in Java

- ArrayList : Implemneted as an array with default capacity as 10. Hence better to specify a capacity if known.
  ListBenchmarking Project
- CopyOnWriteArrayList : thread safe version of ArrayList but too costly if mutating.
- LinkedList
- Stack : legacy code
  Use ArrayDeque for single-threaded stacks (faster, modern).
  Use ConcurrentLinkedDeque if you need a thread-safe stack.
- Vector: Uses an array and array will resize overtime and it is thread safe
- LinkedList : No array to store but there are pointers. Add/removal will not resize the arrays
- HashMap: : Key-value pair. Time taken to retrieve any value is same irrespective of the size. Items in random order
- LinkedHashMap : Iterate in defined order

ListBenchmarking Project

Microservices Performance Optimization

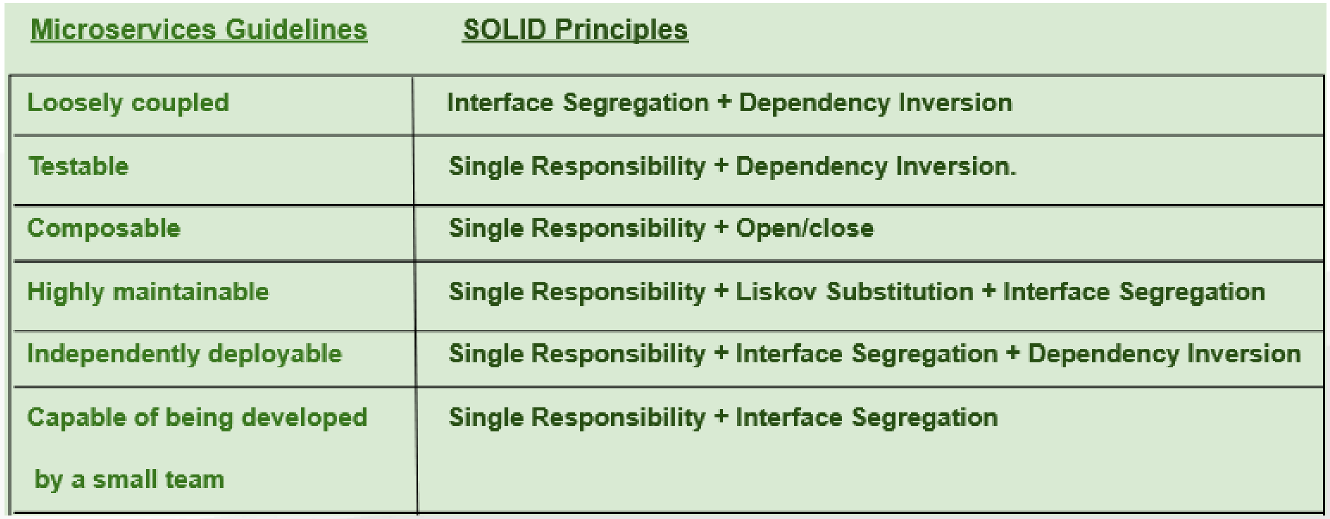
- **Choose the right communication protocol:**
- There are different communication protocols available, such as HTTP, gRPC, AMQP, MQTT, and others, each with its own advantages and disadvantages.
- **Implement caching and load balancing**
- Caching is the process of storing frequently accessed or expensive data in a fast and accessible storage layer, such as memory or disk, to reduce the number of requests to the original data source, such as a database or an external API.
- Caching can improve the response time, availability, and scalability of your microservices, as well as reduce the load on your resources.
- Load balancing is the process of distributing the incoming requests among multiple instances of your services, to ensure that no single service is overloaded or becomes a bottleneck. Load balancing can also improve the performance, reliability, and fault tolerance of your microservices, as well as enable horizontal scaling and failover.

Microservices Performance Optimization

- **Monitor and optimize resource usage:**
- Monitor and optimize the resource usage of your services, such as CPU, memory, disk, and network.
- By using appropriate tools and metrics, you can track the performance and behavior of your services, identify any issues or bottlenecks, and apply corrective actions or improvements.
- For example, you can use tools like Prometheus, Grafana, or Zipkin to collect and visualize data about your microservices' health, performance, and dependencies.
- You can also use tools like Docker or Kubernetes to manage and optimize the resource allocation and utilization of your services, by applying techniques like containerization, orchestration, resource limits, autoscaling, or service mesh.

Microservices Performance Optimization

- **Best Coding Practices**
- By following some common principles and guidelines, you can improve the quality, maintainability, and efficiency of your code, as well as avoid some common pitfalls and errors.
- For example, you can apply the SOLID principles, which are a set of design principles for object-oriented programming that promote cohesion, coupling, abstraction, and polymorphism.
- You can also apply the DRY principle, which stands for Don't Repeat Yourself, and encourages code reuse and modularity.
- Moreover, you can use code analysis tools, such as SonarQube or Codacy, to detect and fix code smells, bugs, vulnerabilities, or performance issues in your code.



Microservices Guidelines	SOLID Principles
Loosely coupled	Interface Segregation + Dependency Inversion
Testable	Single Responsibility + Dependency Inversion.
Composable	Single Responsibility + Open/close
Highly maintainable	Single Responsibility + Liskov Substitution + Interface Segregation
Independently deployable	Single Responsibility + Interface Segregation + Dependency Inversion
Capable of being developed by a small team	Single Responsibility + Interface Segregation

Microservices Performance Optimization

- **Test and benchmark your services**
- By testing your services, you can ensure that they meet the functional and non-functional requirements, such as correctness, reliability, security, and performance.
- By benchmarking your services, you can measure and compare their performance under different conditions, such as load, concurrency, or network latency.
- Testing and benchmarking can help you identify and resolve any performance problems or bottlenecks, as well as optimize and improve your services' performance.
- For example, you can use tools like JMeter, Gatling, or Locust to perform load testing and performance testing on your services.
- You can also use tools like Apache Benchmark, WRK, or Siege to perform benchmarking and stress testing on your services.

Microservices Performance Optimization

- **Update and upgrade your services**
- By updating your services, you can apply the latest patches, bug fixes, or security updates to your code, libraries, frameworks, or dependencies.
- By upgrading your services, you can adopt the latest features, enhancements, or improvements to your code, libraries, frameworks, or dependencies.
- Updating and upgrading can help you keep your services up to date, secure, and efficient, as well as take advantage of the latest technologies and innovations.
- For example, you can use tools like Git, Maven, or NPM to manage and update your code and dependencies.
- You can also use tools like Jenkins, Travis CI, or GitHub Actions to automate and streamline your update and upgrade processes.

Microservices Performance Optimization

- **Choose Best Fit Technology**
- Choose the technology stack based on the functionalities and business features/use cases.
- Example: AI/ML-based services can be built on Python or a similar language. If we try to build AI/ML models in JAVA, they might not work as expected in terms of performance.
- Once we have identified the best-fit tech stack then we should design and build optimized code for that tech stack **by following the technology’s best practices.**

Microservices Performance Optimization

- **Design Microservice Architecture for Performance and Security**:
- The security and APIs of the microservices should be in the design starters and not the last item of implementation.
- As unsecured services cause more harm to the consumers compared to its benefit.
- OAUTH and Kerberos are well-used security principles and their libraries are readily available in most programming languages.

Microservices Performance Optimization

- **Microservice communication: Asynchronous (Non-Blocking) requests wherever possible**:
- Synchronous requests are blocking and can cause serious performance bottlenecks.
- Asynchronous communication refers to the exchange of data/information/messages between two or more services without the requirement for all the services to respond immediately.
- In simple words, the interacting services are not required to be up and running during communication.
- This can be achieved via either Messaging queues or Database polling. Kafka is one of the widely used solutions for intra-service communication in microservices.

Microservices Performance Optimization

- **Keep memory footprint in limits**:
- Microservices footprint and business logic should be small and atomic.
- A microservice should solve a particular use case and not everything. This way the performance will increase.
- **A good microservice should not expose methods/functions that are not directly related (e.g. sales and procurement).**
- If we integrate with a microservice for a specific use case and we only use less than 30–40% of its features, then we are probably not calling a microservice.
- As such services are more like low-performance mini-monolithic services.

Microservices Performance Optimization

- **Use the right database Type/Technology:**
- Microservices response time directly depends on the data source and underlying database response time.
- Database selection and database modeling are very important.
- A microservice or microservices architecture can have RDBMS, Key-Value store, and/or unstructured data(eg. Images, videos), etc.
- To improve performance, structured data should be stored in RDBMS databases, and unstructured data should be stored in No SQL data store like Mongo DB, or Cassandra.

Microservices Performance Optimization

- **Database side Caching:**
- Database queries and responses should be cached for non-frequently changing data (or reference data) in the database.
- This will reduce hits to the database and saves it from getting overloaded. EHCache is a good open-source service that integrates well with Hibernate, Spring, and JPA.
- Also, we should spend some time identifying the good indexing/partition strategy in database tables.
- Prefer an 80/20 or 70/30 ratio for Database to Cache size i.e keeping 20–30% of more presently accessed data increases the performance.

Microservices Performance Optimization

- **Optimize Database Calls/Queries:**
- Avoid fetching the entire row (tuple) from the database.
- Suppose an API hits a database table and provides the 10 attributes, and the database table has 40 attributes (or more). If we execute a “Select All” or “Select * by id” query, it will return the whole row/tuple.
- A better approach is to pass the attribute names that are required for the API response. This will save the network cost and executes faster as well.

- **Database connection pooling:**
- DBCP is a way to reduce the cost of creating and closing connections by maintaining a connection pool.
- As making a new connection for each request is costly as well as time-consuming, DBCP allows to use of the same database connection for multiple requests.
- Also, spending time identifying the number of idle/stand-by connections is better. This enables good resource utilization.
- 11. Use Database clustering: Database clustering with load balancing allows the database to give a faster response to queries. There are multiple ways of handling this:
- There could be a Master-Slave configuration, where Slaves are read-only and eventually consistent, or

Microservices Performance Optimization

- **Use Database clustering:**
- Database clustering with load balancing allows the database to give a faster response to queries. There are multiple ways of handling this:
- There could be a Master-Slave configuration, where Slaves are read-only and eventually consistent, or
- Master-Master configuration which is a bit slow compared to master slave.

- **Database Tuning & Choosing a good Index and/or Partition Strategy**:
- Fine-tune the table space, disk space, and user space.
- Choosing the right indexes or partition strategy is very integral.
- A good index or partition strategy can optimize the query time and bad index selections can degrade the performance.
- If we are using JPA, it is a good practice to log and review the JPA-generated queries during the development phase, as sometimes these queries add unnecessary joins and self-joins.

Microservices Performance Optimization

- **Server-side caching:**
- Good caching leads to high-performance gains.
- And, a bad caching strategy degrades the performance.
- Caching the response of microservices depending on the request and its parameters.
- If the response is not frequently changing (responses like image, movie, or item details), then the response of the microservice can be cached based on input parameters.
- This will improve the performance as the business logic/compute needs not to be executed for similar/same requests.
- Memcached and Redis are good examples of in-memory caches that stores key-value between application and database.
- Redis is an in-memory, distributed, and advanced caching tool that allows backup and restore facilities as well.
- And, both integrate very well with Spring-based microservices. For Videos (clips, movies, etc), CDN is a good solution

Microservices Performance Optimization

- **Scaling:**
- Vertical Scaling (Scaling up) and Horizontal Scaling (scaling out) are two recommendations to handle the increased load on microservice.
- **Vertical scaling** : refers to increasing the memory of a single service. So, a vertical scaling requires a microservice restart (downtime) and it has hard dependency on the underlying storage availability.
- **Horizontal scaling:** refers to adding new node to service the requests. THis can be done in same host/cloud pool or can be done on different hosts/cloud pools. For same host/ cloud pool, the Auto-scaler is very common service which is available in most cloud providers. We can configure auto scaling based on Http Throughput, memory usage, etc.
- For Different hosts, a Load Balancer is required to route the traffic to microservices running on multiple nodes/cloud pools.
- A load balancer can be Pure Geographic or Round Robin or more customized.
-

Microservices Performance Optimization

- **API Gateways, Rate Limiters, and Proxies:**
- **Rate Limiters:**
- An **API Gateway** or in-house developed **API Rate Limiter** protects the APIs from over usage and increases the Availability of the microservices.
- Load Balancer also helps with Throttling or fixing the number of requests that hit a service at a certain point in time.
- Enabling Auto-scaling along with multiple node deployments, then adding a Load Balancer for distributing the requests.

ThankYou

shalini06mittal@gmail.com

7738460004