Author: William Wang
Student ID: 261052682
email: william.wang3@mail.mcgill.ca

ECSE 429 Project Part B: Report

Due November 5th 2024

Deliverables Summary

User Stories

The user stories were designed around the **/todos** of the Thingifier Rest API, with a total of five different user stories revolving around different use cases a user may have with the API. Each user story is comprised of three different tests for the following scenarios: **Normal flow, Alternative flow and Error flow**. All of the user stories are .feature files stored inside the file path ~/PartB/User_stories/features starting from the home repository.

User Stories Test Automations

Environment

A file called **environment.py** can be found in the **~/PartB/User_stories/features** file path. This python file specifies testing configurations so that before any tests run, a **before_all()** function is called to verify whether the Rest API is currently running. If it is running, the rest of the tests proceed to execute. However, if the API is not running, the function immediately raises an error and exits. This allows for a gracious termination of the automated tests, instead of having each scenario test time out. Then, another function called **after_scenario()** is called after each scenario test is concluded, where it resets the todo list to its original state. This ensures that there are no lingering todo instances that might intervene with other tests that rely on assumptions such as starting the scenario with only the default todo instances. This also allows the API to return to its original state after all the story tests conclude. The environment also contains the function before_feature() which is implemented to randomize the order in which scenarios are tested

Additional Python File

An additional python file called **run_random_features.py** is implemented to randomize the order in which features are executed during the automated testing. This ensure that alongside the environment configuration of scenario randomization, the entirety of the automated tests are executed in random order.

Todos

Each python test file is stored in the **~/PartB/User_stories/features/steps** filepath

todo_user_story_1.feature

It contains the Gherkin Scenarios for the feature **Create a new todo instance** where the scenarios are designed around cases where a user may want to add a new todo instance to the todo list.

todo_story1_create_steps.py

The test script starts off by verifying whether the API is running, since the Background for the corresponding Gherkin test assumes that the Thingifier Rest API is running. Then, it verifies that the todo list starts off containing only the two default todo instances to ensure that the test begins at a correct initial state. Only this test script contains as each test scenario has the same background.

Then, the next test corresponds to the normal flow test by attempting to add a new todo instance by specifying the title, the doneStatus and the description. For this scenario, the return status code is asserted and an assertion is made to ensure that a new todo instance is added to the list and that all of its fields have the correct values.

For the alternating flow, the user attempts to add a new todo instance by only specifying the title and leaves the rest of the fields to be filled with default values by the API. For this scenario, the return status code is asserted and an assertion is made to ensure that a new todo instance is added to the list and that all of its fields have the correct values.

Afterwards, an error flow test is run by attempting to add a new todo instance without specifying the title. For this scenario, the return status code is asserted and an assertion is made to ensure that no changes are made on the todo list.

todo_user_story_2.feature

The second user story revolves around the feature **Delete a todo instance** for different scenarios where a user may want to try and delete a todo instance from the list.

todo_story2_delete_steps.py

The normal flow test verifies whether a user can delete an existing todo instance in the API list by specifying the correct **id** value. For this scenario, the return status code is asserted to ensure that there is not an error and an assertion is made to ensure that

the todo list no longer contains the deleted todo.

For the alternating flow, the test verifies whether the user is allowed to delete the same todo after it has already been deleted by sending out two delete requests in succession with the same **id** value. For this scenario, the return status code is asserted to ensure that there is indeed an error thrown by the API and that the second delete request does not bring any changes to the todo list.

For the error flow, the test verifies whether it is allowed for a user to send a delete request by specifying a non integer **id** value. For this scenario, an assertion is made to ensure that a non integer **id** value is illegal and an assertion is made to ensure that no changes were made in the todo list.

todo_user_story_3.feature
The third user story revolves around the feature **Get an existing todo instance** for the scenarios where a user might want to retrieve information regarding a todo instance that is in the API's list.

todo_story3_get_steps.py
The normal flow test verifies whether a user can retrieve information about a todo instance that exists in the API list by specifying a valid **id** value. For this scenario, an assertion is made on the return status code to ensure that no error is thrown by the API and another assertion is made to ensure that the API returns the correct todo instance with correct values.

The alternating flow test verifies whether a user can request to get information about a todo instance that exists in the API list by specifying a valid **id** value and by specifying the format to be XML. For this scenario, an assertion is made to verify that no unexpected error is thrown by the API and an assertion is made to ensure that the API not only returns a todo instance in XML format, but also whether the todo information is correct.

The error flow test verifies whether a user can get information about a non existent todo instance by specifying an **id** value that is not contained within the API's todo list. For this scenario, the test asserts the return status code to see if the API correctly throws an error and that there is no todo instance returned to the user.

todo_user_story_4.feature
The fourth user story is designed around the feature **Amend a todo instance** for different scenarios where a user may want to update information about a todo instance that was added to the API's list
todo_story4_amend_steps.py
The normal flow test verifies whether a use can request to update the title, doneStatus and description of an existing todo instance that has an **id** value matching the one

provided by the user. The request is done with a JSON body containing the updated values. The test asserts the return status code to ensure that no error is thrown by the API. An assertion is also made to ensure that the affected todo instance has its title, doneStatus and description values updated as intended.

The alternate flow test verifies whether a user can request to updated the title, doneStatus and description of an existing todo by providing an existing **id** value and updated field values in XML format. The scenario asserts the returns status code to ensure that the API does not throw an unexpected error and another assertion is done to ensure that the todo instance has its values appropriately updated.

The error flow test verifies whether the user can request to update an existing todo instance in the API's list by providing a valid **id** value but with a malformed JSON body in the request. For this scenario, the test asserts the return status code from the API to ensure that an error is indeed thrown due to the malformed JSON body. An assertion is also made on the todo instance's values to ensure that the erroneous request does not update the todo instance.

todo_user_story_5.feature

The fifth user story is designed around the feature **Get all existing todo instances** for scenarios where a user may want to get a collection of todo instances with one request.
todo_story5_getAll_steps.py

The normal flow test verifies whether the user can send a GET request on all of the existing todos in the API's list. For this scenario, the test asserts the return status code to ensure that there is no unexpected error thrown by the API and another assertion is done on the collection of todos received by the client. This second assertion ensures that the collection of todos includes all of the existing todos with the correct information for each of them.

The alternate flow test verifies whether the user can send a request on all of the existing todos that have their doneStatus as **False** in XML format. For this scenario, the test asserts the return status code to verify that there is no error thrown by the API. An assertion is also made on the collection of todos received by the user to ensure that the XML format is correct and that the API returns the correct todo instances.

The error flow test verifies whether the user can send a GET request on all of the existing todo instances in the API's list by specifying a non-existent field name and value as filter. For this scenario, the test asserts the return status code to ensure that the API returns an error appropriately. The test also asserts whether the API returns a collection of todo instances to the user, where the expected behavior is that the API

should not return any todo instances.

Bug Summary Form

The Bug summary form was written in an Excel sheet file named **Bug_Report.xlsx** in the **PartB** folder. The form contains a collection of all the bugs that were found during the exploratory testing. For each bug, the form outlines the description of the bug, which user story is related to the bug, instructions on how to reproduce the bug and impacts that the bug can bring to the user.

Unit Test Video

The user test video demonstrates the initial Thingifier *.jar* file execution to ensure that the API is running during testing. Then, the video shows the initial state of the API's todo list before any testing. After showing the initial state, the video demonstrates how to run the automated story tests using the **Behave** Python library by executing the **run_random_features.py** file. After all the tests terminate, the video shows the command-line output from the automated tests and shows that the API has its original state restored properly. Then, the video shows that running the automated tests once more yields the same results but the tests are in a different order due to randomization. The video is stored as an MP4 file in the **PartB** folder.

Written Report

This written report provides a description of all the deliverables for the project and describes the findings of unit test suite execution. The written report can be found as a PDF file in the **PartB** folder.

Source Code Repository

The source code repository for this project is uploaded as a GitHub repository, which is made accessible with the following link:

https://github.com/williamwang1382/ECSE429_Project

The Source Code Repository contains not only the source code for all the user story test scripts used during the story testing , but it also contains all the Gherkin Scenarios and the bug form documents as well. As explained above, the video demo of a user running the tests can also be found in the repository.

Unit Test Suite Execution Findings

The majority of the tests conducted lead to expected behaviors, where user stories 1, 2, 3 and 4 all had results that were expected. However, for user story 5, the error flow test failed. The test verifies whether the user can request a GET on all todo instances using a non-existent field name and value as a filter, the test exposed a bug. Since the request uses an invalid field name, the use should expect the API to raise an error, indicating that the request is invalid. However, the API instead returned a status code of 200, indicating that the API did not consider the bad request illegal. Furthermore, since the request uses a non-existent field, the expected behavior from the API would be to not return any todo instances, since none of them contain a value tied to the erroneous field name. However, the API instead returned all of the existing todo instances without any filter. This bug is further described in the Bug summary form, where the impact of the bug on the user is explained.