ECSE 429 Part A Report

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

Introduction

This report provides a comprehensive overview of the exploratory testing conducted on the 'REST API Todo List Manager.' This application is designed to manage to-do lists, allowing for operations on various entities such as tasks, categories, and projects. Our testing focused on identifying the capabilities and potential areas of instability within the application, covering documented and undocumented functionalities.

Deliverables

Session Notes: We first used 45 min with three teammates to finish our manual exploratory tests with Postman and an online XML editor. The session notes, screenshots of Postmans and our synchronized Google scrap sheet are also attached to this submission.

Bug Report: The bug report summarized all the bugs we explored during this exploratory test session. The bug report is named *Bug Summary* with this submission.

Concerns: during this session, we found the following concerns, and we will further test these concerns with detailed and complete unit test cases with the pytest framework.

Todos:

I. Inconsistent support for XML format bodies.
II. Returns the same list of projects when using corresponding get messages. Indicating flawed implementation of adding connection and getting connections.
    1. /todos/100/categories
III. Not found error does not contain error message for some GET, POST, and Delete messages. Need to further test for the error message
    1. Delete /todos/2/projects 404 but no error message
    2. Get /todos/2/projects 404 but no error message
    3. Post /todos/2/projects 404 but no error message

Projects:

I. Inconsistent support for XML format bodies.
II. Returns the same list of categories when using corresponding get messages. Indicating flawed implementation of adding connection and getting connections.
    1./projects/100/categories returning empty categories list

Categories:

I. Inconsistent support for XML format bodies.
II. Returns the same list of todos and projects when using corresponding get messages. Indicating flawed implementation of adding connection and getting connections.
III. Not found error does not contain error message for some GET, POST, and Delete messages. Need to further test for the error message
    1. Delete /categories/2/projects 404 but no error message

2. Get /categories/2/projects 404 but no error message
3. Post /categories/2/projects 404 but no error message
4. Delete /categories/2/tasks 404 but no error message
5. Get /categories/2/tasks 404 but no error message
6. Post /categories/2/tasks 404 but no error message

General:

For Project and Category classes, the connected todo objects are accessed under the URL name of tasks. It's a confusing naming scheme and can be confusing.

New Test Ideas: During the session, we had limited time to explore the application in more detail. Based on the content of Test Ideas in the lecture. We propose the following test ideas and splitted into 4 main categories:

Capabilities (is part will be solved by User Story test cases):

1. What if a user attempts to create a todo with the same name as an existing todo? The system should prevent this to avoid confusion.
2. What if a user attempts to create a project without any assigned todos? The system should allow this as projects may not always have todos during their creation.

For Failure Modes, all our new ideas that are out of the API documentation have already been tested in the session notes (in our "undocumented tests").

Quality Factors:

1. Can the system handle special characters or letters in other languages in todos, project or category names? The system should handle them correctly or prevent them with appropriate error messages.
2. Check the system whether it can not manage extreme scenarios? For instance, we need to assess the system's maximum capacity for continuous object creation (via POST requests),git and determine the maximum allowable lengths for both the 'name' and 'description' fields.

Usage Scenarios:

1. Imagine a situation where a user wants to update the name of a todo. The system should allow this and reflect the changes accurately in all related projects and categories.
2. Imagine a user wants to delete a project. The system should handle this by removing all associated connections with todos and categories.

Test cases based on user behavior. The unit test can only test whether the single function/endpoint works appropriately. The combinations of several functions and endpoints are unverified.

Automation Unit Test: During our test, we followed the suggested unit testing ideas, and additional test ideas implemented the following:

1. We summarized the errored unit tests during regular unit tests and modified the expected results to the wrong ones.
2. We tested every status code of every response, including undocumented methods, such as PATH, POST, and GET, in specific endpoints.
3. We test almost all endpoints with json/XML payload and json/XML response by changing each request's header and data body.
4. We also construct some cases with mismatched headers and data bodies to test whether the API endpoints work appropriately.

5. Moreover, to ensure the independence of each test case, we employ the *pytest-random-order* plugin to ensure an unordered sequence during the test. We also use a function for writing numbers to verify the local files.
6. In the video, all the test cases will fail when we stop the server.
7. With the concerned points during the test session, we use invalid operations to test the endpoint further to ensure the correctness of their functionalities.

Video: For the video of the live demonstration, the video (*Video Demo.mkv*) shows the execution of unit tests in random order. The execution order of *test_project.py* is written in a file and shown. The specific failing cases have been commented to be a BUG from the api and shown as well. The video (*Video Demo.mkv*) shows the execution twice, and it can be seen the order of execution is different but the result remains the same. Also demonstrate when the server is down (*Video Demo Abnormal.mp4*) , the all test caeses fail.

Code Repository

The following is the folder structure of our repository:

```
ECSE 429 Project Code Repo
├── README.md
├── starter.bat
├── Todo-Manager-swagger.json
├── abnormal.ini
├── requirements.txt
├── runTodoManagerRestAPI-1.5.5.jar
├── test.bat
├── test_server_down.bat
└── tests (all test cases of all python files)
```

- ***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)
- ***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* and *test_server_down.bat (abnormal.ini)***: These two scripts run tests with pytest in random order. The test.bat is a normal test script, expected to find the failed cases in **`test_categories/todos/projects.py`** and not errors in **`test_unexpected.py`**. Also 4 successful and 4 failed test cases in **`test_additional.py`**. The test_server_down.bat simulates

the server is down and then executes the tests which the expected failed all. **`abnormal.ini`** this envriontment configuration is setting environment variable “ABNORMAL” to true, enabling the server down.

Unit Test Suit

The tests are designed to be run using the pytest framework, a popular testing framework in Python. This pytest exploratory testing folder contains a suite of tests designed to test a REST API for a Todo Manager application (runTodoManagerRestAPI-1.5.5.jar). The folder structure is organized to separate tests into different categories. For example, the 'todos', 'categories', and 'projects' subfolders contain tests specific to each application component.

```
└── tests
    ├── __init__.py
    ├── categories
    │   ├── __init__.py
    │   ├── category_test_data.py
    │   └── test_categories.py
    ├── config.py
    ├── conftest.py
    ├── projects
    │   ├── __init__.py
    │   ├── project_test_data.py
    │   └── test_projects.py
    ├── supplement
    │   ├── __init__.py
    │   ├── additional_test_data.py
    │   ├── test_additional_test.py
    │   └── test_unexpected.py
    ├── todos
    │   ├── __init__.py
    │   ├── test_todos.py
    │   └── todos_test_data.py
    └── utils.py
```

- ***config.py***: This file contains the base URLs for the different endpoints of the REST API that are used throughout the tests.
- ***conftest.py***: This file contains fixtures (similar like Bean in Springboot) that are used across multiple test cases. Specifically, it includes setup and teardown procedures that are run before and after the entire test suite and individual tests.
  - *client()*: This fixture is run before and after each test case. It starts the API server, waits until it is ready to accept requests, and then shuts it down after the test.
- ***utils.py***: This utility file contains helper methods for making HTTP requests to the REST API and processing the responses.
- ***categories, projects, and todos*** subfolders: These folders contain the actual test cases for the different components of the application. Each test file contains multiple test cases, sending a specific request to the API and verifying the response.
  Since we separate three folders for three people for each test file, the order of test cases in each test case is different. For test_todos.py and test_categories.py, the test cases are followed through the endpoints, for example, /todos, to /todos/id, /todos/id/categories; for test_projects.py, the test cases are separated by payload format and edge cases, the first part is edge cases for each function, the second part is the json payload, the third and last part is the XML payload test cases.

- ***supplement***: This fold includes the additional test cases. The ***test_additional_test.py*** includes the 4 good JSON and XML payloads and *4 corresponding malformed JSON and XML payloads*. The ***test_unexpected.py*** includes the module showing the *actual behaviour working of all failed test cases from categories, projects, and todos* 3 folders.

Pytest Improvment Idea:

1. Use python logger and pytest-logger extension for debugging
2. Async application, would be fast if own the application in python

Findings

Exploratory Test

Before initiating this project, our familiarity with RestAPI and its associated components was limited, leading to uncertainties in implementing exploratory tests for the "runTodoManagerRestAPI." Compounding this challenge was the inadequacy of details on the runTodoManagerRestAPI's webpage, rendering the delineation of appropriate test parameters somewhat elusive.

Despite these initial setbacks, our dedicated week of rigorous investigation led to two pivotal discoveries that clarified our ambiguities. Our first revelation was a comprehensive manual located within the API Documentation. This resource proved invaluable, outlining specific commands applicable to the runTodoManagerRestAPI and significantly directing our testing approach. Furthermore, we adopted the use of Postman, as recommended by the teaching assistant, and were quickly able to discern its substantial efficacy in streamlining the testing process.

Leveraging these instrumental tools, we meticulously examined the End Points delineated in the API Documentation using Postman, facilitating a profound comprehension of the runTodoManagerRestAPI's operations. This hands-on approach also endowed us with invaluable experience in formally and exhaustively testing web-based software applications. Interestingly, the systematic testing of End Points across various sections (namely, todos, projects, and categories) did not fully utilize the allocated 45-minute testing duration. This surplus of time allowed for the incorporation of additional tests that extended beyond the essential End Points, including specific undocumented tests derived from our anticipations based on CRUD (Create, Read, Update, Delete) operations and minor variations in input syntaxes.

Regarding the documentation of our sessions, our prior experience with Freemind significantly streamlined this process. The lessons learned previously were instrumental in enhancing both the structural and grammatical quality of our session notes for the runTodoManagerRestAPI. Notably, our proficiency in recording these sessions markedly improved, underscoring the benefits of accumulated experience.

In conclusion, while the journey was laden with challenges, our persistent efforts were rewarded with the surmounting of initial hurdles, the discovery of innovative and beneficial resources, and the accrual of extensive experience. This bolstered our confidence in conducting thorough and formal testing. Our commitment and relentless pursuit of knowledge and expertise culminated in the successful completion of this assignment, underscoring the importance of passion and diligence in overcoming obstacles.

Unit Test Suite Execution

The commencement of our Unit Test Suite Execution phase was marked by deliberation and introspection, as our team grappled with a seemingly fundamental yet intricate choice: selecting between Java and Python as the primary language for our suite. This quandary was a testament to our team's diverse skill set. On one hand, our prowess in utilizing JUnit within the Java ecosystem made it a strong contender. On the other, our extensive engagements with APIs using Python painted a compelling case for its adoption.

Our exploratory testing phase proved instrumental in guiding this decision. The challenges we encountered during this phase illuminated the complexities of programmatically interfacing with End Points, essentially replicating the capabilities provided by tools like Postman. We realized that this task, when juxtaposed against crafting the Test Suite based on our session notes, presented a more formidable challenge. This revelation tilted the balance in favor of Python, recognizing its versatility and our team's collective experience with it.

Diving into the Python-driven implementation, we were immediately confronted with the task of mastering test block constructions in a language environment that, while familiar, presented unique challenges in a unit testing context. But this was just the tip of the iceberg.

Our journey led us to the intricate world of data transmission, particularly with JSON and XML files. Ensuring the accuracy of data being sent and received was paramount. This involved not only a deep understanding of these data structures but also the implementation of rigorous assertion checks, making certain that the data remained incorrupt and true to its intended format.

Yet another layer of complexity was introduced when considering the interdependence of test blocks. In an ideal testing environment, each test operates in isolation, ensuring that one test's outcome doesn't inadvertently influence another. Achieving this necessitated frequent server restarts before each test. While operationally demanding, this approach was crucial in eliminating potential data residues or lingering state conditions that could skew test results.

Reflecting on our journey, it was evident that the Unit Test Suite Execution phase was far from a linear process. Each challenge encountered compelled us to adapt, think critically, and innovate. The experience not only enriched our technical toolkit but also fostered a deeper appreciation for the intricacies of unit testing, particularly in the realm of APIs. As we navigated each challenge, from language selection to data integrity, the journey underscored the ever-evolving nature of technology and the importance of continuous learning and adaptation in our rapidly advancing field.