ECSE 429: Software Validation

Project Part B: Written Report

Group 7

March 18th, 2025

Ryan McGregor – 260868511 – ryan.mcgregor@mail.mcgill.ca

Brenden Trudeau – 260941695 – brenden.trudeau@mail.mcgill.ca

Emmy Song - 261049871- emmy.song@mail.mcgill.ca

Summary of Deliverables

1. Story Test Suite

Our team defined 15 user stories focusing on todos, projects, and categories, each with three tests covering:

- Normal Flow (expected behavior)
- Alternative Flow (edge cases)
- Error Flow (handling invalid inputs)

We implemented 15 feature files (one per user story) and 15 step definition files, with a helpers.py file for reusable functions.

2. Story Test Automations

We used Behave to automate story tests, structured as:

- features/ (Gherkin test scripts)
- features/steps/ (step definitions)

Each test restores the system state, and the API must be restarted before re-running tests. Tests are executed by running *behave*

3. Video

A recorded demo shows story tests running.

4. Bug summaries

Bugs found were documented in bug_summary_B.md, detailing description, impact, and reproduction steps**.**

5. Written report

This document

Story Test Suit Structure

For Part B of our project, we structured our test suite within the SOFTWARE-VALIDATION-PROJECT directory, specifically inside the B folder. This directory contains essential files such as bug_summary_B.md, where we document the bugs we identified, and README.md, which provides instructions on running the test suite and lists the necessary dependencies. Additionally, get-pip.py is included for setting up the required Python environment.

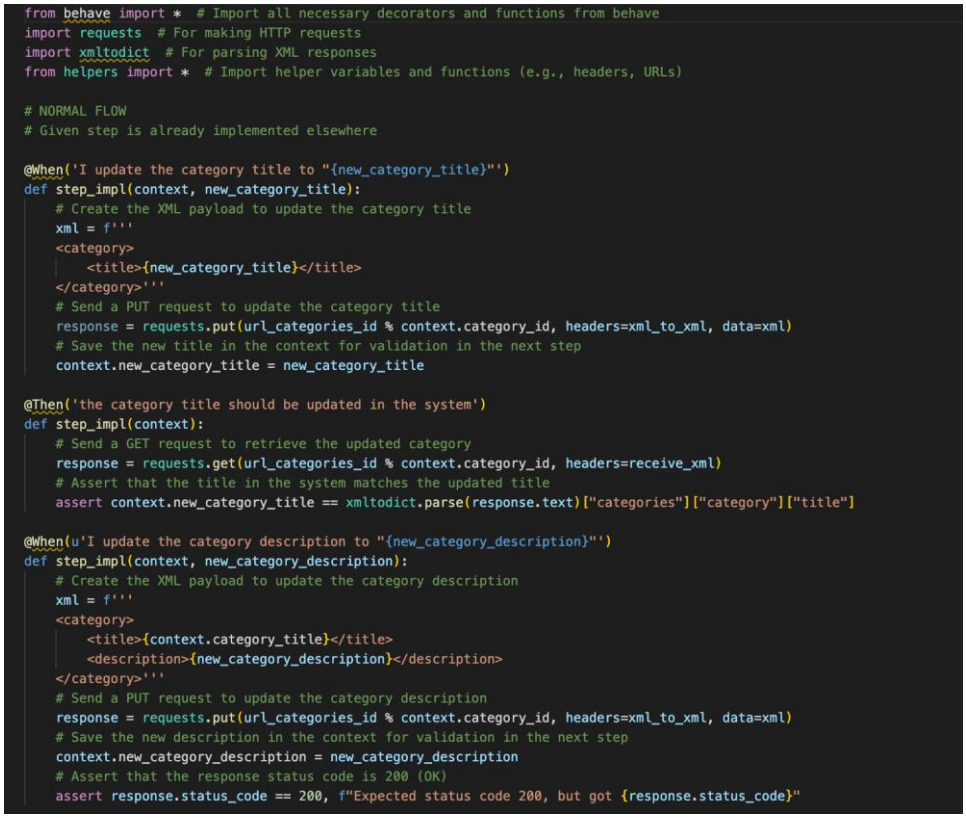
Our test suite is structured into two main directories:

- features/: Contains 15 feature files (5 for todos, 5 for projects, and 5 for categories). Each feature file follows a structured Gherkin format to define user stories and test scenarios. An example of a feature file can be seen in Figure 1 below.

```
Feature: Create a new project
    As a user,
    I want to create a new project in /projects
    so that I can organize my tasks efficiently.

    # Normal Flow
    Scenario Outline: Successfully create a new project
        Given there is no existing project with title "<project_title>" and with description "<project_description>"
        When I create a project with title "<project_title>" and with description "<project_description>"
        Then the project with title "<project_title>" and description "<project_description>" should be saved in the system
        Examples:
            | project_title         | project_description                    |
            | ECSE 429 Final Report | Complete and submit report for ECSE 429 |
            | Home Renovation       | Plan and execute home renovation tasks  |
            | Fitness Program       | Organize weekly workout routine         |

    # Alternative Flow
    Scenario Outline: Create a project with no description
        Given there is no existing project with title "<project_title>"
        When I create a project with title "<project_title>" but with no description
        Then the project with title "<project_title>" should still be created successfully
        Examples:
            | project_title    |
            | Startup Business |
            | Reading List     |

    # Error Flow
    Scenario Outline: Attempt to create a project without a title
        When I attempt to add a project without a title
        Then the project should be created with an empty title
        Examples:
            | project_title |
            |               |
```

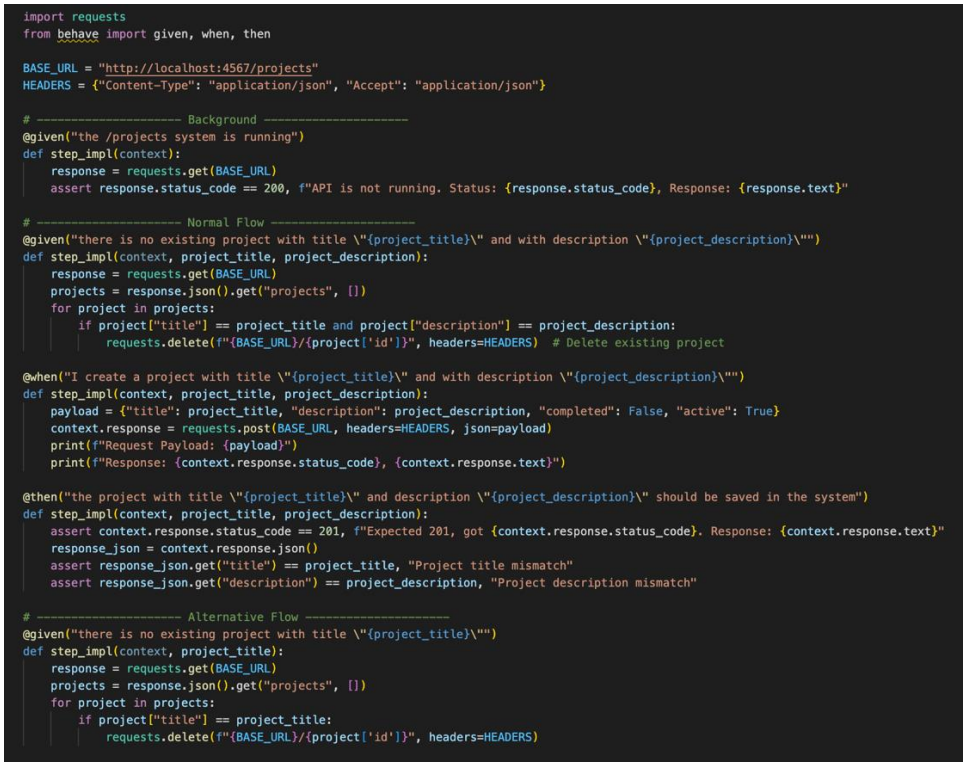
Figure 1: Example of a feature file (create_project.feature)

- features/steps/: Contains 15 corresponding step definition files, implementing test logic in Python with Behave. Two members used JSON, while one worked with XML. Example step files are shown in Figure 2 and Figure 3.



```python
from behave import *  # Import all necessary decorators and functions from behave
import requests  # For making HTTP requests
import xmltodict  # For parsing XML responses
from helpers import *  # Import helper variables and functions (e.g., headers, URLs)

# NORMAL FLOW
# Given step is already implemented elsewhere

@When('I update the category title to "{new_category_title}"')
def step_impl(context, new_category_title):
    # Create the XML payload to update the category title
    xml = f'''
    <category>
        <title>{new_category_title}</title>
    </category>'''
    # Send a PUT request to update the category title
    response = requests.put(url_categories_id % context.category_id, headers=xml_to_xml, data=xml)
    # Save the new title in the context for validation in the next step
    context.new_category_title = new_category_title

@Then('the category title should be updated in the system')
def step_impl(context):
    # Send a GET request to retrieve the updated category
    response = requests.get(url_categories_id % context.category_id, headers=receive_xml)
    # Assert that the title in the system matches the updated title
    assert context.new_category_title == xmltodict.parse(response.text)["categories"]["category"]["title"]

@When(u'I update the category description to "{new_category_description}"')
def step_impl(context, new_category_description):
    # Create the XML payload to update the category description
    xml = f'''
    <category>
        <title>{context.category_title}</title>
        <description>{new_category_description}</description>
    </category>'''
    # Send a PUT request to update the category description
    response = requests.put(url_categories_id % context.category_id, headers=xml_to_xml, data=xml)
    # Save the new description in the context for validation in the next step
    context.new_category_description = new_category_description
    # Assert that the response status code is 200 (OK)
    assert response.status_code == 200, f"Expected status code 200, but got {response.status_code}"
```

Figure 2: Example of a step definition file XML handling (test_update_category_title.py)



```python
import requests
from behave import given, when, then

BASE_URL = "http://localhost:4567/projects"
HEADERS = {"Content-Type": "application/json", "Accept": "application/json"}

# ---------------------- Background ----------------------
@given("the /projects system is running")
def step_impl(context):
    response = requests.get(BASE_URL)
    assert response.status_code == 200, f"API is not running. Status: {response.status_code}, Response: {response.text}"

# ---------------------- Normal Flow ----------------------
@given("there is no existing project with title \"{project_title}\" and with description \"{project_description}\"")
def step_impl(context, project_title, project_description):
    response = requests.get(BASE_URL)
    projects = response.json().get("projects", [])
    for project in projects:
        if project["title"] == project_title and project["description"] == project_description:
            requests.delete(f"{BASE_URL}/{project['id']}", headers=HEADERS)  # Delete existing project

@when("I create a project with title \"{project_title}\" and with description \"{project_description}\"")
def step_impl(context, project_title, project_description):
    payload = {"title": project_title, "description": project_description, "completed": False, "active": True}
    context.response = requests.post(BASE_URL, headers=HEADERS, json=payload)
    print(f"Request Payload: {payload}")
    print(f"Response: {context.response.status_code}, {context.response.text}")

@then("the project with title \"{project_title}\" and description \"{project_description}\" should be saved in the system")
def step_impl(context, project_title, project_description):
    assert context.response.status_code == 201, f"Expected 201, got {context.response.status_code}. Response: {context.response.text}"
    response_json = context.response.json()
    assert response_json.get("title") == project_title, "Project title mismatch"
    assert response_json.get("description") == project_description, "Project description mismatch"

# ---------------------- Alternative Flow ----------------------
@given("there is no existing project with title \"{project_title}\"")
def step_impl(context, project_title):
    response = requests.get(BASE_URL)
    projects = response.json().get("projects", [])
    for project in projects:
        if project["title"] == project_title:
            requests.delete(f"{BASE_URL}/{project['id']}", headers=HEADERS)
```

Figure 3: Example of a step definition file JSON handling (create_project_steps.py)

Additionally, we included a Helper.py file, which defines reusable elements such as headers for sending and receiving JSON and XML requests, base URLs for API endpoints, and XML templates for different API operations. This ensured consistency and streamlined our test implementation. A key component of our test suite is the helpers.py file, which streamlines API request handling by centralizing common values. This file defines headers for sending and receiving data in both JSON and XML formats, ensuring flexibility in testing different response types. It also includes base URLs for todos, projects, and categories, allowing consistent and reusable API calls across different tests. Additionally, helpers.py provides XML templates for todos and categories, enabling structured request body generation when creating new resources.

To execute the test suite efficiently, we use the command: **python -m behave features/**

This ensures all defined user stories are validated against the API, confirming that todos, projects, and categories function as expected. By structuring our test suite in this way, we achieved modularity, maintainability, and comprehensive coverage of our assigned user stories.

Source Code Repository

All of our deliverables are stored in a repository on GitHub. The repository is structured to ensure clarity and maintainability, with separate directories for feature files, step definitions, and helper scripts. Each team member contributed to their assigned user stories, ensuring that the test suite comprehensively validated the functionality of todos, projects, and categories, as defined in Part A of the project.

Beyond the test implementation, our repository also contains:

- A README.md file with setup instructions, dependencies, and how to run the tests.
- A bug summary document, detailing any issues encountered during testing.
- A video recording demonstrating the execution of our story tests.

One important consideration when running the tests is that the API must be restarted before executing the scripts to ensure a clean testing environment. This prevents inconsistencies in test results due to residual data from previous runs. By structuring our repository in this way, we ensured that our test suite remains modular, reusable, and easy to maintain, while also providing clear documentation for future development or debugging.

Findings of Story Test Suite Execution

During the execution of our story test suite, we encountered several challenges and limitations with the API. The most significant issue was the difficulty in deleting and modifying tasks, projects, and categories. Since the API does not allow users to manually assign an id when creating an entity, we could only retrieve or modify items by searching for their system-generated id. This made operations like deletion and updates particularly complex. For example, when attempting to delete a task, we first had to create it, then query all existing tasks to find its id, and only then issue a deletion request. This process became even more cumbersome when multiple tasks had identical attributes, requiring additional steps to identify the correct one.

Another key finding was that the API retains modifications across test runs, meaning that changes made during one test persist in the system unless manually reset. This required us to restart the API before running tests consecutively to ensure consistent initial conditions and avoid conflicts from previous test executions.

Despite these challenges, our story tests executed as expected, correctly validating the API's behavior across normal, alternative, and error scenarios. The results aligned with the API documentation, and all implemented tests accurately reflected the expected system responses.