ECSE 429 Part B Report

Antony Zhao (260944810), Ziyue Wang (260951986), Peter Yu (261005015)

Introduction

This report offers a thorough examination of the entire building process and story testing for the 'REST API Todo List Manager.' This system is engineered to manage to-do lists effectively, facilitating operations across different entities such as tasks, categories, and projects. Our team constructed fifteen user stories, concentrating on uncovering potential issues that emerge when the application receives multiple consistent instructions from the user.

Deliverables

Features: Behavior description of user stories and code implementation of each individual steps. Each test consists of a normal flow, alternative flow, and error flow. A total of 15 tests are designed to test 15 different user stories, confirming the correctness of each corresponding feature.

User Story	Given	And	When	Then
As a student, I add a task to a course to do list, so I can remember it. From: add_task_to_course.feature	POST projects GET projects	POST todos GET todos POST todos/:id/tasksof	POST todos GET todos/:id POST todos/:id/tasksof	GET todos/:id
As a student, I want to adjust the priority of a task, to help manage my schedule. From: adjust_priority.feature	POST todos POST categories POST categories/:id/todos	POST categories/:id/todos	DELETE categories/:id/todos POST categories/:id/todos	GET categories/:id
As a student, I want to change the title of a category, to specify the works. From: category_title.feature	POST categories GET categories		PUT categories/:id	GET categories/:id
As a student, I want to update the projects' priority, to better manage the tasks wait to be done. From: change_project_priority.feature	POST projects POST projects/:id/categories	GET projects/:id	DELETE projects/:id/categories/:id POST projects/:id/categories	GET projects/:id
As a student, I want to update the todos' priorty, to better manage the tasks wait to be done. From: change_todo_priority.feature	POST todos POST todos/:id/categories	GET todos/:id	DELETE todos/:id/categories/:id POST todos/:id/categories	GET todos/:id
As a student, I create a to do list for a new class I am taking, so I can manage course work. From: createToDoList.feature	POST projects GET projects		POST projects GET projects	GET projects
As a student, I want to replace a project object with newer version by delete-recreate, to better managing my projects list. From: drop_add_new_project.feature	POST projects		DELETE projects/:id POST projects	GET projects
*As a student, I want to **fix** the mistake of wanting a todo but wrongly created a project under categories, so that I can get a correct todo under categories.* From: fix_category.feature	POST categories POST categories/:id/projects		DELETE categories/:id/projects POST categories/:id/todos	GET categories/:id

As a student, I mark a task as done on my course todo list, so I can track my accomplishments. From: mark_task.feature	POST todos GET todos		Put todos/:id	GET todos/:id
As a student, I query the incomplete tasks for a class I am taking, to help manage my time. From: query_incomplete_class.feature	POST todos POST projects Post projects/:id/tasks	POST todos POST categories/:id/tasks		GET projects/:id/tasks
*As a student, I query all incomplete HIGH priority tasks from all my classes, to identify my short-term goals.*From: query_incomplete_high_priority.feature	POST todos POST categories POST categories/:id/tasks	POST todos POST categories/:id/tasks		GET categories/:id/todos
As a student, I remove an unnecessary task from my course to-do list, so I can forget about it. From: remove_task_from_course.feature	POST projects GET projects	POST todos GET todos/:id	GET todos/:id DELETE todos/:id GET todos/:id	GET todos/:id
As a student, I remove a to-do list for a class which I am no longer taking, to declutter my schedule. From: remove_todo_list.feature	POST projects GET projects		GET projects GET projects/:id DELETE projects/:id GET projects/:id	GET projects
As a student, I want to change a task description, to better represent the work to do. From: task_description.feature	POST todos GET todos		PUT todos/:id	GET todos/:id
As a student, I categorize tasks as HIGH, MEDIUM or LOW priority, so I can better manage my time. From: taskpriority.feature	POST todos POST categories GET todos GET categories	GET categories/:id	POST todos/:id/categories	GET todos/:id

Additional Bug Summary:

During testing the scenarios of user stories, we found the following bugs, which their outputs did not fit our expectation.

Bug I: Unsuccessful Linkage of New Task Categories

Description: Failure to link new categories to tasks despite successful unlinking of previous categories.

Failed Scenarios: features/adjust_priority.feature:19-21 and :33-35 - Issues changing task priority to low and high.

Bug II: Incomplete Task Linkage to Class

Description: Classes expected to be linked to at least one task show no tasks in their list.

Failed Scenarios: features/query_incomplete_class.feature:16-18 - Incomplete task querying issues for a class.

Bug III: Incorrect or Unreachable Endpoint Status Codes

Description: Endpoints either not reached or returning incorrect status codes.

Failed Scenarios: features/query_incomplete_high_priority.feature:16-18 - Issues querying incomplete high-priority tasks.

Bug IV: Task-Category Linkage Failure

Description: Assigned tasks are not linked to their designated categories.

Failed Scenarios: features/task_priority.feature:16-18 - Linking tasks to categories.

Bug V: Category-Task Linkage Failure

Description: Newly assigned categories not linked to their tasks.

Failed Scenarios: features/task_priority.feature:29-31 - Linking categories to tasks.

Bug VI: Occupied IDs Post-Category Deletion

Description: Deleted category IDs remain occupied, indicating data integrity issues.

Failed Scenarios: features/change_project_priority.feature:15-17 - Updating the category of a project.

Overall Bug Summary

These bugs highlight critical issues in the task management application, primarily around task and category linkage, query failures for incomplete tasks, and priority adjustment. Incorrect status code handling and occupied IDs post-deletion raise concerns about database management and API functionality, impacting overall application reliability and user experience.

Additional Story Test Consideration: To ensure the story test fail is not running, we involve one extra environment variable to disable the service during initialization environment for each scenario. More details will be introduced in the Behave Test Suit part and codes. And the overall effect will be seen in the video.

Story Test Suit Video: For the video of the live demonstration, the video (*Project PartB Video.mkv*) shows the execution of unit tests in random/default order and demonstrates when the server is down, all testcases are failed.

Code Repository

The following is the folder structure of our repository:

```
ECSE 429 Project Code Repo
├── README.md
├── Todo-Manager-swagger.json
├── features
│   └── PART B Code
├── requirements.txt
├── runTodoManagerRestAPI-1.5.5.jar
├── starter.bat
├── test.bat
├── test_random.bat
├── test_server_down.bat
└── tests
    └── PART A Code
```

- ***Todo-Manager-swagger.json***: This file is a JSON file that contains the Swagger documentation for the Todo Manager REST API downloaded from http://localhost:4567/docs/swagger. This file describes the endpoints, request/response structures, and other important information about the API. Postman can import this file for the manual testing session.
- ***requirements.txt***: This text file lists all the Python dependencies the Todo Manager Rest API requires. Package managers (pip) use it to install all the libraries and versions required to run the API. (pip install -r requirements.py)
- ***Features***: This folder contains fifteen *.feature* files that describes the user stories being tested. Each *.feature* file contains a normal flow, an alternative flow, and an error flow.
- ***runTodoManagerRestAPI-1.5.5.jar***: This file is a Java Archive (JAR) file that contains the executable code for running the Todo Manager Rest API. It is a packaged file with all the dependencies and libraries required to run the API. The version number (1.5.5) indicates the specific version of the API.

- ***starter.bat***: This batch script file can be executed on a Windows machine to start running the Todo Manager Rest API.
- ***test.bat, test_random.bat* and *test_server_down.bat:*** These three scripts run tests with behave library for cucumber test cases. The *test.bat* is a normal test script. The *test_server_down.bat* simulates the server is down and then executes the tests which the expected failed all. And the *test_random.bat* runs all tests scenarios in random order.

Behave Test Suit

The tests are designed to be run using the behave framework, a popular Behavior-driven development (or BDD) framework in Python. This exploratory testing folder contains a suite of user stories designed to test a REST API for a Todo Manager application (runTodoManagerRestAPI-1.5.5.jar). The folder structure is organized to separate user stories into different python scripts in the subfolder stories. In the steps/steps.py file, defined all the python functions we design to simulate the user's actions and results verification steps.

```
├── features
│   ├── __init__.py
│   ├── behave_random.py
│   ├── environment.py
│   ├── fixture.py
│   ├── steps
│   │   └── steps.py
│   └── stories
│       ├── add_task_to_course.feature
│       ├── adjust_priority.feature
│       ├── category_title.feature
│       ├── change_project_priority.feature
│       ├── change_todo_priority.feature
│       ├── create_todo_list.feature
│       ├── drop_add_new_project.feature
│       ├── fix_category.feature
│       ├── mark_task.feature
│       ├── query_incomplete_class.feature
│       ├── query_incomplete_high_priority.feature
│       ├── remove_task_from_course.feature
│       ├── remove_todo_list.feature
│       ├── task_description.feature
│       └── task_priority.feature
```

- ***behave_random.py:*** the script for running the behave BDD test suite. It overrides the regular runner to randomly shuffle the order of feature locations for the tests. The random order of tests can be useful to uncover any hidden dependencies between tests.
- ***environment.py:*** appears to be a behavioral-driven development (BDD) file. It initiates the behavior scenario by calling a fixture from the *features.fixture*, the "app".
- ***fixture.py:*** that provides a function 'app' (decorated with the @fixture decorator). What it does depends on the presence and value of the environment variable "ABNORMAL". If the variable "ABNORMAL" has value "true", it exits straight away. Otherwise, it launches a "starter.bat" batch file in a new process

and runs a curl command to access http://localhost:4567 until it is successful, and then yields control back to the caller, preserving the state of the function to resume later. After the yield, there is a curl command to shut down the local server and killing the process.

- ***steps/steps.py***: This file contains all steps used in the user stories. Functions that include the code implementation of the **Given, When Then** steps written in the user stories
- ***stories/*.features***: Files that gives a representation of the user stories using the cucumber's syntax with **Give, When, Then, And**. Each file contains the following 3 flows.
  - Normal Flow: A flow where the user story is according to the features design and produces no error.
  - Alternative flow: A flow that alternates from the Normal flow that still follows the features design.
  - Error flow: A flow that contradicts the design purpose of the feature but is a possible combination of operations from the clients' standpoint.

Findings

Story Test Suite Execution

In our exploration of utilizing multiple consistent tests to simulate a comprehensive user experience, we endeavored to implement this concept in Part A of our project. However, the nature of Part A required individual tests, necessitating the incorporation of a reloading method for each function within the Test Suite. This approach resulted in a fragmentation of the complete narrative into separate, independent test cases. Interestingly, our initial concepts were more effectively realized in Part B of the project.

Contrary to our expectations, the implementation process differed markedly. We had anticipated that the stories would be executed exclusively in multiple Python files. However, guidance from the provided instructions and input from teaching assistants revealed that employing the Cucumber framework and *.feature* files was a more efficient and comprehensive method for conducting story testing across a wide array of distinct scenarios. Consequently, our approach was structured into three distinct phases: initially, we created a table outlining general ideas; subsequently, these ideas were integrated into the Cucumber framework; finally, we developed functions corresponding to the stories in the *.feature* files. The first phase required considerable effort to conceive fifteen unique and non-repetitive test stories. The second phase, which involved expanding these fifteen ideas into detailed scenarios with diverse input examples and constructing various testing flows in *.feature* files via the Cucumber framework, was less challenging. The third phase involved translating these ideas into Python code, a process that was engaging yet demanding. It was critical to ensure consistency between the function titles in the code and the specific actions in the *.feature* files, as any minor discrepancy could lead to errors, adding complexity to the process. We encountered difficulties in aligning the names between functions and those in the *.feature* files. To address this, we consolidated all functions into a single Python file named *steps.py*, enhancing the efficiency and convenience of searching for step functions and reusing some of them. Despite these challenges, we successfully navigated through them, resulting in a fully functional and effective outcome.

This comprehensive journey significantly enhanced our understanding of constructing story tests and effectively organizing the testing process. We gained proficiency in transitioning from abstract concepts to specific implementations and learned to adeptly use the keywords 'Given' and 'And then' to craft story tests in various

sequences. A notable discovery was the ability of *.feature* files to activate functions in Python files, facilitating diverse testing sequences and inputs to create distinct testing flows. This approach markedly increased testing efficiency and time savings, enabling us to manipulate testing functions in varied sequences using *.feature* files rather than exhaustively scripting all desired testing narratives, which would have involved substantial repetition of code and functions.