ECSE 429 F2020 - Group 14 - Final Report

Arndtsen, Ryan
260867670
dept. of ECSE
McGill University
Montreal, Canada
ryan.arndtsen
@mail.mcgill.ca

Dufault, Louca
260870863
dept. of ECSE
McGill University
Montreal, Canada
louca.dufault
@mail.mcgill.ca

Lim, Caleb
260869612
dept. of ECSE
McGill University
Montreal, Canada
caleb.lim
@mail.mcgill.ca

Mihaylov, Volen
260746982
dept. of ECSE
McGill University
Montreal, Canada
volen.mihaylov
@mail.mcgill.ca

I. Introduction

In this project, we were tasked with analyzing a given RestAPI to uncover bugs and evaluate the overall reliability and performance during use, with the aim of formalizing a set of recommendations concerning the major vulnerabilities. The RestAPI selected was the Todo Manager RestAPI, a rudimentary API developed by the thingifier group and provided as a runnable Java Archive file.

This API comprises three main entities; todo, project, and category; and exposes several endpoints for interacting with these entities in a standard fashion. The API provides the facilities to satisfy CRUD operations on entity persistence, using the id field for all entities as a primary key for indexing and retrieving data. Entities can also be related to one other as prescribed by the documentation, which requires several endpoints on the API for the sake of interacting with the relationships themselves.

The API is well documented with a comprehensive description of the paths and types of all data the application may expect of request and that may be returned in responses. However, it is worth noting that the documentation is not always accurate as will be discussed later.

In order to fully exercise the application under test, our semester-long testing project was divided into three distinct iterations. Our team collaborated to implement the test classes and method in support of the test approaches specified for each iteration. For the first iteration, we implemented an exhaustive unit test suite covering all facets of the RestAPI under test. During the second iteration, we first had to design the user stories that we would later glue to corresponding step definitions in order to implement was Story Testing. Lastly, in the third iteration we performed Non-Functional testing of the provided RestAPI with Static analysis of the source code and Dynamic analysis of the application while running in the form of Performance tests.

II. Test Utilities

We designed and made use of several files to house functionality that was common to multiple test classes across the packages, namely the Resources, Server, and Utils classes.

The Resources class provided all the constants relevant to the project, e.g. HTTP codes and URLs, as statically instantiated variables made accessible to any and all test classes. This class was created after the duplication of literals across several test classes was noticed, and as such we adopted best practices to cut down on code duplication and potential debt that would accumulate should any of these literals need to be updates.

The *Server* class encapsulated the life-cycle management of the given RestAPI server. In order to start the server, we used a Process Builder to execute the Java Archive file in another thread with the command *java -jar runTodoManagerRestAPI-1.5.5.jar*. Once built, the process builder returns an instance to act as a handle on the running process, that we statically store in the *Server* class. Many efforts were made to ensure that the server was indeed running, including sleeping the main testing thread in order to give the server enough time to start up, and polling the RestAPI's default endpoint at a fixed rate until

a successful response. Stopping the server could be achieved either by requesting the shutdown endpoint as indicated in the documentation, or by destroying the process instance. Logging was added early on to this *Server* class to ensure that the functioning of this critical part of our testing behaved according to expectations. As a whole this utility class served as a valuable abstraction to ensure that the running RestAPI server process could be managed by the tests (usually during the setup and tear-down phases) in a mere matter of lines, avoiding vast amounts of duplicated code that would be very much unrelated to the artifact under test.

The *Utils* class comprised of numerous methods that provided commonly-used functionality for interacting with the server, such as creating objects, counting the number of objects, finding an object by name, etc. which were implemented for all three object types.

III. Part A - Exploratory and Unit Testing

The first part of the project included performing Exploratory tests and Unit tests on the ToDo List Manager RestAPI. These tests were done separately and served separate purposes in the quality assurance process.

Firstly, exploratory testing is an unstructured type of testing which is used to discover bugs at a high-level. The approach loosely follows some of the goals and ideas established ahead of time, and involves a tester or testers going through the software like they are a user of the system and discovering what isn't working as it is supposed to. The testers record in sufficient detail the noteworthy actions made in performing the test, making special note of any correct or unexpected behaviour that is uncovered. Exploratory testing is most commonly used in the Agile development process and serves as a simple way to discover major, more obvious bugs.

Unit testing is a more in-depth testing approach, which tests each component of the system individually instead of integrated with each other. Unit test are performed on the smallest self-contained sub-components in isolation to insure that the overall components are fully exercised from the bottom up. Unit tests comprise boundary tests to see if the components are functioning correctly. In the quality assurance process, unit tests are usually one of the first approaches taken, since components must function individually before they are able to be integrated together.

A. Implementation

1) Exploratory Testing: The implementation of exploratory testing began by creating a template for the charter document that would be created. This document included:

- Charter Description: Identifying the capabilities and areas of potential instability of the ToDo List Manager RestAPI.
- Build: The release version, and how the RestAPI was run.
- Area: The area that would be tested (main scenarios in this case).
- Environment: Operating system that the test was run on.
- Start: Start time and date of charter
- Other data: Other relevant documents to help with the charter.
- Tester: The name of the tester(s) that worked on the charter.
- Task Breakdown: Time that each task in the charter took.
- Charter Notes: All timestamped notes from the charter.
- Areas Not Covered: Areas not covered within charter.
- Bugs Logged/ Closed: Bugs found and closed within the charter.
- Issues/ Other Session Ideas: Ideas that were found during the session.
- Areas of Potential Instability: Potential Bugs.

These charter documents were written in TextEdit on MacOS and NotePad on Windows.

The next step of the exploratory testing implementation was to assign groups. Two testers were assigned to each group and would execute a single 45 minute charter each. To execute the charters, one team member would be in charge of going through the RestAPI and testing all queries while the other team member would write the notes to record the process in the charter document.

To go through the RestAPI, testers used Postman, an open source tool which connects to any exposed API endpoint and provides a simple and powerful interface for running requests against that endpoint and any of its paths. This tool proved very useful in allowing testers to

maximize their time actually testing the RestAPI instead of worrying about the logistics of running queries.

Finally, the charters were brought together and compared to find any trends and differences, and decipher if there was any useful information produced from the findings.

2) Unit Testing: The Unit Test Suite was setup as a Java Maven project relying on the JUnit framework for managing the test execution, with IntelliJ IDE used as the editor. JUnit provided a clean and useful API to hook into the Java code we wrote with minimal clutter, and allowed us to carry out the design principles we had planned for the various testing projects without imposing too many restrictive conventions. The tests were separated into different files within the unit test package based on the object type that was being tested. The file tree is as follows:

```
../group14
├── BaseTest.java
├── CategoriesTest.java
├── DocsTest.java
├── ProjectsTest.java
├── Resources.java
├── Server.java
├── TestRunner.java
├── TestUtils.java
└── TodosTest.java
```

Firstly, *BaseTest* is an abstract class which contained common methods and attributes that most test would need to use. Therefore, each class inherited from this class (with the *extends* method). *CategoriesTest*, *DocsTest*, *ProjectsTest*, and *TodosTest* were the main classes that the JUnit tests were created for each object. These tests included all boundary cases and cases using other data types (XML instead of JSON). *TestUtils*, *Resources* and *Server* were mentioned before and were used as library classes to help the functionality of the unit tests. Finally, the unit tests could all be run together with the *TestRunner* as a JUnit test suite.

B. Collaboration

The distribution of work during this part of the project was even. For the exploratory testing section, the members were split into two groups of two where they carried out separate 45 minute charters. As explained before, one of the people in each group would do the testing, while the other would write the charter document. This ensured that all team members understood the charter evenly and worked the same amount. These reports were shared through Discord.

For the unit testing, one member (with the help of others) was assigned to creating the project with Maven and JUnit dependencies, creating the Base Test, and a few utility tests, while the other three team members split the unit test sections evenly and wrote many tests for each of them. At the end, all team members ended up with the same amount of lines written, and the difficulty of tasks was evenly distributed.

The report was easy to split up and collaborate on, as members were responsible for writing about what they worked on specifically.

C. Findings & Recommendations

1) Exploratory Testing Results: The charters were mostly successful in ensuring the functionality of many major features in the RestAPI. Both charters made use of the full 45 minutes, and even went slightly over the limit. The first charter had 41 entries for its *Charter Notes* section, while the second charter had 46 entries.

Most features tested worked successfully, but some bugs were still found throughout the testing. The first bug that was discovered by both teams was the JSON body POST query, which is supposed to take the BOOLEAN as a string given the documentation, but instead took it as a BOOLEAN. Therefore, the JSON object shown in figure 1 was unsuccessful and returned in error codes.

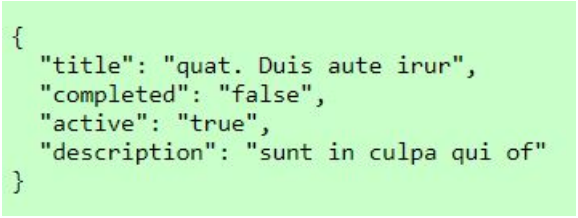

Fig. 1. Buggy JSON Object: unsuccessful with POST query

A second bug that was found in both charters was that using the PUT query to change the ID of a todo returns a successful status code, but does not change

the ID. There was no other impact of this query (no new todos were created or deleted).

The second charter found a third bug in the RestAPI, which happens when attempting to update an ID using a string. This returns an odd error saying: *"Failed Validation: id should be ID."*.

Finally, the first charter discovered a potential area of instability. When a todo or project is deleted, its ID cannot be reused and is therefore never used again. This may have been done on purpose by the developer, but does not seem like an efficient use of memory and seems vulnerable to overflow or exceeding persistence field size in the long run.

The biggest recommendation from these findings would be to revise the documentation in order to better describe how exactly the API accepts and interacts with data. Another recommendation would be to update and improve ID management, and a final recommendation would be to address bugs related to updating certain IDs.

2) Unit Testing Results: Within the JUnit test suite hierarchy, a total of 116 tests were written. These ranged across all endpoints which can be run with the RestAPI. Before each test, a method was run to start the server, and after each test, a method was run to stop the server. Overall, 105 tests passed and 11 tests failed. Below is a list of tests that failed and how they failed:

- *testGetTodosCategoriesWithInvalidID*: Category was found despite taking "ABC" as ID parameter
- *testPostTodosWithFloatTitle*: Float title is accepted for ToDo when it shouldn't be according to documentation
- *testGetTodosTasksofWithInvalidID*: ToDos task was found despite taking "ABC" as ID parameter
- *testDeleteCategoryProjectRelationshipInvalidCatIDand ValidProjID*:
- *testHeadTodosOfCategoryWithInvalidID*: Todos retrievalfrom Category through header with invalid ID returns as successful when it shouldn't
- *testHeadProjectsOfCategoryWithInvalidID*: Project retrieval from Category through header with invalid ID was successful when it shouldn't
- *testGetTodosOfCategoryWithInvalidID*: Todos retrieval from Category with invalid ID was successful when it shouldn't
- *testGetProjectsOfCategoryWithInvalidID*: Project retrieval from Category with invalid ID was successful when it shouldn't
- *test_GetProject_StringId*: Project Retrieval from root end point using invalid (string) id returns as successful when it shouldn't
- *test_CreateProject_FloatTitle*: Float title is accepted for Project when it shouldn't be according to documentation
- *test_GetProject_NegativeIntegerId*: Project Retrieval from root end point using invalid (negative int) id returns as successful when it shouldn't

These failed tests could point out potential instabilities in the system, and stem from either of two things: the improper ID management in the system, and the improper variable type management in the system. Therefore, recommendations would be similar to the exploratory testing recommendations, but would also include fixing all bugs mentioned in the list.

IV. Part B - Story Testing

The second part of the project consisted of performing Acceptance Testing on the open source RestAPI Todo List Manager through the use of user stories. Acceptance Testing is defined as the part of testing which is conducted to determine if the requirements of the project have been met. This means that each requirement or user story which has been determined before the project began is broken up into many different tests. Since User Stories are being used as the main form of requirement elicitation, test cases are broken into "Scenarios" which describe different flows of data when users are using certain features. The following user stories were tested in this part of the project:

- As a student, I categorize tasks as HIGH, MEDIUM or LOW priority, so I can better manage my time.
- As a student, I add a task to a course to do list, so I can remember it.
- As a student, I mark a task as done on my course to do list, so I can track my accomplishments.
- As a student, I remove an unnecessary task from my course to do list, so I can forget about it.
- As a student, I create a to do list for a new class I am taking, so I can manage course work.
- As a student, I remove a to do list for a class which I am no longer taking, to declutter my schedule.
- As a student, I query the incomplete tasks for a class I am taking, to help manage my time.

- As a student, I query all incomplete HIGH priority tasks from all my classes, to identity my short-term goals
- As a student, I want to adjust the priority of a task, to help better manage my time.
- As a student, I want to change a task description, to better represent the work to do.

Cucumber was used to complete the tests which in turn uses Gherkin, a simple language that enables us to describe system behavior and acceptance tests. Cucumber is a tool made for proper Agile and Behavior Driven Development and allows for iterable automated testing implemented through use cases. The structure for Gherkin Scripts will be discussed in the following section.

A. Implementation

1) Gherkin Script Construction: The first part of story testing is to first write the Gherkin scripts. Gherkin scripts describe a feature or a requirement in human readable language. Our Gherkin scripts explain who the user is, what they want to do and why they want to accomplish that specific feature. Before going into the details of each scenario, our gherkin scripts contain a background section that describe the steps that prefix each scenario. Each feature may contain many scenarios that can be broken down into three main categories: normal flows, alternate flows and error flows.

- A normal flow is the ideal path that is taken by the user and system when using the feature
- An alternate flow is a flow that differs from the normal flow, but still ends in the system successfully executing the feature.
- An error flow is a path that would lead to a system failure

Furthermore, each scenario in a Gherkin script has a Given, When, and Then section which all contain multiple steps:

- Given: describes the preconditions and the initial state of a scenario
- When: describes the conditions that trigger the outcome of a scenario
- Then: describes the outcome or end state of a scenario

Tidy Gherkin is a useful tool for writing the Gherkin scripts because we can then generate a skeleton for the java test class. Some of our group members used the Gherkin and Cucumber plugins available within IntelliJ to generate the skeletons as well. The test classes are grouped together to form a test suite, containing all the tests.

2) Story Test Suite: The Acceptance Testing was categorized and split into multiple classes; one class was made for each feature we are testing. In addition to those files, we grouped up common methods into the *BaseStepDefinitions.java* file and we have the *CucumberFeaturesTestRunner.java* file, which runs the test suite. Each of the *...StepDefinitions.java* files extend the file *BaseStepDefinitions.java* to make use of the *setup()* and *teardown()* functions as well as to properly have a set of variables to be shared between tests. These variables are given certain scenarios such as the ones in *QueryIncompleteTasksStepDefinitions* as half of the feature test functions are to be found in the class *QueryIncompleteHighPriorityTasksStepDefinitions*. Such doubling of use of functions was done to have shorter and more readable code.

Apart from the classes containing step definitions, we have several other helper classes. The *Resources* class it is used by both Unit and Acceptance testing to hold all the needed Static and final variables. These variables are either Status Codes or are URLs/Ports. Regarding the Server class which used to be in Unit in the prior deliverable it serves to properly boot up and shut down the RestAPI Todo Manager. It is called in the previously mentioned *setup()* and *teardown()* functions. Finally, the Utils java class serves to provide a basic set of utility functions to further ease the implementation of each step.. These functions provide a way to obtain either the project IDs, remove an object, verify existence or count its members. Each function has as well its overrides given if one uses as arguments the ”title” or ”id” of the object and *Utils* provides as easy way to obtain RequestSpecifications using RestAssured for all the needed http requests. RestAssured is a lightweight library that is used throughout the rest of the project.

B. Collaboration

The work was split evenly between group members as each member wrote the Gherkin scripts and implemented

the test classes for 2-3 of the user stories. The members who worked on 2 user stories contributed more towards the report for that deliverable and members would assist each other whenever someone needed assistance.

Before writing out each of our Gherkin scripts, we planned out which step definitions would be common between each test class, which we grouped together in the *QueryIncompleteTasksStepDefinitions* and *QueryIncompleteHighPriorityTasksStepDefinitions* files as mentioned previously. That way, the workload would be lessened and group members would be able to use each others' test methods when implementing the tests in the test suite. When working on the test suite, collaboration between group members was facilitated through the use of GitHub, which was used for sharing code and for version control.

C. Findings & Recommendations

A total of 61 scenarios were tested as each feature had varying amounts of scenarios. After testing the features in multiple orders (which was done simply through testing the features individually as well as group-wise), the results obtained were that all tests successfully passed as seen in figure 2. All the scenarios are presented in this figure as well. Given the use of Gherkin grammar, these scenarios are well self descriptive and do not need to be further discussed to avoid redundancy in this paper. Though it is worth mentioning that each scenario may have more than one "Example" as per multiple occasions to further test in completion the Todo List Manager.

Now while the scenarios all pass as shown, some assumptions were made to have them function properly. One assumption is that the "ID"s when provided in a JSON Body for GET commands had to be of data type string to function properly. Failure to provide it as a string would result in a 4xx error. The documentation simply says it has to be of data type "ID" which is assumed to be a typo, this can be seen in figure 3.

Another assumption is that the scenarios mentioned "tasks" in terms of the the student priority todos, but those were in fact todos given that they had to be categorized in terms of priorities (priorities were set as categories given that there is no such thing as a priorities field). As such it is assumed the user understands the difference between a task and a todo. This can be

Fig. 2. List of Every Scenario Tested in Part B

project

Fields:

Fieldname	Type	Validation
id	ID	▪ Mandatory?: false
Example: "86"		
title	STRING	▪ Mandatory?: false
Example: ", consectetur adipis"		
completed	BOOLEAN	▪ Mandatory?: false
Example: "false"		
active	BOOLEAN	▪ Mandatory?: false
Example: "true"		
description	STRING	▪ Mandatory?: false
Example: "itation ullamco labo"		

Fig. 3. Fields for Projects from http://localhost:4567/docs

confusing especially given the fact that a todo that is attached to a project is obtained or posted using a "tasksof" in the url- this can be see in figure 4.

/todos/:id/tasksof

e.g. `http://localhost:4567/todos/:id/tasksof`

- **`GET /todos/:id/tasksof`**
  - return all the project items related to todo, with given id, by the relationship named tasksof
- **`HEAD /todos/:id/tasksof`**
  - headers for the project items related to todo, with given id, by the relationship named tasksof
- **`POST /todos/:id/tasksof`**
  - create an instance of a relationship named tasksof between todo instance :id and the project instance represented by the id in the body of the message

Fig. 4. Relationship Http Requests for Todos

V. Part C - Non-Functional Testing

In this part, a static analysis was done on the software using SonarQube, this in turn allowed us to see the smelly code, bugs and warnings of the code as well as the technical debt. On top of a static analysis, a performance analysis was done on a non isolated Windows 10 system to see the impact of addition and deletion of multiple objects, in the software, on the machine. The RAM usage as well as the processor usage were the two tracked resources.

A. Implementation

1) Static Analysis: Static Analysis was run on the entire To-do manager Rest API source code in order to find any potential code inconsistencies, errors, or debt. This was done through the SonarQube tool which was able to spot many potential issues with the code. SonarQube was run with the help of Docker and hosted as a local server on the tester's system. To create the specific analysis instance, Command Line commands and Maven were used, which returned a analysis report under *localhost:9000*.

SonarQube is an open-source platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells and security vulnerabilities. The version used was the community edition, which is available for free and although it lacks features that the premium versions have, it is sufficient for the testing of the source code of the Rest API.

2) Dynamic Analysis: The first part of the dynamic analysis of the To-do manager Rest API was to perform Performance tests on the system. Performance testing is a process used for testing various properties of a software application under a particular workload such as speed, response time and resource usage. The test suite from the previous deliverable was modified to implement a performance test suite, used for Dynamic Analysis, a form of non-functional programming. To do this, *logging helper functions* were first implemented. These functions are used to take data which is outputted by the performance test and log it into a time stamped log file. After these methods were implemented, each unit test in the test suite was modified to log both the starting and ending time of the test, and then log these two numbers and the overall execution time into the correct log file. Finally, more advanced performance testing functions were created which populated the database with many entries, then performed tests adding, deleting, and modifying every type of data in the table (to-dos, projects, categories).

The second part of the dynamic analysis process was to measure CPU usage and Available Free Memory used by the system for each of the advanced performance test experiments. This was done using the Windows Performance Monitor tool, and logged.

B. Collaboration

The analytical work was split evenly between group members as each member either worked on the dynamic or static part of the project. One pair of the group ran the SonarQube on his machine and shared the results with his team mate to then analyze its results while the other pair had as well one member code in the logger functions as well as the object transaction tests which allowed the other member to run said tests which were then analyzed by both members.
Each pairing wrote the analysis for their part, then added their input on the results of the other pair where needed and contributed it to the paper. The supplementary parts of the paper such as introduction, conclusion, and previous deliverable summaries were split evenly as well.

C. Findings & Recommendations

1) Static Analysis: The results of the static analysis and their related testing fields were as follows:

- Reliability: 5 bugs
- Security Review: 1 Security Hotspot
- Maintainability: 4 days and 2 hours of technical debt; 477 code smells

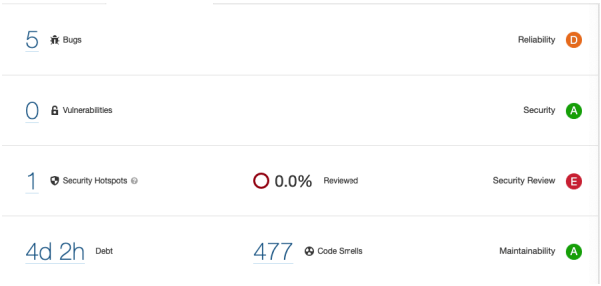

Fig. 5. SonarQube Static Analysis Results

It can easily be seen that the main problem with the Source code is that it is very hard to maintain. This can be seen from its extremely high amount of technical debt, which is the amount of time that

would be lost by not creating efficient code without redundancies and order, and the numerous code smells contributing to this debt. A recommendation to fix this problem would be to create many more common ”utility” functions which can be used by all classes, rather than making a new similar function for each class. An example of this is in the *AcceptHeaderParser* class in the *isSupportedHeader* method. This method checks whether the *acceptMediaTypeDefinitionsList* is empty by comparing the size of the list to 0, which is less efficient than using the common *isEmpty()* method, which would return true if the list is empty, and false otherwise. This method is more efficient and would reduce technical debt (since any bug with this method could be switched once instead of at every instance where a similar method is called), which is why it is recommended by SonarQube.

Another useful recommendation which is mentioned many times in the Code Smells is to replace all *System.out* and *System.err* calls by a logger. This is very useful as all information will be in a few easily accessible log files which can be time stamped and accessed in future iterations of the project.

SonarQube’s inspection of the source code detected 5 bugs with varying severities as shown in fig 6. According to SonarQube, bugs are coding errors that will break the code and needs to be fixed immediately. Out of the 5 bugs, there was 1 critical bug, 3 major bugs and 1 minor bug. A critical bug is a bug with a low probability of impacting the behavior of the application. In our case, a new *Random* object is created each time, which is inefficient and may produce numbers which are not random.

A major bug is a quality flaw which can highly impact the developer productivity. 2 out of the 3 major bugs detected were caused by a possible *NullPointerException* because a reference to *null* should never be dereferenced or accessed. In the worst case, this may lead to exposed debugging information useful to an attacker or other possible security issues. The other major bug detected was that the return values from functions without side effects should not be ignored or else the program may generate false-positives.

On the other hand, a minor bug is a quality flaw which can slightly impact the developer productivity. The minor bug in the source code is that math operands should be cast before assignment or the result will likely not be what was expected. SonarQube was very useful in detecting these bugs and it estimated that it will take around 40 minutes to fix all 5 of them.

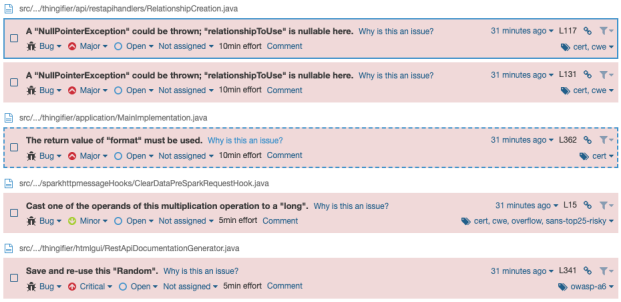

Fig. 6. Bugs detected by SonarQube

2) Dynamic Analysis: As previously mentioned a study was done on the delta transaction time given number of objects to see the impact of the increasing amount has on it. Multiple graphs were drawn to aid for the proper analysis of the resources used. The figure 7 serves as an example to demonstrate the graph type.

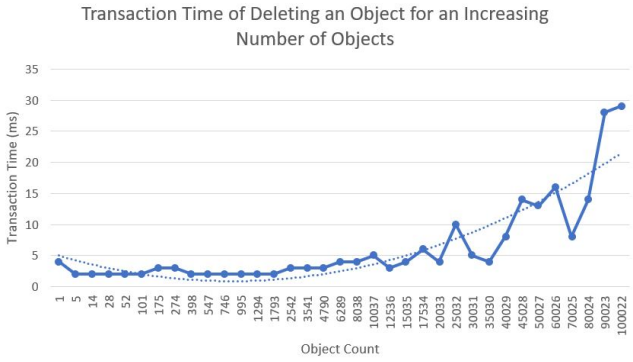

Fig. 7. Transaction Time of Deleting an Object for an Increasing Number of Objects

As it can be seen given that graph title, the tests were split in the three main functionalities; object changing, addition, and deletion. Overall, a lot of the graphs demonstrated a higher than expected transaction time at the beginning. This is explained in the Rest-Assured API as caused by the fact the JVM is not ”hot”. More specifically are the following observations:

- Object Changing: There is an increasing amount of transaction time as more and more objects are changed (as they reach the thousands). This is expected as objects must be searched and found to be changed.

- Object Adding: Adding a higher amount of objects barely impacted the transaction time. This is in turn is most likely due to the fact that an added object is simply put at the end of the list of the already existent objects, no searches need to be done.
- Object Deletion: As the object amount transacted increase there was more defined impact on the transaction time. This can be accredited to the fact that the delete function must not only find the object but change (remove) a much larger amount of data than the simple change function that was previously discussed

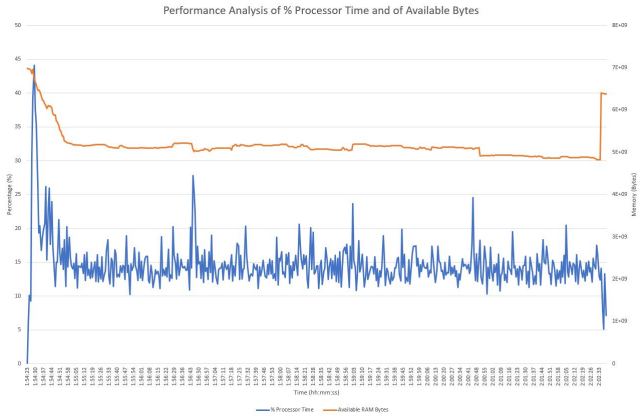

Fig. 8. Performance Analysis of % Processor Time and of Available Bytes

As for the computer resources that were tracked they were the memory in bytes available and the % processor time. Now figure 8, it shows the impact on the computer's % processor time of the increasing amount of transacted objects. At the beginning of the graph there is a huge spike, which is mainly caused by starting up the tests through the IDE, but it can be later seen that the impact if any is neglectful throughout the process. There are spikes at the end of each tests but that can also be attributed to switching tests classes. The figure as well shows the impact on the computer's memory use of the increasing amount of transacted objects. A lot more obviously than the CPU graph, there is almost no impact on RAM given the number of objects. There is a slight dip in available memory space which comes back up at the end only when the server is booted and shutdown. A graph which made use of the times of each test was used to cross reference the resources used and the objects transacted; figure 9.

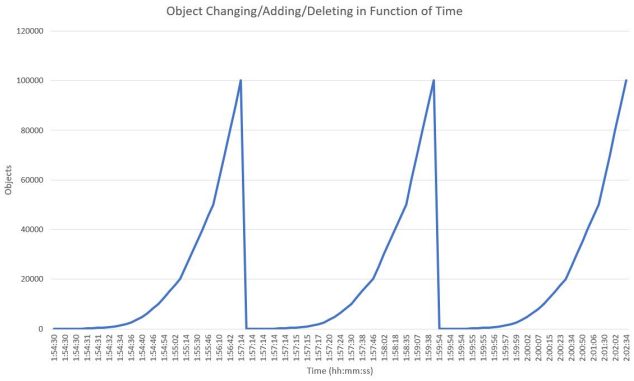

Fig. 9. Object Changing/Adding/Deleting in Function of Time

REFERENCES

[1] "rest-assured/rest-assured", GitHub, 2020. [Online]. Available: https://github.com/rest-assured/rest-assured/wiki/Usage#measuring-response-time. [Accessed: 03- Dec- 2020].