ECSE 429 – Part C

Non-Functional Testing of Rest API

Naomie Lo | naomie.lo@mail.mcgill.ca | 261018690

Deniz Emre | deniz.emre@mail.mcgill.ca | 261029931

Performance Testing

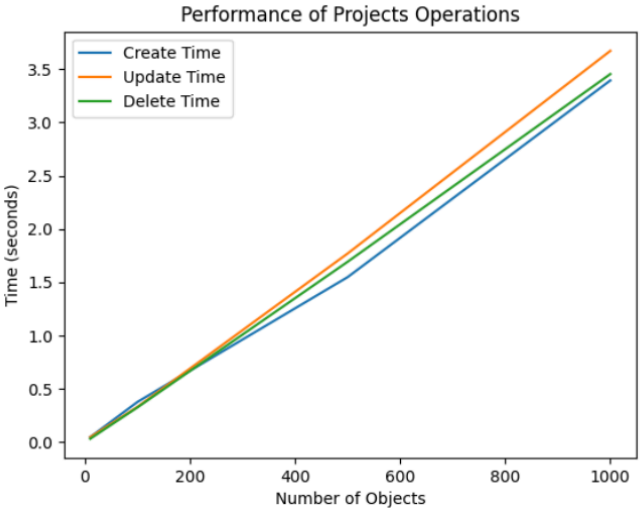
Implementation of performance test suite

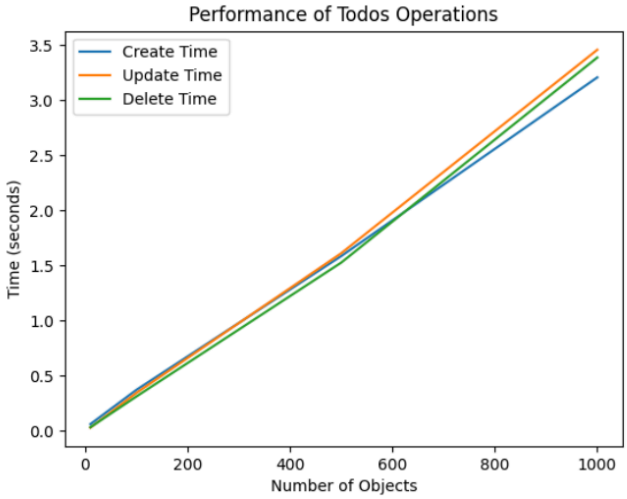
The performance test suite utilizes Python’s psutil library and the system's perfmon (Performance Monitor) to analyze the resource consumption during API operations. psutil helps in programmatically tracking CPU and memory usage, providing detailed metrics for dynamic analysis. The script generates thingifier objects, with operations creating, deleting, and updating. This simulates real-world application scenarios under varying loads.

Using psutil, CPU usage and memory allocation are monitored dynamically as the tests are executed. These metrics are logged to identify trends, such as increased resource usage with the growing number of objects or the duration of operations.

The perfmon tool complements this by providing a system-level view of performance, including metrics like available free memory, network usage, and overall system health. This ensures accurate benchmarking and validation of results.

Time to objects





Observations

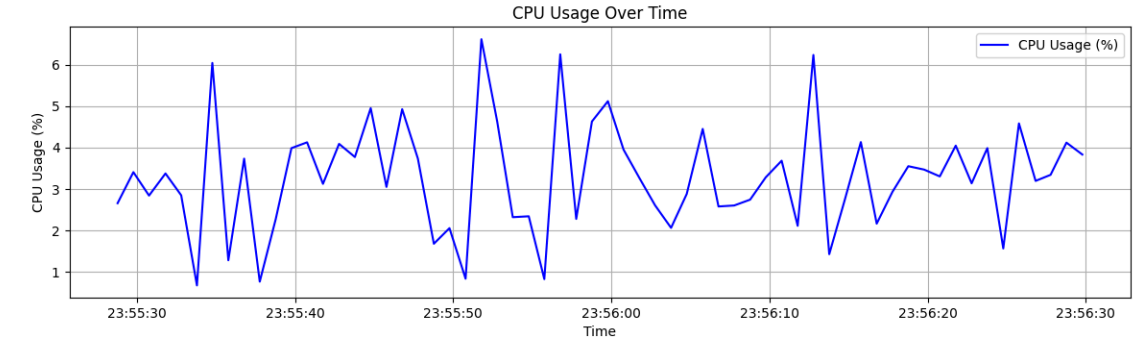
The plots indicate that the time taken for create, update, and delete operations scales linearly with the number of objects, demonstrating that the system maintains consistent performance as the workload increases. However, the relative timing of these operations—whether creation, updating, or deletion takes the longest—varies between test runs. This variability could be influenced by factors such as system resource availability, background processes, or slight differences in how operations are executed during each run. For instance, in one run, create might take the most time due to database insertions or validations, while in another run, delete may take longer, potentially influenced by cleanup tasks or related dependencies. Similarly, update operations may sometimes be the slowest due to the complexity of modifying existing data. This lack of a fixed ranking highlights the dynamic nature of resource allocation and system behavior during tests.

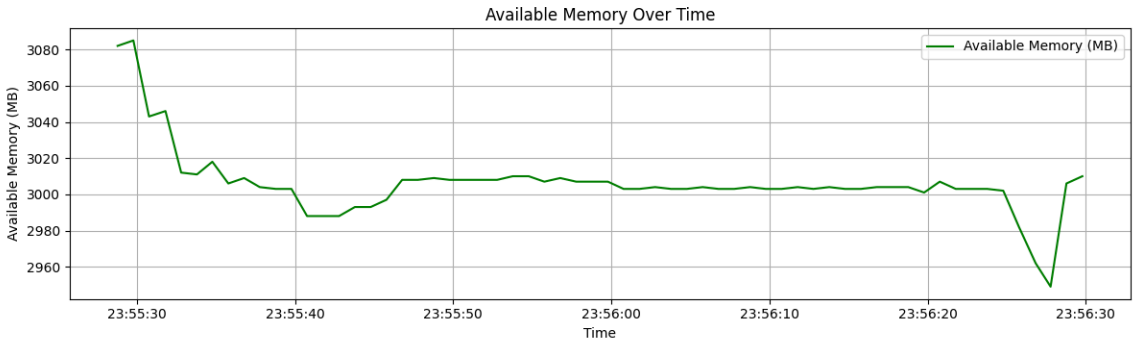
The key takeaway is that, despite the fluctuations in the order of operation times, the linear increase in execution time with the number of objects suggests predictable scalability. Further profiling could explore why certain operations occasionally dominate and identify areas for optimization under specific conditions.

Recommendations

The linear scalability observed in the time taken for create, update, and delete operations is a positive sign of the system's performance. To further optimize, profiling should be conducted to pinpoint the underlying causes of variability between operations, such as database locks or background processes. Reducing inconsistencies in operation timing will enhance predictability and efficiency. Additionally, consider implementing caching or batch processing for frequently performed operations to minimize system overhead during high workloads.

Time to CPU Usage and Memory Availability





Observations

The plot illustrates the system's resource utilization during the performance tests. The CPU usage fluctuates significantly, ranging from around 1% to a peak of approximately 6%. These variations indicate that different phases of the tests place varying demands on the CPU. For instance, operations like creating, updating, or deleting objects may have different computational requirements, leading to peaks during more intensive tasks and dips during idle periods or less demanding operations. The available memory graph shows an initial decline, likely due to the system allocating resources for data handling or caching during the tests. This is followed by a plateau, indicating that memory usage stabilizes as the workload becomes consistent. However, the sharp drop near the end suggests a memory-intensive operation, such as bulk processing or cleanup tasks, with a recovery indicating the release of memory, possibly due to garbage collection or the completion of the tests.

Overall, the resource utilization reflects a dynamic yet stable system response, with no signs of extreme spikes or crashes. The consistent

behavior after the initial adjustment phase suggests the system handles the workload efficiently. However, the initial memory drop and sharp fluctuations at specific points highlight opportunities for optimization, particularly in memory management. This analysis underscores the importance of monitoring resource usage during performance testing to identify areas for improvement and ensure the application can handle varying workloads effectively.

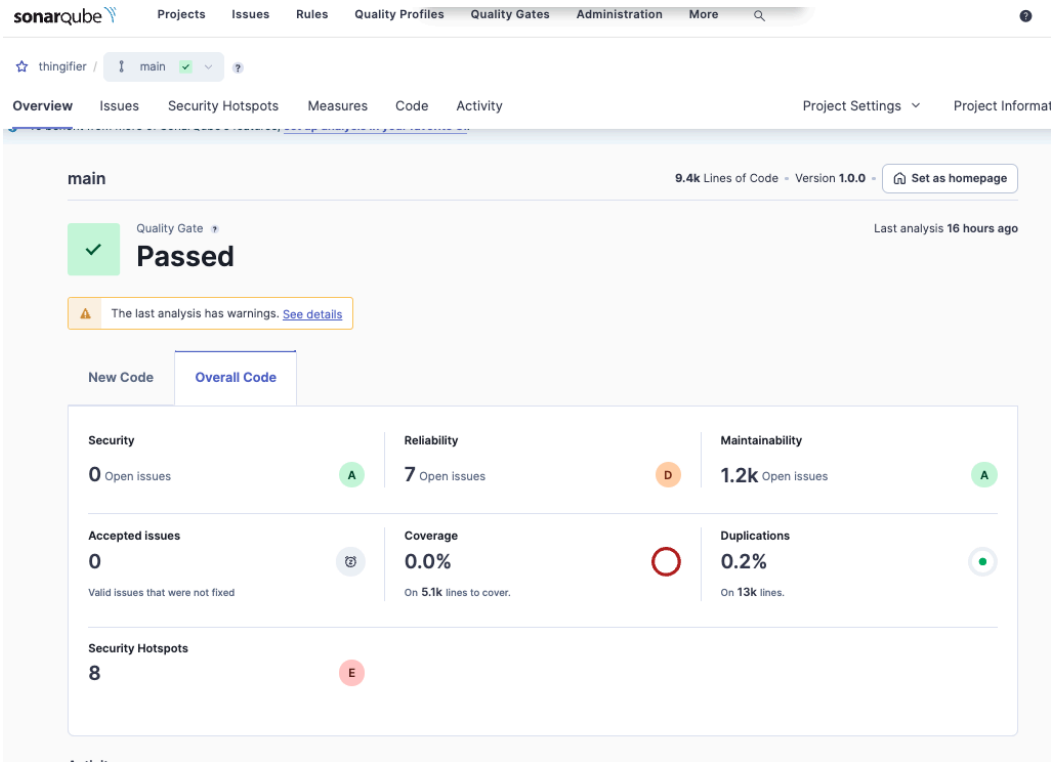
Recommendations

The system demonstrates stable resource utilization with no extreme CPU spikes or crashes, which is promising. However, the initial memory drop and sharp fluctuations near the end suggest potential inefficiencies in memory management. To address this, investigate memory allocation strategies during peak tasks and optimize garbage collection or resource release mechanisms. Ensuring smoother transitions during memory-intensive operations will enhance system reliability and prevent potential bottlenecks under heavier workloads.

Static Code

Implementation

We set up SonarQube locally on a Macbook Air M1 using Homebrew, which manages the installation and service lifecycle. The setup runs on Java 17, provided by Homebrew's OpenJDK package, and is configured as a background service accessible via http://localhost:9000. By default, it uses an embedded H2 database for local use. For code analysis, we installed the Sonar Scanner CLI tool, which runs from its installation directory and uploads analysis results to the SonarQube server for further quality assessment and reporting. We created a Maven project within the SonarQube interface by navigating to Create Project > Local Project, assigning “thingifier” as the project name, and configuring the Maven build. After the analysis ran, we saw the project and visualize its metrics on the sonarqube interface.



Code Complexity

Cyclomatic Complexity

Examples

Method	Branch Points
ThingifierHttpApi Constructor	Two if conditions (apiRequestHooks == null, apiResponseHooks == null)
requestWrapper Method	Five branching points (httpResponse, apiResponse, verb, etc.)
validateContentTypeHeader	Two branching points (willApiEnforceContentTypeHeaderForRequests, `accept.isMissing()

Recommendations

1. RequestWrapper is the most complex as outlined above. We would refactor this into:

```
private boolean isContentValidationRequired(HttpVerb verb) {
   return verb == HttpVerb.POST || verb == HttpVerb.PUT || verb == HttpVerb.PATCH;
}
if (httpResponse == null) {
   apiResponse = validateAcceptHeader(acceptHeader);
   if (apiResponse == null && isContentValidationRequired(verb)) {
      apiResponse = validateContentTypeHeader(request.getHeader("Content-Type", ""));
   }
}
```

2. Combine similar logic like:

```
this.apiRequestHooks = apiRequestHooks == null ? new ArrayList<>() : apiRequestHooks;
this.apiResponseHooks = apiResponseHooks == null ? new ArrayList<>() : apiResponseHooks;
```

Cognitive Complexity

Examples

Method	Reason
ThingifierHttpApi Constructor	Sequential decisions without nesting.
requestWrapper Method	Deeply nested conditions and multiple branches.
validateContentTypeHeader	Simple conditions without significant nesting.

Recommendations

1. Move nested logic into a helper method to reduce depth and improve readability

```
private boolean isContentValidationRequired(HttpVerb verb) {
   return verb == HttpVerb.POST || verb == HttpVerb.PUT || verb == HttpVerb.PATCH;
}
```

2. Replace deeply nested logic with guard clauses

```
if (httpResponse == null) {
   apiResponse = validateAcceptHeader(acceptHeader);
   if (apiResponse == null && isContentValidationRequired(verb)) {
      apiResponse = validateContentTypeHeader(request.getHeader("Content-Type", ""));
   }
}
```

3. Combine similar logic to reduce repeated 'if'

```
this.apiRequestHooks = apiRequestHooks == null ? new ArrayList<>() : apiRequestHooks;
this.apiResponseHooks = apiResponseHooks == null ? new ArrayList<>() : apiResponseHooks;
```

Code statement counts

Lines of code	Lines	Statement	Functions
9428	13390	4172	779

Classes	Files	Comment lines	Comments %
9428	13390	4172	779

Thingifier

The thingifier/src folder has 2335 statements. We noticed long methods, large classes, repetitive code, and as mentioned before unnecessary complexity.

Recommendations

1. Refactoring to improve readability and maintainability. This will also make it easier to unit test the individual components.

```
public ValidationReport validate(final BodyParser bodyargs, final Thing thing) {
   ValidationReport report = initializeReport(bodyargs, thing);
   boolean relationshipsValid = validateRelationships(bodyargs, thing, report);
```

```
    report.setValid(relationshipsValid);
    return report;
}
private ValidationReport initializeReport(final BodyParser bodyargs, final Thing thing) {
    ValidationReport report = new ValidationReport();
    // Initialization logic here
    return report;
}
private boolean validateRelationships(final BodyParser bodyargs, final Thing thing, ValidationReport report)
{
    // Logic for validating relationships
    return true; // or false based on logic
}
```

2. Methods like validateCompressedRelationshipDefinition have a lot of nesting.

```
if (parts.length != 4) {
    reportIsNotValidRelationship(complexKey, report);
    return false;
}
String relationshipPart = parts[0];
```

3. For thingifier/src/…/BodyRelationshipValidator.java, some logic can be updated with a helper for actions like splitting strings, checking parts length and validation.

```
if (parts.length != 4) {
    reportIsNotValidRelationship(complexKey, report);
    return false;
}
String relationshipPart = parts[0];
```

Technical Risk

Weak Cryptography

This is a medium level priority in the security hotspots tab. The main feedback for this is making the pseudorandom number generator safe. Currently the code is using "ThreadLocalRandom.current()". An alternative to this would be:

```
SecureRandom secureRandom = new SecureRandom();
int randomValue = secureRandom.nextInt();
```

Insecure Configuration

This is a low level priority in the security hotspots tab. There is a debugging feature that was left in the code despite it being in production, "printStackTrace()". This is not a big issue as the user will not likely see this. These print statements are in files related to storage, specifically, challenger/src/main/java/uk/co/compendiumdev/challenge/persistence/ChallengerFileStorage.java and challenger/src/main/java/uk/co/compendiumdev/challenge/persistence/AwsS3Storage.java. Two files that the user does not directly interact with.

Technical Debt

The current debt is 8d 6h. The thingifier/src path has the highest debt of 4d 3h. And the api/http/bodyparser file has almost 25% of this debt.

Recommendations

1. Type checking and casting is an issue here. Instead of using instanceof checks with a factory method, this can be done dynamically with a map. This will make is easier to add new types in the furture.

```
interface TypeHandler {
    String convert(Object value);
}

Map<Class<?>, TypeHandler> typeHandlers = Map.of(
    Boolean.class, value -> String.valueOf(value),
    String.class, value -> (String) value,
    Double.class, value -> String.valueOf(value)
);

private String convertToString(Object value) {
    TypeHandler handler = typeHandlers.get(value.getClass());
    if (handler != null) {
        return handler.convert(value);
    }
    throw new IllegalArgumentException("Unsupported type: " + value.getClass());
}


..
stringsInMap.put(key, convertToString(theValue));
```

2. There are many complex methods that are lengthy and have more than one action is is trying to perform. Best practice is for functions to have one purpose. An example is the flattenToStringMap function, please see below.

```
private void handleMapType(Object value, String prefixKey, List<Map.Entry<String, String>> result) {
    Map<?, ?> map = (Map<?, ?>) value;
    for (Map.Entry<?, ?> entry : map.entrySet()) {
        String nestedKey = prefixKey + "." + entry.getKey();
        result.addAll(flattenToStringMap(nestedKey, entry.getValue()));
    }
}

private List<Map.Entry<String, String>> flattenToStringMap(String prefixKey, Object value) {
    List<Map.Entry<String, String>> result = new ArrayList<>();
    if (value instanceof Map) {
        handleMapType(value, prefixKey, result);
    } else if (value instanceof List) {
        handleListType(value, prefixKey, result);
    } else {
        result.add(new AbstractMap.SimpleEntry<>(prefixKey, convertToString(value)));
    }
    return result;
}
```

Code Smells

There are many of these throughout the project, the table below outlines some common issues from the 1200 flagged issues in maintainability.

Issue	Recommendation
Use isEmpty() to check collections	Replace manual size checks with isEmpty() for readability and performance.
Replace System.out with a Logger	Use a logging framework like SLF4J or Log4j.
Remove commented-out code	Delete unnecessary commented-out code; rely on version control for history.
Rename fields to match conventions	Use consistent naming conventions (e.g., camelCase).

Use dedicated exceptions	Replace generic exceptions with specific or custom exceptions.
Refactor high cognitive complexity methods	Break into smaller methods and reduce nesting with guard clauses.
Define constants for duplicate strings	Use constants to replace repeated literals in the code.