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

We submitted this part of the project on October 8, 2024. In this stage, we were able to run the REST API To-Do List application locally, do some exploratory testing, write a unit test suite, and write a report about our work and findings. Following this part, we were able to use our notes and reports to continue on our work in testing the REST API To-Do List application.

Part B: Story Testing of Rest API **(this report)**

In this part, we are testing the REST API To-Do List application through the lens of the end users through story testing. As a group of two, we defined a total of ten user stories each describing an available API that users could use. We mainly focused on the API groups Todos and Projects.

These APIs groups were already explored in Part A, but now, we get to create user scenarios and run automated tests in a more user-centric context. To get this done, we wrote Gherkin tests and ran them using Cucumber.

Through this part of the project, we ultimately were able to:

1. Define ten user stories based off documentation + notes from our Part A
2. Include at least one normal flow, one alternative flow, and one error flow in each scenario
3. Write step definitions to perform the actions specified in the Gherkin feature files
4. Create a report summarizing our work and findings

The ten user stories were divided evenly (five-five) for API groups Todos and Projects. We needed to write the user stories in the right form to run the Gherkin script and include any

background needed to set up any initial conditions of the automated tests. Each flow also describes different situations that may occur from the user perspective.

As for the step definitions, we had to connect the user outlines we had defined within the Gherkin files and validate the correctness of the results from the actions. The tests had to be able to run in a random order, so that all cases could be covered and so that we could thoroughly test the application from the user perspective. A video showing the tests running and running in different order was attached.

The report summarizes the content of the deliverables, the structure of the story test suite, a description of the source code repository, and the findings from executing the story test suite.

Part C: Non-Functional Testing of Rest API

This is the next part of the project to complete. We will need to implement two forms of non-functional testing: performance testing and static analysis. For the performance testing, we will be using dynamic analysis, testing the application under different conditions of speed and stability to ensure that the application meets performance requirements. As for static analysis, we will be analyzing the performance without executing the code. Instead, we will analyze the source code for code complexity or any technical debt.

This will also be the final part of the term project. The report will include any description of implementing the non-functional tests along with any recommendations we might have to potentially improve the source code of the REST API To-Do List application.

Structure of Story Test Suite

Since we were a team of two, we wrote a total of ten user stories with each having its own feature file within the resources directory of the test folder. Five of the user stories focused on the application's todos feature and the other five focused on the application's projects feature.

The story test suite had two main components, the feature files and the step definition files. The feature files were written in Gherkin, and they are scripts that could be run in an order that describes a user using the features from their perspective. Each feature file corresponded to a user story or API feature. As for each scenario outline, we followed the format of: given some sort of condition, when we want to perform a certain action, then these things would happen, which will validate the correctness of the feature. There is also an examples section, and this is used to run the same scenario outline with different values for certain variables within the scenario outline. Testing multiple values ensures that the correctness of the results works for multiple different expected inputs.

As for the step definitions, a few files were created in the stepdefinitions directory: ResponseContext.java, CommonStepDefinitions.java, ProjectStepDefinitions.java, and TodosStepDefinitions.java. The ResponseContext.java ensures that the correct response is passed to the step definitions, because certain step definitions are reused across different acceptance tests within the CommonStepDefinitions.java file. This file includes validating the response code or validating the error message, which are common validation steps. As for the step definitions specific to the two API groups, the functions are picked up by @Given, @When, @Then where the step string would match and correspond to each other between the feature files and step definition files. We were able to ensure that all scenario outlines were covered, because otherwise, the tests would not run.

For the project Gherkin features, we defined user stories for get project, create project, edit project, delete project, and create project task features. For the todo Gherkin features, we had user stories for get todo, create todo, edit todo, head todo, and delete todo features.

The tests were run using Cucumber as a framework, and we used IntelliJ as our IDE to run the Gherkin scripts. To ensure that the tests would run in different order, we would right click on the features folder and run all features. Each time this is run within IntelliJ, the features run in different order. We also know that the tests are correctly set up because the actions are performed on objects that were created for specifically each scenario.

As required in the assignment, we also made sure that the tests would fail when the application was not run; we also demonstrated this on our attached video within the repository. The way the APIs are called are through using a base URL that uses http://localhost:4567/, which is the port that is used when the application that's being tested is run. We used the java library RestAssured to test and validate REST APIs.

Description of Source Code Repository

We chose to include all the different components that were needed for this deliverable's submission in the same GitHub repository in order to make it easy to find everything. This includes all the user stories, the story tests, the video of the tests running randomly and even the report. The repository also includes a README.md file introducing the reader to this project and this deliverable. We also included how to run the story tests that we wrote as well as the details on the devices that we've tested our code. In general, we need to ensure that we're running the "rest api todo manager" application on the terminal in the background before running the story tests.

A crucial file for the test suite to come together is the pom.xml file as it contains all the dependencies we needed to run the Gherkin scripts using a Cucumber environment. We also

needed to include the dependencies for RestAssured, the java library we used to test and validate REST APIs specifically.

Below is a breakdown of our GitHub code repository:

```
├── README.md
├── pom.xml
├── src
│   └── test
│       ├── java
│       │   └── org
│       │       └── ecse429
│       │           └── stepdefinitions
│       │               ├── CommonStepDefinitions.java
│       │               ├── ProjectStepDefinitions.java
│       │               ├── ResponseContext.java
│       │               └── TodosStepDefinitions.java
│       └── resources
│           └── features
│               ├── ID001_Get_Projects.feature
│               ├── ID002_Create_Project.feature
│               ├── ID003_Edit_Project.feature
│               ├── ID004_Delete_Project.feature
│               ├── ID005_Create_Project_Tasks.feature
│               ├── ID006_Get_Todos.feature
│               ├── ID007_Create_Todo.feature
│               ├── ID008_Head_Todo.feature
│               ├── ID009_Edit_Todo.feature
│               └── ID010_Delete_Todo.feature
├── story_tests_all_running_randomly_video.mov
└── ECSE 429 Part 2 Report - Group 98.pdf
```

Findings of Story Test Suite Execution

This deliverable really pushed us to think in depth about a wider range of test cases - more specifically the edge cases.

This is most likely due to the requirement of including the following three types of acceptance tests per user story:

1. Acceptance test representing a normal flow
2. Acceptance test representing an alternate flow

3. Acceptance test representing an error flow

Through exploring different scenarios, it allowed us to have an even better understanding of the application under the test: "rest api todo manager" application.

Below is a summary of the different findings or observations we made through writing and running the story test suite.

Todos Feature Findings

For the story tests related to the todos user stories, this deliverable really motivated us to test the edge cases of the "rest api todo manager" application that we didn't think about that much during our first deliverable as we were focused on other matters.

In fact during our last deliverable, we were focused on unit testing. This means that our attention was checking that an individual piece of the application in isolation was behaving correctly. On the other hand, for this deliverable, we had the opportunity to focus on story testing which allowed us to think more about full user stories and seeing how, in the perspective of the user, the todo list features worked.

Through implementing these user stories, we were able to think about other valid use cases of a feature and also potential ways that a user story could fail. This allowed us to verify that the "rest api todo manager" application was handling those rarer but still possible scenarios properly.

The following todos features were tested:

- ➢ Get todos
- ➢ Create todos
- ➢ Retrieve todos headers
- ➢ Edit todos
- ➢ Delete todos

For the most part, we were able to successfully verify that the different features properly handled the normal flow but also error flows that check how the application handles APIs when input is given in an invalid format (ex: expected a boolean and received a string) or that check how the application handles APIs when it needs to perform an operation on todo that doesn't exist in the system. Alternate flows were also tested for the todos features such as cases where different valid input fields are given.

Projects Feature Findings

The story test suite allowed us to test out the project feature from a more user-centric approach, where we focused on how end-users would interact with the REST API To-Do List application.

We were able to automate testing for creating, editing, and deleting projects, as well as managing tasks within each project. Through testing the features from a user perspective, we were also able to gain more insights on how usable the application is and how it could be improved.

One of the findings that made us reflect is the fact that users are able to create a project without a title, but they are required to include a title for the tasks. It made us wonder why this design decision was made or whether it was a mistake, because it could be unintuitive for the user. Having a similar pattern to create entities within the application would make this application more usable. This was discovered as we were writing the story tests from the user perspective and realized that when creating a project versus a task, the different possible flows were different when switched. This gave us a better insight on the user experience when creating different entities within the application.

Also, for the error flow, we made sure to test out some wrong formats for the request body. We questioned how user friendly it is for the user to use this application without a front-end and whether this would be a common use case of the APIs. The error message also would be confusing for an end-user that is not a software developer as it mentions the json format and an error at a specific line. These are all thoughts that we had when running the tests on a more use case level from an end-user perspective.