**Setup for Performance Testing**

1. **Simulate a Workload**: Create a workload that mimics real-world database and Redis interactions, such as multiple read/write operations.
2. **Measure Metrics**: Use a library like System.nanoTime() for latency measurement and maintain counters for throughput.
3. **Comparison**: Run the tests sequentially and in parallel. Compare the results using performance metrics like average latency and requests per second (throughput).

**Implementation**

**1. Performance Test Code**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.\*;

@Component

public class PerformanceTestRunner implements CommandLineRunner {

@Autowired

private UserService userService;

@Override

public void run(String... args) throws Exception {

int testSize = 100; // Number of test operations

List<User> testUsers = createTestUsers(testSize);

// Sequential Execution

System.out.println("Running Sequential Execution...");

long sequentialTime = executeSequentially(testUsers);

System.out.println("Sequential Execution Time: " + sequentialTime + "ms");

// Parallel Execution

System.out.println("Running Parallel Execution...");

long parallelTime = executeInParallel(testUsers);

System.out.println("Parallel Execution Time: " + parallelTime + "ms");

// Report Results

System.out.println("Performance Improvement: " + ((sequentialTime - parallelTime) \* 100.0 / sequentialTime) + "%");

}

private List<User> createTestUsers(int size) {

List<User> users = new ArrayList<>();

for (int i = 0; i < size; i++) {

User user = new User();

user.setName("User" + i);

user.setEmail("user" + i + "@example.com");

users.add(user);

}

return users;

}

private long executeSequentially(List<User> users) {

long startTime = System.currentTimeMillis();

for (User user : users) {

userService.createUser(user);

userService.getUserById(user.getId());

userService.deleteUser(user.getId());

}

return System.currentTimeMillis() - startTime;

}

private long executeInParallel(List<User> users) throws InterruptedException, ExecutionException {

ExecutorService executorService = Executors.newFixedThreadPool(10);

List<Callable<Void>> tasks = new ArrayList<>();

for (User user : users) {

tasks.add(() -> {

userService.createUser(user);

userService.getUserById(user.getId());

userService.deleteUser(user.getId());

return null;

});

}

long startTime = System.currentTimeMillis();

executorService.invokeAll(tasks);

executorService.shutdown();

executorService.awaitTermination(10, TimeUnit.SECONDS);

return System.currentTimeMillis() - startTime;

}

}

**2. Results Logging**

Run the performance test and observe logs similar to this:

Running Sequential Execution...

Sequential Execution Time: 1200ms

Running Parallel Execution...

Parallel Execution Time: 450ms

Performance Improvement: 62.5%

**Performance Report**

Create a performance report summarizing the results:

| **Metric** | **Sequential** | **Parallel** | **Improvement** |
| --- | --- | --- | --- |
| Execution Time (ms) | 1200 | 450 | 62.5% |
| Throughput (ops/sec) | 83 | 222 | 168% |
| Average Latency (ms/op) | 12 | 4.5 | 62.5% |

**Visualization**

To make the report more readable:

1. Export results to a CSV or JSON file.
2. Use tools like Python’s Matplotlib or Excel to plot graphs (e.g., bar charts for time comparison).

**Key Insights**

1. **Sequential Execution**: Processes operations one by one, resulting in higher latency and lower throughput.
2. **Parallel Execution**: Leverages multithreading to run tasks concurrently, reducing latency and increasing throughput significantly.

This approach demonstrates how parallel threading optimizes system performance.