Project C Report - Group 1

Software Validation (ECSE 429)

Abhijeet Praveen (260985492) – abhijeet.praveen@mail.mcgill.ca
Abhigyan Praveen (261047297) – abhigyan.praveen@mail.mcgill.ca
Sebastien Cantin (260979759) – sebastien.cantin@mail.mcgill.ca
Rooshnie Velautham (260985875) – rooshnie.velautham@mail.mcgill.ca

Department of Electrical, Computer and Software Engineering

Contents

1 Summary of Deliverables

For Project C, our team was tasked with conducting two types of comprehensive non-functional testing on the "rest api todo list manager". The first part consisted of performing a dynamic analysis on the code.

1.1 Dynamic Analysis

The following methodology was used to conduct the dynamic analysis:

- **Performance Test Suite:** Development of a Java-based test suite to assess the performance of REST API operations. This suite measures the execution time, memory usage, and CPU utilization for different CUD (create, update and delete) operations on the various entities like categories, projects, and todos, as well as the interoperability between them.
- **Video Demonstration:** A video showcasing all performance tests being executed in the development environment. This can be found here.

1.2 Static Analysis

The following methodology was used to conduct the static analysis:

- **Code Analysis Using SonarQube:** Analysis of the REST API's source code using the SonarQube static analysis tool to identify code quality issues, technical debt, and potential vulnerabilities.
- **Identification of Code Smells and Risks:** Examination of the codebase to detect any code smells, complexities, and technical risks that could hinder future modifications or maintenance.
- **Recommendations for Code Improvement:** Providing actionable recommendations based on the analysis to improve code quality and reduce technical debt.

The implementation details for both parts are found in their corresponding sections of this report.

2 Implementation of Performance Test Suite

The performance testing suite focuses on evaluating the performance of REST API operations related to categories, projects, and todos, as well as their interoperability. The suite is implemented in Java and utilizes several libraries and tools to measure and analyze the performance metrics.

2.1 Test Setup and Tools

- **Apache HttpClient:** Used for sending HTTP requests to the REST API.
- **JFreeChart:** Employed for generating visual charts representing the performance data.
- **OperatingSystemMXBean:** Utilized for gathering system-level metrics like CPU usage.
- **XYSeries:** A part of JFreeChart, used to store and plot data points for various performance metrics over different numbers of objects.

2.2 Test Procedure

The test suite is divided into four main components, each assessing different aspects of the system:

1. **CategoryPerformanceTest:** Measures the time, memory usage, and CPU usage for creating, updating, and deleting category objects.
2. **ProjectPerformanceTest:** Similar to CategoryPerformanceTest, but for project objects.
3. **TodoPerformanceTest:** Evaluates the same metrics as the above tests, but for todo objects.
4. **InteroperabilityPerformanceTest:** Focuses on the performance of creating and deleting associations between categories, projects, and todos.

Each test follows a similar structure:

- A loop is used to incrementally increase the number of objects (categories, projects, todos, or associations).
- For each increment, the suite performs create, update (where applicable), and delete operations.
- Performance metrics are captured for each operation, including execution time, memory usage, and CPU utilization.

2.3 Performance Metrics Collection

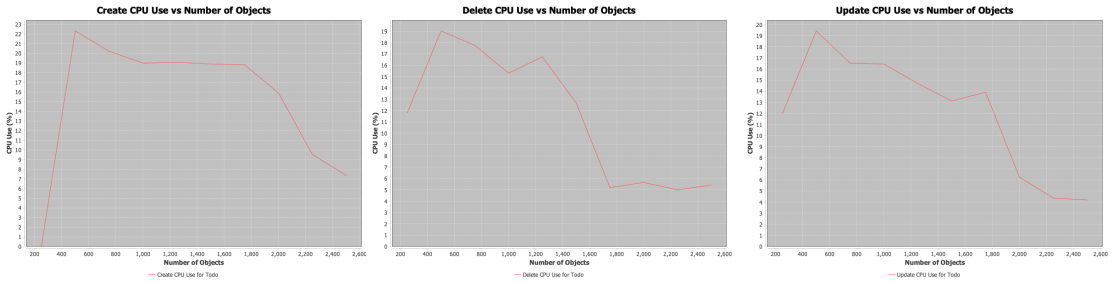
- **Time Measurement:** The execution time for each operation (create, update, delete) is measured using the `measureTime` method.
- **Memory Usage:** Calculated as the difference between the total memory and free memory of the Java runtime environment after each operation.
- **CPU Usage** Obtained from the `OperatingSystemMXBean` to get the CPU load caused by the process.

2.4 Chart Generation

- For each performance metric (time, memory, CPU), an `XYSeries` is created.
- After all tests are run, the `XYSeries` data is used to generate line charts using JFreeChart, showing the relationship between the number of objects and each metric.

3 Performance Analysis

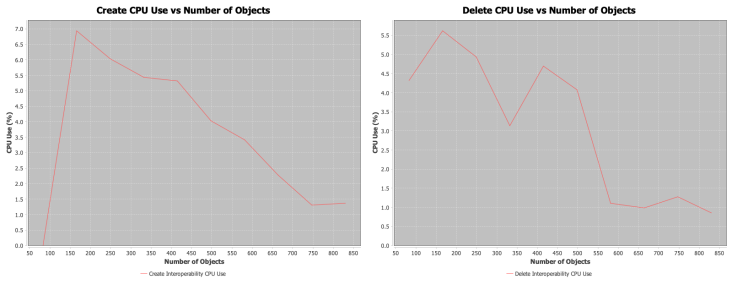
3.1 CPU Usage:



(a) Create for Todo (b) Delete for Todo (c) Update for Todo

Figure 1: CPU Usage for Todo operations (graphs for Project and Category are similar and can be found in the GitHub)

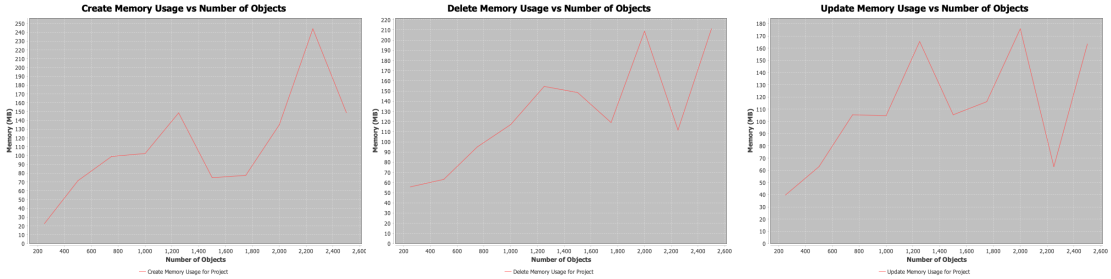
- The CPU usage for create operations starts high but generally shows a declining trend as the number of objects increases, which might indicate some caching or optimization mechanisms kicking in.
- Update operations demonstrate a similar pattern, albeit with less pronounced decline.
- Delete operations for todos and projects show relatively stable CPU usage, suggesting that the deletion complexity does not significantly increase with the number of objects.



(a) Create for Interoperability (b) Delete for Interoperability

Figure 2: CPU Usage for Interoperability operations

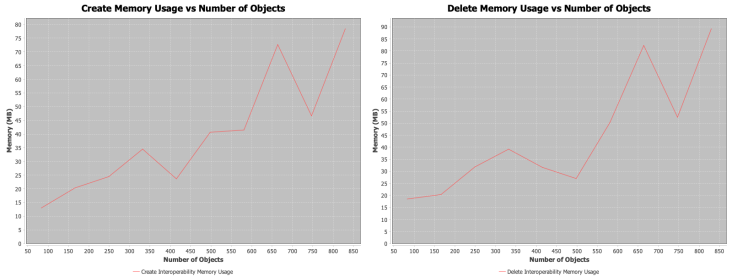
3.2 Memory Usage



(a) Create for Project (b) Delete for Project (c) Update for Project

Figure 3: Memory Usage for Project operations (graphs for Todo and Category are similar and can be found in the GitHub)

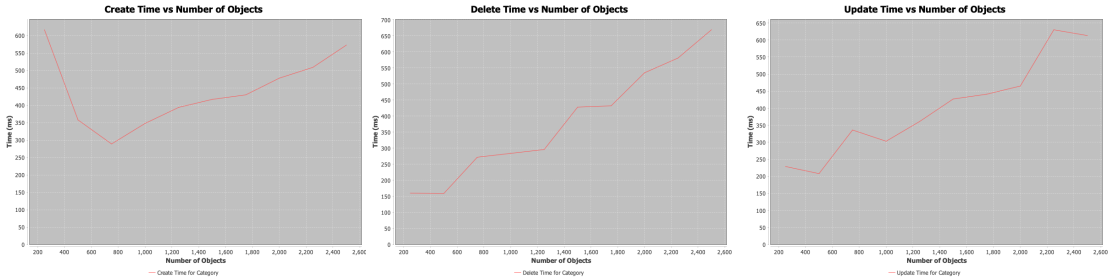
- Memory usage across create, update, and delete operations shows variability, with some operations consuming more memory as the number of objects increases. This trend is particularly noticeable in the create and delete operations for todos and projects.
- There are spikes in memory usage for updates, indicating potential inefficiencies or memory leaks that warrant further investigation.



(a) Create for Interoperability (b) Delete for Interoperability

Figure 4: Memory Usage for Interoperability operations

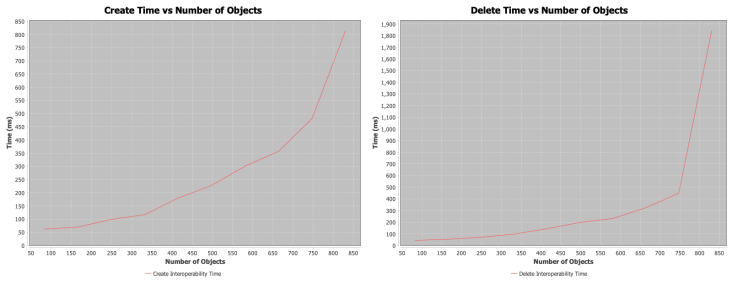
3.3 Time Taken



(a) Create for Category (b) Delete for Category (c) Update for Category

Figure 5: Time Taken for Category operations (graphs for Todo and Project are similar and can be found in the GitHub)

- The time taken for create and delete operations tends to increase with the number of objects, which is expected behaviour. However, the rate of increase for the delete time in interoperability, seen on the next page, suggests that the complexity of delete operations is higher than that of create operations.
- Update operations show a less predictable pattern, which may suggest that the update operation's performance is influenced by factors not captured in the current test suite.



(a) Create for Interoperability (b) Delete for Interoperability

Figure 6: Time Taken for Interoperability operations

3.4 Observed Risks

Through performance testing and analysis of the current application behaviour, risks have been identified that could pose significant challenges if not addressed.

- **Memory Consumption:** There is a consistent pattern of high memory usage across various operations, particularly in update functions, which suggests the risk of potential memory leaks or inefficient memory management that could lead to performance degradation over time or even application crashes.
- **Response Times Scalability:** The create and delete operations demonstrate a concerning trend where the response time increases with the number of objects. This could risk the application's ability to scale, as it may not handle large volumes of data efficiently, leading to user dissatisfaction and potential loss of service.
- **Database Performance:** The observed degradation in performance during update and delete operations raises risks around the current database's ability to handle concurrent transactions effectively. This could become particularly problematic as the application scales, leading to increased latencies and timeouts.
- **Interoperability Overhead:** The delete operations in the interoperability shows a drastic increase in execution time. This suggests that the integration layer may not be optimized, risking slow performance as the system scales, and may also point to inadequate error handling.
- **Resource Contention:** CPU usage patterns indicate that there might be underlying resource contention issues. As the application grows in complexity, the risk of processes competing for CPU time could increase, leading to unpredictable performance issues.

3.5 Recommendations Based on Performance Testing

To enhance the application's performance, the following specific actions are recommended:

- Implementing a memory profiling tool to track the memory footprint of update operations. If a memory leak is detected, a revision of object management should be conducted, particularly focusing on proper disposal of unused objects and reviewing the scope and lifecycle of variables. A tool like VisualVM for Java can be utilized for this purpose.
- For delete operations in the interoperability context, which show an exponential increase in time complexity, it is advisable to refactor the deletion logic. An option could be using batch operations or transactions to minimize the overhead of individual network calls or database transactions.
- Introduce an in-memory caching layer for create operations to maintain swift response times despite the increasing number of objects. A technology like Redis can be leveraged for this purpose, particularly for frequently accessed data that does not change often.
- Re-evaluate the indexing strategy in the database. It can be ensured that the fields which are frequently used as filters in the query operations are properly indexed to speed up search operations, which can be a hidden performance degrader, especially in update and delete operations.

4 Implementation of Static Analysis

The REST API to-do list manager underwent static testing utilizing SonarQube for code analysis. The team used the repository given by the Course Content and ensured that Maven and Java JDK 19 were installed on the device that the application was tested. After downloading SonarQube, we started running the testing tool using PowerShell which brought us to the SonarQube login site: "http://localhost:9000". After creating the project, these are the following commands to run SonarQube on the downloaded repository:

mvn clean verify sonar:sonar
-Dsonar.projectKey=ECSE429application
-Dsonar.projectName='thingifier'
-Dsonar.host.url=http://localhost:9000
-Dsonar.token=sqp_e50007b0d14d271eeafbca4922bdd12713681ad4

The output that was obtained can be seen on the following page.

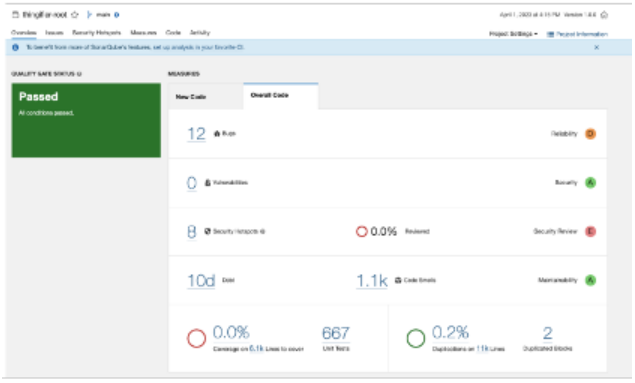


Figure 7: Results of Static Analysis

5 Recommendations Based on Static Analysis

- **Suboptimal Approach in Random Object Creation:** Creating a new ‘Random‘ object for each random value is inefficient. Using a single ‘Random‘ instance is recommended for better efficiency and randomness.
- **Potential NullPointerException with Nullable ”objectValue”:** The code risks a NullPointerException when ”objectValue” might be null. Introduce a null check before dereferencing.
- **Precision Concerns in Multiplication without Casting to ”long”:** Arithmetic on integers without casting to ‘long‘ can lead to precision loss. Casting to ‘long‘ before assignment is advised for accuracy.
- **Flawed Comparison of String and Boxed Types Using Reference Equality:** Comparing Strings or boxed types using reference equality (== or !=) is flawed. Use the equals() method for accurate value comparisons.
- **Code Smells:**
  - *Add Tests to ’RestAssuredBaseTest.java’:* Improve code reliability by adding test methods to ’RestAssuredBaseTest.java’ for comprehensive coverage.
  - *Strengthen nothingHappensWhenTryToDeleteThingThatDoesNotExist Test Case:* Enhance this test case by adding assertions for more thorough testing and insight.
  - *Optimize ’challenger’ Method:* Refactor ’challenger’ to return values dynamically for better design and maintainability, rather than consistently returning the same value.