Group 98:
Samantha Ines Perez Hoffman (261039555)
Lucy Zhang (261049310)

Summary of Deliverables

The ECSE 429 Software Validation Term project is divided into the following three deliverables:
- ➢ Part A: Exploratory Testing of Rest API
- ➢ Part B: Story Testing of Rest API
- ➢ Part C: Non-Functional Testing of Rest API

Part A: Exploratory Testing of Rest API

In this stage, we needed to perform some exploratory testing of the rest api of the "rest api todo list manager" application. This application consists of a simple todo manager that has categories and projects. The capabilities of the todo list manager can be divided into main four API groups:
1. Todos
2. Projects
3. Categories
4. Interoperability

We were able to run this application locally in our computers in order to achieve the following goals:
1. Do some exploratory testing
2. Write a unit test suite
3. Write a report

In the exploratory testing step, we needed to study the behavior of the "rest api todo list manager" application by following the given "Charter Driven Session Based Exploratory Testing" throughout different sessions. In each session, the aim was to explore the capabilities as well as the areas of potential instability of the application under test.

In the unit testing step, a unit test suite was required to be created. In the suite, we needed to create at least one unit test for every API mentioned in the exploratory testing sessions, as well as creating tests for different edge cases for those APIs. A video recording where all the tests run and pass was also required.

In the report step, we were required to write this report summarizing the deliverables, the findings from our testing as well as the structure of our unit test code repository.

Part B: Story Testing of Rest API

In this stage, each team member will be required to define five user stories that need to be related to the different APIs that we've explored in the first deliverable of the project. We will then need to implement a Story Test Suite, where we define at least three acceptance tests for each story, and implement Story Test Automations, where we implement step definitions through gherkin scripts. Finally, similarly to part A, we will again need to write a report summarizing the work that we did for that deliverable.

Part C: Non-Functional Testing of Rest API

In this stage, we will need to implement two types of non-functional testing of Rest API: performance testing and Static Analysis. For the performance testing, we will be using dynamic analysis while for the static analysis testing, we will be performing analysis of the source code like code complexity or technical debt. This final deliverable will also require us to write a report that summarizes the implementation of the non-functional testing as well as any recommendations we might come up with to improve the source code of the application under test.

Findings of Exploratory Testing

Exploratory testing was divided into two sessions, session 1 Todos endpoints and session 2 for Projects endpoints. Notes from these sessions can be found in files SessionNotes_Todos_PartA.xlsx and SessionNotes_Projects_PartA.xlsx.

To explore the different endpoints available and identify the capabilities of the application, postman was used to request different endpoints. Documentation helped get us started as to what the different endpoints were and how they may function.

Screenshots of the postman requests as well as the responses we received can be found at Session_Notes/Session1/Screenshots and Session_Notes/Session2/Screenshots respectively.

<u>Session 1: Todos</u>
For the Todos endpoints, we only ended up having time for some of the basic functionalities in the short 45 minute exploratory testing session that we had. The following APIs were tested:

- /todos (GET, HEAD and POST)
- /todos/:id (GET, HEAD, POST, PUT, DELETE)

For the most part, all the tested functionality worked as expected through following the given documentation. However, at times it definitely felt like the documentation lacked clarity and details. For example, I struggled for a significant portion of the session to figure out how to properly use the field values in the body of the message. I believe that some additional documentation could have really enhanced my overall user experience.

The endpoints mentioned above were also tested for their edge cases. For instance, I tried performing a GET http://localhost:4567/todos/:id where the id field passed didn't match any of the ids of the todos that were currently in the database at that time. This allowed us to confirm that the endpoint does in fact return a helpful error message. However, I found that not all edge cases were treated with the same level of clarity. For example, when you would perform a HEAD http://localhost:4567/todos/:id, with an id that wasn't in the database, the only thing that would indicate that it failed was the status code returned in the response. I think that including an error message in the body of the response would be very beneficial for the user experience of the application.

Session 2: Projects

For the project endpoints, basic functionalities were tested (GET all projects, HEAD-ers of all projects, POST a new project). These were also tested for edge cases like posting a project with an empty body or having the wrong format in the request body. It mostly behaved as expected, as the title, id, and such parameters seemed to be returned correctly in the response body. However, what needs to be further explored is using query parameters. Documentation mentioned that the project endpoints could be filtered with fields as URL query parameters with not much more details. Setting the title in the query parameter in the post did not quite work, but this felt like a lot of information that could be explored in another testing session.

Another functionality was the creation of relationships between projects and tasks (also known as todos). A potential bug or unclarity was in the edge case of GET-ing tasks with a project Id that did not exist. As a tester, we understood that if the project did not exist, then no tasks should be associated with a non-existent project. However, the unexpected happened with this where other tasks were returned regardless. This may have either been a bug or misunderstanding with how the documentation conveys the endpoints. This finding was later confirmed in the unit tests. It was also observed that when tasks are returned, there is a field indicating the id of the project it is associated with. Otherwise though, the task functionality and relationship with projects seemed to work as expected.

Categories also played a part within the project endpoint section. There is a similar relationship between tasks and projects as described above. There was a similar bug mentioned with tasks that was noticed for categories. This was when we realized that the bug may be just noticed because of unclear description within documentation. Within this functionality, it was also tested the possibility of creating categories within projects that did not exist. This functioned as expected. Since we were a team of 2, we did not have the chance to deeper explore and test categories, but this could be a goal for a future session.

Structure of Unit Test Suite

As we were only a team of 2, we were only required to focus on the capabilities related to the todos and projects when it came to performing the exploratory testing as well as implementing the unit test suite.

The unit test suite used JUnit as a testing framework. There are three test files within the source code: TodosTest, Projects Test, and SystemTest. The first are self-explanatory: they are divided into the two main functionalities that we focused on in this exploratory testing report. The last file's purpose is to test that the system is ready to be tested; in other words, the application is up and running correctly.

As for the TodosTest and ProjectsTest files specifically, they each include main APIs that will validate the core functionalities of Todos and Projects. They also include edge cases when possible for all APIs, including the cases where an xml request body lacks a </Project> or when the json body is malformed. It also includes cases like the json request body is empty, which in some APIs, the request is still expected to go through correctly.

To correctly validate APIs in an isolated manner, it was also important that the unit tests would not be affected by other unit tests. Hence, to ensure this, we made sure that each unit test would start at the same start point, and if any changes were made to the database within the test, it would be reverted back at the end. Another crucial way to ensure isolated unit tests is through adding @TestMethodOrder(MethodOrderer.Random.class) at the beginning of the test files. This would allow the unit tests to run in random orders to ensure that the testing sequence would not affect the results of the tests. We can also confirm that the APIs are actually working and not because of certain specific setups that would cause it to work.

@BeforeAll was also important to set up the tests correctly before running. An HTTP client was needed to make requests to the endpoints and then validate the response and also content of the responses when possible.

Source Code Repository

Here is a breakdown of our Github Source Code Repository:

```
.
├── README.md
├── bin
│   ├── ProjectTest.class
│   ├── SystemTest.class
│   └── TodosTest.class
├── lib
│   ├── gson-2.11.0.jar
│   └── junit-platform-console-standalone-1.11.2.jar
├── session_notes
│   ├── session1
│   │   ├── SessionNotes_ToDos_PartA.xlsx
│   │   └── screenshots (folder with session 1 screenshots)
│   └── session2
│      ├── SessionNotes_Projects_PartA.xlsx
│      └── screenshots (folder with session 2 screenshots)
├── src
│   ├── ProjectTest.java
│   ├── SystemTest.java
│   └── TodosTest.java
└── unit_tests_all_running_randomly_video.mov
└── ECSE 429 Part 1 Report - Group 98.pdf
```

We decided to have all of the different documents that we needed to submit for this deliverable in a Github repository so that it's easy to navigate.

Findings of Unit Test Suite Execution

Below is a summary of the different findings we uncovered through the unit test suite execution. A video showing that all tests run correctly is attached as unit_tests_all_running_randomly_video.mov. You can observe that every time the tests are ran, the order of the unit tests are randomized.

Todos Unit Tests

The Todos Unit Tests allowed us to test the related capabilities a lot more meticulously. It allowed us to inspect more thoroughly the response returned by the APIs.

For instance, through including tests of invalid operations for each API identified in the exploratory testing stage, it allowed us to find some additional edge cases that we hadn't focused on before. In fact, it allowed us to check how the APIs behave when giving a request with a malformed JSON or XML payload. They are in fact capable of dealing with such edge cases.

Furthermore, by having to check if the different unit tests run in any order, it allowed us to notice how the different APIs calls interacted with each other as well as the database. In fact, only the PUT, POST and DELETE endpoints affect the database.

Projects Unit Tests

The unit tests allowed the project functionality to be tested more rigorously. As mentioned earlier, it helped confirm a bug that was potentially found when doing exploratory testing. To make sure that our observations were correct, within the test for the bug, an AssertEquals with the wrong response value and an AssertNotEquals with the right response value was written. It confirmed our observations correctly consistently. This bug was identified when trying to get tasks and get categories with a project id that did not exist. Tasks and categories were returned regardless, which was not an intuitive response.

Also, the fact that writing certain test cases individually made us think "what could go wrong with this specific request" helped us isolate the scope of thinking and write up more granular potential risks. For example, during exploratory testing, we did not think of the case that someone may try to create a task within a project that did not exist. We thought of what would happen if they were trying to delete an object that did not exist, but not if they tried to create a relationship with an object that did not exist. This makes testing more rigorous and to have a better understanding of the areas of instabilities of the application.

With more granular tests, we were also able to vary for all possible APIs the valid and non-valid json and xml request bodies to ensure that the application could accept/return different formats or respond well to malformed request bodies.

System Tests

Testing that the application is running correctly is important, as not only does it ensure that the application is, first of all, running, it also confirms that the other unit tests are passing/not passing within the expected environment, especially when the entire suite is run all together. This was good to test whether the other unit tests were passing when the application was no longer running in the background. As all the other unit tests failed, the system test also failed as expected. Command Line queries were also validated and ran as expected.