ECSE 429 – Part B

Story Testing of Rest API

Naomie Lo | naomie.lo@mail.mcgill.ca | 261018690

Deniz Emre | deniz.emre@mail.mcgill.ca | 261029931

Deliverables overview

In this part of the assignment, we dug deep to what a typical user would do when approaching Thingifier. For todos and projects, we have brainstormed 5 user stories each. The stories consist of normal, alternate, and error flow and defined using gherkin scripts. For each capability, there is an environment and a step definition script. This is to ensure the environment has its initial conditions maintained and the step definitions will test the flows in the user stories.

Regarding our tool to parse Gherkin scripts, we chose Behave. This tool, like Cucumber, works well with Gherkin's syntax. Behave's integration with Python enables us to run the stories from our IDEs.

Below is the breakdown of our deliverables:

Report

Readme

- Videos of our tests are located here

Todos

1. steps > step_definitions.py
    - Steps for the gherkin scripts using library "Behave"
2. environment.py
    - To capture initial environment and restore it
3. delete_todos.feature
    - Test the DELETE for todos instances
4. get_taskof.feature
    - Test the GET for tasksof instances
5. get_todos.feature
    - Test the GET for todos instances
6. post_categories.feature
    - Test the POST for categories instances
7. put_todos.feature
    - Test the PUT for todos instances
8. random_behave_runner.py

- script to run tests in random ordering

9. Video showing the tests running

Projects

1. steps > step_definitions.py
    - Steps for the gherkin scripts using library “Behave”
2. environment.py
    - To capture initial environment and restore
3. delete_projects.feature
    - Test the DELETE for projects instances
4. get_projects.feature
    - Test the GET for projects instances
5. get_categories.feature
    - Test the GET for categories instances
6. post_categories.feature
    - Test the POST for categories instances
7. put_projects.feature
    - Test the PUT for categories instances
8. random_behave_runner.py
    - script to run tests in random ordering
9. Video showing the tests running

Structure of Story Test Suite

Each story has 5 components: description of feature, background, normal flow, alternative flow and error flow.

The description outlines what exactly the user wants to test.

```
Feature: Update Todo
  As a user, I want to retrieve the headers for all the categories linked to a specific todo so that I can quickly
  assess the details of these categories without loading the full content.

  Background:
    Given the API is responsive
    And there is an existing todo with title 'Workout' in the database

  # Normal Flow
  Scenario: Successfully retrieve project items (tasksof) for a todo
    Given the todo with the title 'Workout' already has task items 'Morning Run' and 'Evening Stretch'
    When the user makes a GET request to /todos/:id/tasksof for the todo with the title 'Workout'
    Then the status code 200 will be received
    And the response of the todo with title 'Workout' will include the project with title 'Morning Run' and 'Evening Stretch'

  # Alternate Flow
  Scenario: Retrieve project items for a todo with no tasksof
    Given the todo with the title 'Workout' has no tasks instances
    When the user makes a GET request to /todos/:id/tasksof for the todo with the title 'Workout'
    Then the status code 200 will be received
    And the response contains an empty list for 'projects'


  # Error Flow
  Scenario: Fail to retrieve project items for a non-existent todo
    When the user attempts to retrieve a taskof with a todo with id 10987654321
    Then the status code 200 will be received
    And the response contains a non empty list for 'projects'
```

Feature Description: Each feature begins with a concise description, specifying the user's goal in using the feature. This defines the context and scope of the tests, ensuring clarity about the functionality being verified.

Background Setup: Background steps set up preconditions necessary for the test, such as ensuring the API is responsive and that specific entities like todos or projects exist in the database. This ensures that each scenario has a consistent starting point, eliminating setup redundancies across scenarios.

Normal Flow: The normal flow scenario captures the expected behavior of the feature under typical conditions. It tests whether the feature works as intended when all preconditions are met, allowing us to validate that the basic functionality aligns with user expectations.

Alternate Flow: Alternate flow scenarios explore variations in typical usage, such as handling empty or unusual data cases. This is essential for checking the flexibility of the API. For example, we test scenarios where a project exists but has no categories, ensuring the API handles these cases correcty.

Error Flow: Error flow scenarios test the feature's response to invalid or unexpected inputs. These scenarios check the API's ability to handle errors, such as attempting to delete a non-existent project or retrieve categories for a project

that does not exist. This flow is particularly useful for validating appropriate error codes and messages, enhancing the reliability of the API in real-world usage.

Environment Setup and Restoration: The environment script (environment.py) is used to set up and reset the environment before and after each test scenario. This ensures data consistency across tests, particularly for scenarios that involve modifying data (e.g., deleting or updating entities). By restoring the initial state, we prevent test contamination and ensure that each scenario is tested independently of previous tests.

Gherkin Steps Implementation: The step definitions (step_definitions.py) implement the Gherkin steps using Behave. They interact with the API to perform CRUD operations, retrieve necessary data, and make assertions. Helper functions such as get_todo_id(title) and get_project_id(title) dynamically retrieve entity IDs, which is essential due to the API's behavior of assigning new IDs upon entity recreation. This setup minimizes hard-coded dependencies and makes the tests resilient to changes in data.

Source Code Repository

- ➔ Report.pdf
- ➔ README.MD
  - Videos for todos and projects
- ➔ PART B
  - ◆ todos
    - features
      - steps
        - ◆ step_deinitions.py
      - environment.py
      - delete_todos.feature
      - get_taskof.feature
      - get_todos.feature
      - post_categories.feature
      - put_todos.feature
      - random_behave_runner.py
  - ◆ projects

- features
    - steps
        - step_deinitions.py
    - environment.py
    - delete_projects.feature
    - get_categories.feature
    - get_projects.feature
    - post_categories.feature
    - put_projects.feature
    - random_behave_runner.py

Findings of Story test suite

Todos

Environment Reset Results in ID Changes
When resetting the environment (e.g., deleting and recreating todos, projects, or categories), the IDs of these entities change. This is because the API assigns new IDs each time an entity is recreated. I was unable to create an ID during creation without causing errors.

Unexpected Behavior with Non-Existent Todos
A GET request to /todos/:id/tasksof for a non-existent todo returns a response with a 200 OK status code instead of a 404 error. This is unexpected because the todo does not exist. The expected behavior would be to return a 404 Not Found response, indicating that the resource is missing. The fact that something is returned with a 200 OK code is unusual and may be a bug in the API. In return I got the tasks for the other todo instances. Another finding is that the GET request returns a project instance. As a team of two we did not explore relationships between instances, but logically this was confusing.

ID Handling
Since IDs change upon recreation of entities (todos, projects, categories), any test that relies on fixed IDs will need to dynamically retrieve the new IDs after the entities are recreated. This is why helper functions like get_todo_id(title) are necessary.

Consistency

The todos API mostly performed as expected during typical Post, Get, Put, and Delete operations as seen in the exploratory and unit tests. The POST /todos operation correctly allowed new project creation without requiring an ID, automatically assigning the next available ID. The DELETE /todos/:id operation worked as expected, successfully removing the resource as long as the instance existed.

Projects

Similar to todos, resetting the environment for projects (e.g., deleting and recreating projects or categories) changes the IDs of these entities. The API generates a new ID each time an entity is recreated, which requires us to dynamically fetch IDs for each test to ensure consistency. We implemented helper functions, such as get_project_id(title), to retrieve the correct project ID based on the title, ensuring that tests always interacted with the intended entities.

Consistency: The projects API mostly performed as expected during standard CRUD operations (Post, Get, Put, and Delete) as observed in the exploratory and unit tests. The **POST /projects** operation correctly allowed new project creation without requiring an explicit ID, automatically assigning the next available ID, which streamlined the creation process. The **DELETE /projects/** operation also behaved as expected.

Inconsistency: In our testing of the Retrieve Categories of a Project feature, we encountered an inconsistency when attempting to retrieve categories for a non-existent project using a numerical ID. The expected behavior was to receive 404 Not Found status code, indicating that the project does not exist in the database. However, our test returned a 200 OK status instead. This result contradicts the intended response, as a successful retrieval (200 OK) should only occur when accessing valid, existing resources.

This inconsistency was initially identified as a potential bug in Part A, where it was observed that the API sometimes returns 200 OK responses even when querying non-existent resources. In this case, rather than returning a 404 Not Found error, the API behaves as if the non-existent project exists, but without any associated data. This behavior is misleading to users who expect an error response when attempting to retrieve data for an invalid resource.

The skipped step in our test output (And the error message will be empty) further supports the presence of this bug. Since the API returned 200 OK instead of 404, subsequent checks for an error message were skipped, as Behave does not execute steps following a failed assertion. This reinforces the need for correction in the API response handling for non-existent resources, ensuring that appropriate error codes are consistently used to communicate resource availability to the user. This issue, identified in Part A and confirmed in our tests here, highlights an area where API response standards should be reinforced.