

ECSE 429 – Software Validation Project

Part C – Non-Functional Testing of REST API

Authors

Ahmed Mossa – 261001739 – ahmed.mossa@mail.mcgill.ca

Lounes Azzoun – 261181741 – lounes.azzoun@mail.mcgill.ca

1. Summary of Deliverables

In the final stage of this project, we carried out both **performance testing** and **static code analysis**.

For performance testing, we measured how long it takes the system to create, update, and remove objects while progressively increasing the total number of entries. Throughout these experiments, we also monitored system resources such as **CPU** and **memory usage**.

For static analysis, the project's codebase was examined using the **SonarQube Community Edition**, with attention placed on identifying code smells, maintainability concerns, complexity issues, and potential future risks. We proposed several improvements aimed at strengthening the overall quality and reliability of the code.

Notable updates made to the test suite and utility tools include:

- **RandomObjectGenerator.java** – now uses JavaFaker to generate randomized test data for performance experiments.
- **PerformanceUtils.java** – expanded with helper methods for tracking CPU load and available memory via **OperatingSystemMXBean** and **ManagementFactory**.
- **CsvWriter.java** – added to export performance metrics into CSV files (`todo_performance_results.csv` and `project_performance_results.csv`).

The complete, updated test suite is available in the repository, along with a video demonstration that shows the tests executing in randomized order.

2. Implementation of Performance Test Suite

2.1 Todo

2.1.1 Summary of Results

Performance testing on the **todos** endpoint shows a clear and consistent increase in transaction time for all operations as the number of objects grows (see *Figure 1*).

Among all operations, **Create** exhibits the highest overall duration, followed by **Delete**, while both **Update (PUT)** and **Update (POST)** remain significantly lower and scale more predictably.

CPU usage patterns (see *Figure 2*) are more irregular. Both types of Update operations occasionally trigger noticeable spikes, particularly **Update (POST)** around mid-range object counts. Create and Delete also show occasional increased CPU activity, but these remain less frequent and generally lower in magnitude.

Memory usage (see *Figure 3*) demonstrates the most variability across operations. Create displays several large spikes at multiple object counts, reflecting the cost of generating and handling bulk data during the creation process. Update (PUT) and Update (POST) also exhibit memory jumps—especially Update (PUT) at 800 objects and Update (POST) at 1000 objects—while Delete shows moderate but consistent increases. These fluctuations are expected in bulk operations but indicate areas where memory handling could be optimized.

Overall, the results suggest that while the system scales functionally with load, Create and Delete operations become increasingly expensive at higher volumes, and Update operations occasionally trigger localized CPU and memory spikes.

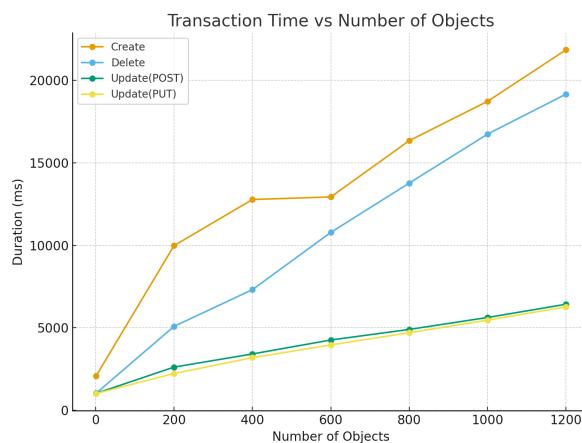


Figure 1. Transaction time versus number of manipulated “todo” objects.

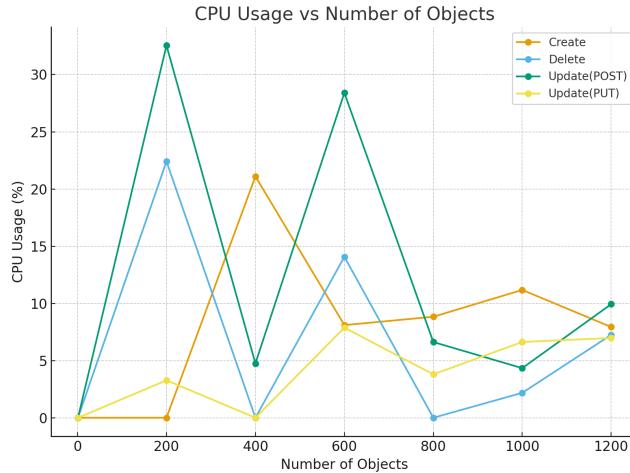


Figure 2. CPU usage versus number of manipulated “todo” objects.

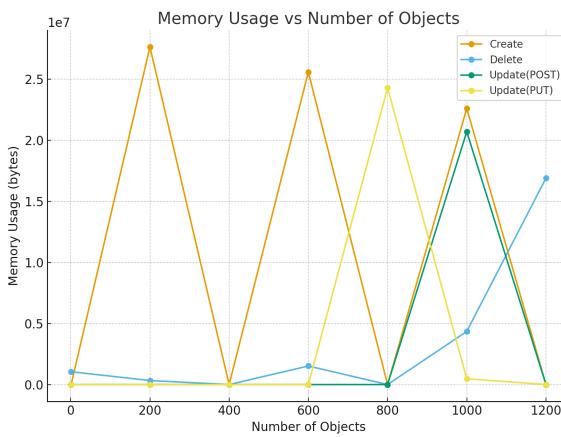


Figure 3. Memory usage versus number of manipulated “todo” objects.

2.1.2 Recommendations for Code Enhancements

Based on the observed behavior, several enhancements can improve performance and resource stability:

- **Optimize Create and Delete logic.** These operations show the largest growth in duration. Reviewing object construction, JSON serialization/deserialization, and database write paths can help reduce overhead. Batch inserts/deletes or minimizing redundant validation steps would reduce latency at scale.
- **Improve memory management during Create operations.** The large memory spikes indicate that temporary allocations or large in-memory structures are being used. Streaming approaches, object reuse, or more efficient data generation techniques may mitigate this.

- **Analyze Update(PUT/POST) pathways for CPU spikes.** Although overall update performance is stable, localized CPU peaks suggest portions of the update logic may involve expensive computations or non-optimized loops.
 - **Ensure proper indexing and efficient database queries.** As object counts grow, inefficient queries or missing indexes can amplify latency, particularly in Create and Delete operations that interact heavily with storage.
 - **Consider asynchronous or incremental processing.** Offloading parts of Create/Delete operations to background tasks (e.g., logging, cleanup) can help reduce the synchronous response time experienced during the test suite.
-

2.1.3 Performance Risks

The primary risks identified for the todos endpoint include:

- **High transaction times for Create and Delete.** These operations scale steeply with input size. In real-world deployments, large batches can lead to slow API responses, timeouts, or backpressure on the system.
 - **Memory spikes during Create and Update operations.** These fluctuations could lead to instability under heavy load, particularly on resource-constrained environments or when many concurrent create/update operations occur.
 - **Irregular CPU spikes on Update operations.** While not constant, these spikes indicate potential inefficiencies that may worsen under concurrent workloads, affecting throughput and responsiveness.
 - **Overall scaling concerns at high object counts.** Although the system scales, the performance degradation observed in Create and Delete operations suggests that optimizations will be necessary to maintain responsiveness in large datasets or under production-level traffic.
-

2.2 Project

2.2.1 Summary of Results

Performance testing of the **project** endpoint reveals a consistent increase in transaction time across all operations as the number of objects increases (see *Figure 4*).

Create, Amend(POST), Update(PUT), and Delete all scale in roughly the same pattern, with Delete showing the steepest growth at higher object counts due to the overhead of bulk project removal.

CPU usage patterns (see *Figure 5*) fluctuate across operations, with occasional sharp spikes—most notably in Update (PUT) at 200 objects — and moderate increases during Create and Amend (POST) at higher loads. Delete operations exhibit relatively low CPU usage until very high object counts are reached, suggesting that most of the cost associated with Delete is I/O or database-bound rather than CPU-bound.

Memory usage (see *Figure 6*) reveals the most variability across the Project endpoint. Create and Update (PUT) operations show significant memory spikes at 600, 1000, and 1200 object counts, consistent with the increased cost of building and modifying large batches of project objects. Delete also exhibits large memory spikes at higher object counts, reflecting the cost of retrieving and cleaning up many project entities. Amend(POST), however, remains mostly stable with only minor increases.

Overall, the Project endpoint scales functionally but shows sensitivity to memory usage during Create, Update, and Delete operations at higher object counts. These spikes indicate that bulk operations may be allocating large intermediate structures or performing memory-intensive transformations.

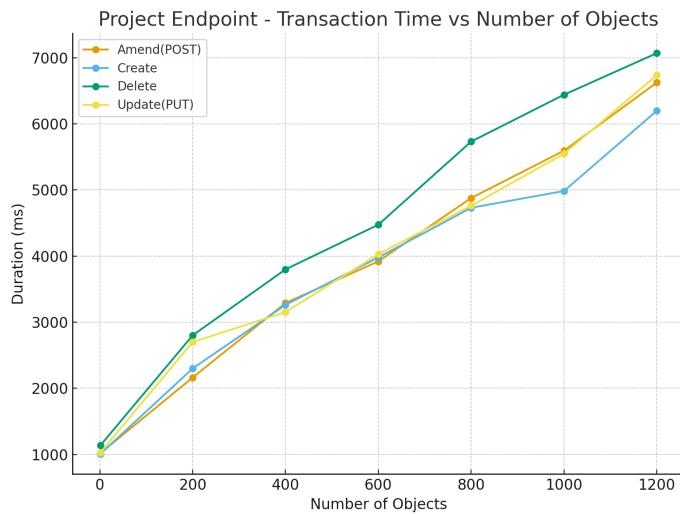


Figure 4. Transaction time versus number of manipulated “project” objects.

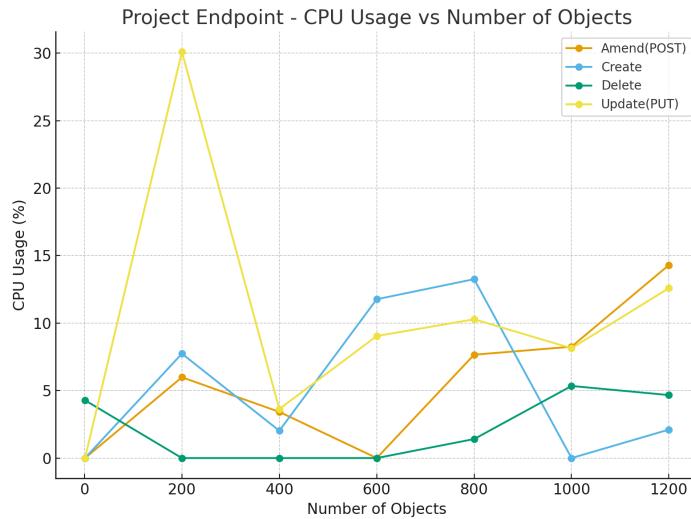


Figure 5. CPU usage versus number of manipulated “project” objects.

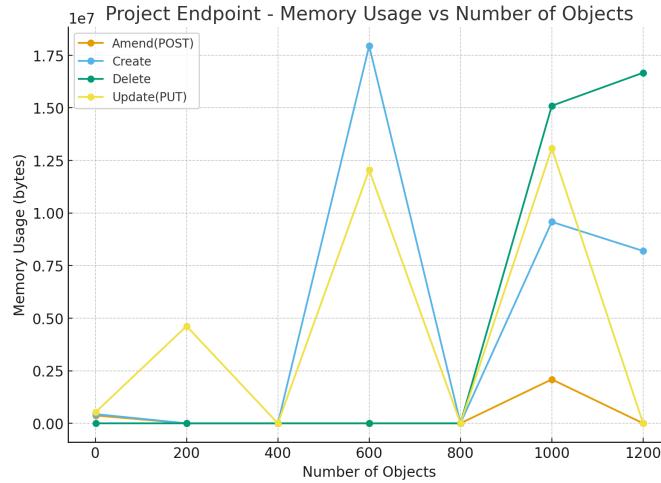


Figure 6. Memory usage versus number of manipulated “project” objects.

2.2.2 Recommendations for Code Enhancements

To improve the performance and scalability of the Project endpoint, the following enhancements are recommended:

- **Optimize memory handling in bulk operations.**

The memory spikes observed during Create, Update, and Delete operations suggest the presence of large in-memory structures, repeated allocations, or inefficient data copying. Investigating opportunities for streaming, chunking, or reusing objects can significantly reduce peak memory load.

- **Review and optimize Update(PUT) workflow.**
The CPU spikes at specific object counts indicate that certain parts of the update logic may involve heavy computations or inefficient loops. Profiling these code paths may help locate redundant computations or opportunities for algorithmic improvement.
 - **Improve Delete operation efficiency.**
Delete scales the steepest in transaction time and memory usage, possibly due to cascading deletions or inefficient batch handling. Database-level optimizations, such as indexed lookups, batched deletion, or deferred cleanup, can reduce overall cost.
 - **Evaluate asynchronous or background processing.**
Offloading expensive parts of Create/Delete operations (e.g., logging, cleanup, relationship updates) to asynchronous jobs would help reduce synchronous response times during high-volume operations.
 - **Ensure database queries are optimized.**
Slowdowns at large object counts may be due to non-indexed fields or N+1 query patterns. Ensuring efficient indexing strategies and verifying that ORM/database logic avoids redundant queries will improve overall throughput.
-

2.2.3 Performance Risks

The following risks were identified for the project endpoint:

- **High memory usage under bulk operations.**
Memory spikes during Create, Update(PUT), and Delete pose a risk of instability or out-of-memory behavior under production-level traffic, especially when many large operations occur concurrently.
- **Localized CPU spikes.**
Although generally stable, Update (PUT) demonstrates significant CPU spikes at certain object counts, which can reduce throughput and increase latency under unpredictable workloads.
- **Steady growth in transaction times.**
As object volume increases, all operations exhibit predictable but significant growth in execution time. In high-traffic environments, this could result in slow responses, timeouts, or reduced system responsiveness.
- **Scalability limits in bulk workflows.**
The combination of memory spikes and rising transaction times suggests that the Project

endpoint may hit scalability limits when handling very large batches, making optimization essential for long-term maintainability.

3. Implementation of Static Analysis with SonarQube

To conduct static analysis, we set up the **SonarQube Community Edition** locally by downloading and extracting the package, then launching the server. After accessing the web interface, the project was scanned using the **Maven Sonar Scanner**, with a **sonar-project.properties** file specifying the project key.

A summary of complexity metrics, issue categorizations, and improvement recommendations is listed below. Duplicate findings were removed for clarity. Issues are grouped by **severity** (high, medium, low) and by **type** (bug or code smell).

Here's the data outputted by SonarQube:

Complexity

Cyclomatic Complexity: 1,900

Cognitive Complexity: 1,442

Size

New Lines: 0

Lines of Code: 9,412

Lines: 13,377

Statements: 4,127

Functions: 777

Classes: 136

Files: 139

Comment Lines: 888

Comments (%): 9.5%

Issues

Severity: HIGH

Type: Code smells

- **Refactor this method to reduce its Cognitive Complexity:** the method is too complex and difficult to follow; simplify it by splitting it into smaller, focused functions.
- **Define a constant instead of duplicating literals:** avoid repeating the same hardcoded values by introducing constants.

- **Make the method "static" or remove unnecessary field usage:** convert the method to static or eliminate unneeded references to instance fields when they are not required.

Severity: MEDIUM

Type: Bug

- **The return value of “format” must be used:** the result of calling `format` should be captured and applied; ignoring it may cause unexpected behavior.

Severity: MEDIUM

Type: Code smells

- **Remove this block of commented-out lines of code:** old commented code should be deleted to keep the file clean.
- **Remove this unused method parameter:** unused parameters should be removed to avoid confusion.
- **Remove this unused private method:** delete private methods that are not invoked anywhere.
- **Remove this useless assignment:** eliminate redundant assignments to variables that have no effect on program logic.
- **Define and throw a dedicated exception instead of using a generic one:** replace broad exceptions with specific custom types to enhance clarity and debugging.
- **Remove this use of Thread.sleep():** avoid using `Thread.sleep()` as it is generally poor practice in both production and test environments.
- **Remove unused imports/fields:** delete unused imports or class fields to minimize clutter.
- **Reduce the number of assertions in a single test:** break large tests into smaller ones to improve clarity and maintainability.

Severity: LOW

Type: Code smells

- **Complete TODO comments:** outstanding TODO items should be addressed and resolved.
- **Static final fields:** several fields should be converted to `static final` to make them immutable and clearer in intention.
- **Remove public modifiers in JUnit tests:** public modifiers are unnecessary for JUnit test methods and should be omitted.
- **Replace if-then-else statements with single return statements:** simplifying conditionals makes the code easier to read.
- **Refactor long or complex methods:** lengthy or intricate methods should be reorganized for better maintainability.
- **Refactoring redundant variables:** remove temporary variables that simply store values before returning them; return expressions directly.
- **Refactor "Brain Methods":** overly complex and deeply nested methods should be broken down into smaller, more manageable parts.
- **Avoid redundant fields in classes:** eliminate fields that are better represented as local variables.

- **Standardize method and variable naming:** rename inconsistent identifiers to follow common naming conventions.
- **Refactor for validation:** add necessary validation checks (such as null checks) to prevent potential runtime issues.