McGill UNIVERSITY

Group 20

Samy Sabir	261119166
Yonatan Bensimon	261104350

Table of Contents

Summary of Deliverables

The required deliverables for part B of this project are:

- ➢ Gherkin scripts in feature files defining ten user stories featuring each of three scenario outlines executable with multiple different input values:
  - ○ Normal Flow
  - ○ Alternate Flow
  - ○ Error Flow
- ➢ Complete Story Test Suite consisting of Step Definitions implemented using Behave. Automation is handled by a master script *runAllTests.py* which ensures that the API is activated and that feature files are randomly executed by Behave.
- ➢ Bug summaries
- ➢ Story test Suite video

Source Code Repository

The source code repository for this project is hosted on GitHub and serves as the central version control system for managing the implementation and automation of story tests for the REST API explored in Part A. The repository is structured to ensure modularity, reusability, and maintainability while adhering to Bob Martin's guidelines on Clean Code principles.

In the main directory 'Part B/features' of the project, you must run *pip install behave* to install the *Behave* testing framework, which we used to run our steps implementations of our features with Python. *Behave* is a semi-official Python implementation of cucumber, which uses components from cucumber. We used it to implement our step definitions and run the gherkin scripts. Additionally, Python3 must be installed to run any of the source code.

Afterward, run *python runAllTests.py* in the terminal to execute our main running script. This setup script defines two important methods: *shutdown_system()* and *start_system(),* which allow every scenario to reset the API to ensure test independence. Furthermore, *behave* possesses the particular feature of using a custom runner class. In that regard, the main script defines the class *ShuffleRunner* which randomly shuffles the feature files' execution using Python's *random* module. This class is passed through an internal *Behave* method *run_behave(),* which ensures the random execution of story tests.

Similarly to Part A of this project, the main script asks the user if the API is running, and if it isn't, it will start as a subprocess. Therefore, the API is always guaranteed to be running.

Structure of Story Test Suite

The Story Test Suite is structured as follows:

1. /features

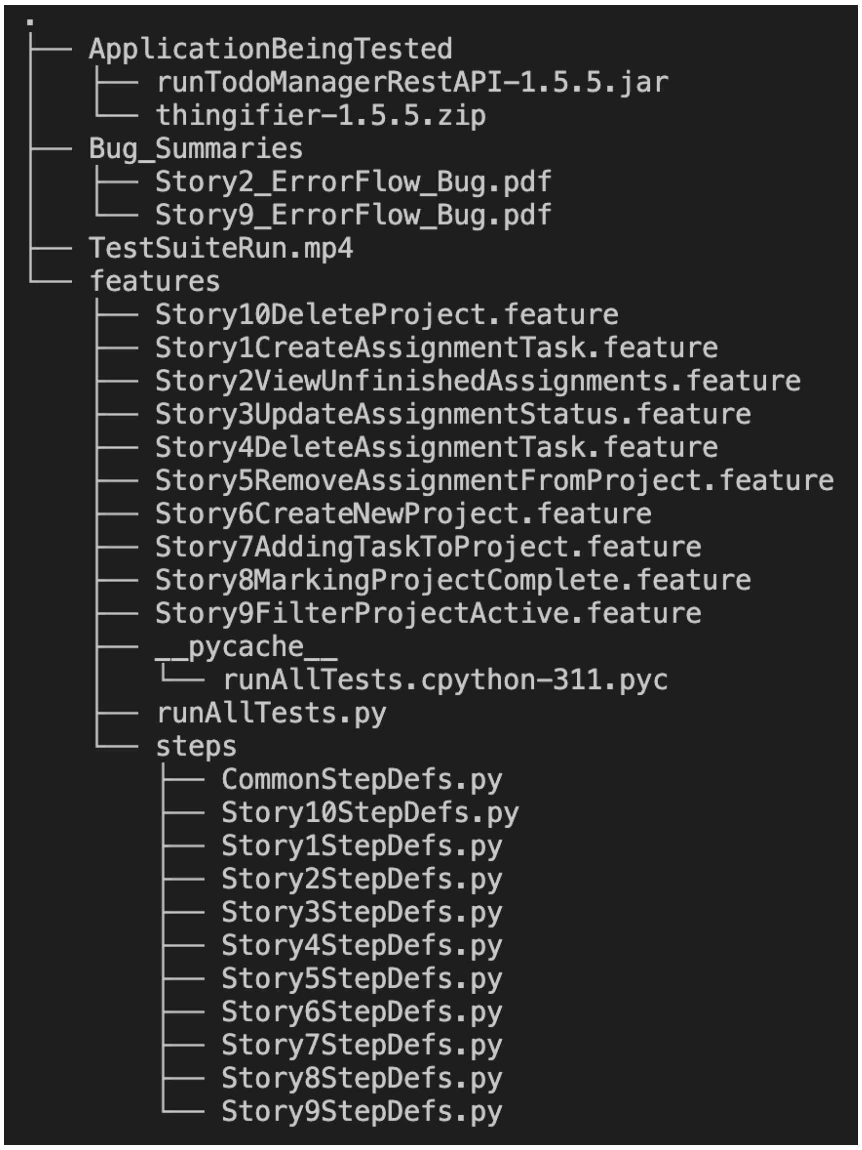
   The features/ directory houses all 10 Gherkin .feature files, each corresponding to a specific user story. Each file is identified by its StoryNumber and its StoryName. These files define scenario outlines that include normal, alternate, and error flows, with placeholders for variable data to enable parameterized execution. Additionally, each feature file contains a background section to establish the necessary preconditions before executing scenarios.

2. /features/steps

   The steps/ directory contains 11 step definitions implementations, which are written using *Behave*. 10 of these step definitions map the natural language Gherkin steps to executable Python code. They are named accordingly with their StoryNumber. Additionally, they each possess a *before_scenario()* and *after_scenario()* method which in *behave* allows for code execution at the beginning and end of each scenario. These methods allow us to properly restart the API with a clean state for each scenario. The last executable step definition file, *CommonStepDefs.py,* houses all common step definitions that occur in more than one story. This allows the reusability of frequently used code in multiple testing files.

3. /features/runAllTests.py
   As explained in the section *Source Code Repository*, the main script *runAllTests.py* lists useful utilities for starting and shutting down the API. Additionally, it contains the random shuffler class parsed through *behave* to randomly execute story tests every time. Finally, it asks the user for confirmation on whether the API is running or not and starts it if necessary, ensuring that the tests can proceed without interruptions due to an inactive or unavailable service.

Refer to Figure 1 for the Working directory structure.



```
.
├── ApplicationBeingTested
│   ├── runTodoManagerRestAPI-1.5.5.jar
│   └── thingifier-1.5.5.zip
├── Bug_Summaries
│   ├── Story2_ErrorFlow_Bug.pdf
│   └── Story9_ErrorFlow_Bug.pdf
├── TestSuiteRun.mp4
└── features
    ├── Story10DeleteProject.feature
    ├── Story1CreateAssignmentTask.feature
    ├── Story2ViewUnfinishedAssignments.feature
    ├── Story3UpdateAssignmentStatus.feature
    ├── Story4DeleteAssignmentTask.feature
    ├── Story5RemoveAssignmentFromProject.feature
    ├── Story6CreateNewProject.feature
    ├── Story7AddingTaskToProject.feature
    ├── Story8MarkingProjectComplete.feature
    ├── Story9FilterProjectActive.feature
    ├── __pycache__
    │   └── runAllTests.cpython-311.pyc
    ├── runAllTests.py
    └── steps
        ├── CommonStepDefs.py
        ├── Story10StepDefs.py
        ├── Story1StepDefs.py
        ├── Story2StepDefs.py
        ├── Story3StepDefs.py
        ├── Story4StepDefs.py
        ├── Story5StepDefs.py
        ├── Story6StepDefs.py
        ├── Story7StepDefs.py
        ├── Story8StepDefs.py
        └── Story9StepDefs.py
```

Figure 1. Project B Working Directory Structure

Findings of Story Test Suite Execution

The Story Tests were written from the application domain of a Student. Therefore, we associated TODOs with Assignments and projects with courses. This is the most appropriate real-life example. All defined user stories/features for TODOs passed all tests as seen in the attached video. This allowed us to confirm the findings explored in part A. However, additional bugs were discovered during the tests' execution. In fact, *Story2ViewUnfinishedAssignments.feature's* Error flow didn't produce the expected behavior. For context, the concerned user story is “As a student, I want to view unfinished assignments, So that I can keep track of unfinished work,” and the scenario outline for this flow is “View unfinished assignments with invalid doneStatus.” We expected the test to return an error message when doneStatus is not a boolean value with a 400 Bad request status code. However, it returned a 200 OK status code and listed no error messages. This is unwanted behavior, and it is not documented in the swagger documentation. Refer to Story2_ErrorFlow_Bug for more details.

Similarly, an almost identical bug was also found during the execution of the Story9FilterProjectActive.feature Error Flow scenario. More specifically, a GET request was made to gather all projects with an invalid (non-boolean) active status. We would usually expect a 400 Bad request status code to be returned, but, instead, a 200 OK status code was returned with an empty list. This does not make much sense, as an invalid request would be expected. More information can be found in the Story9_ErrorFlow_Bug file.