Ding Ma, 260871301, ding.ma@mail.mcgill.ca
Michael Li, 260869379, er.li@mail.mcgill.ca
Petar Basta, 260735072, petar.basta@mail.mcgill.ca
William Zhang, 260865382, william.zhang2@mail.mcgill.ca

Automation Project Part B
Group #2

Link to video of the tests running

https://www.youtube.com/watch?v=TqNS565xi7c&feature=youtu.be

In the video, you will see that the test runs in different orders when the *runToDoManagerRestAPI* server is active. If it is not, all the tests will come back as an error since HTTP request cannot reach the end point.

Deliverable Summary

The goal of this deliverable was to test the *runToDoManagerRestAPI* with user stories. This allows us to use behavior driven testing where a user gives an input, we can expect a certain output. For each of the story, there are three flows that we need to accomplish:

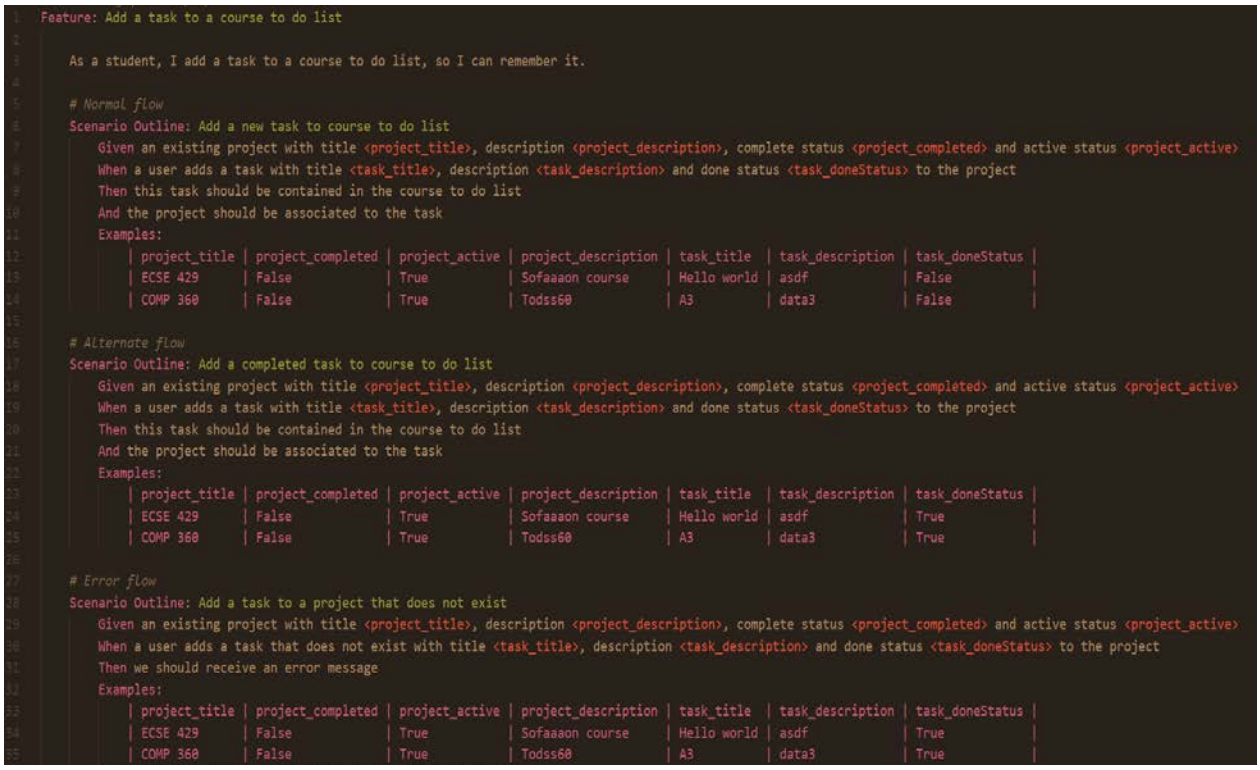
1- Normal flow
2- Alternate flow
3- Error flow

Compared to last deliverable, this one is a lot more straightforward. Since we know from last iteration some possible bugs from the *runToDoManagerRestAPI*, we can either fix or avoid them this time. Hence, in this deliverable we need to make multiple API calls in order to achieve that the user wants. To determine which API endpoints are needed we first began by going through the Swagger API. For each of the user story, the table below describes what API endpoints are needed as well as the HTTP verb. The table on the next page describes which endpoints are needed for each user story.

A Gherkin script was created for each user story containing all 3 flows (normal, alternate and error). We then use a third-party package in Python called “behave” to run these Gherkin scripts which allowed us to test different scenarios. Within each Gherkin script, for each one of the flows, we have used scenario outlines in order to create a scenario and inject data into this scenario using an example data table. This allowed us to execute the same scenario multiple times (as many times as there are rows in the example table). Hence, in this way we were able to automate the scenario testing with different input data.

In fact, Gherkin allowed us to specify scenarios using natural language and a set of special keywords to give structure and meaning to executable specification. It supports many languages but, in this deliverable, we used English as shown in figure 1. The scenario outline keyword allowed us to run the same scenario multiple times with different combinations of values.

User Story	Gherkin keyword to HTTP verb and API endpoint			
	Given	And	When	Then
As a student, I categorize tasks as HIGH, MEDIUM or LOW priority, so I can better manage my time.	POST todo POST category GET todo GET category	GET categories/id	POST todo/id/category	GET todo/id
As a student, I add a task to a course to do list, so I can remember it.	POST Projects GET Projects	POST todos GET todos POST todo/id/tasksof	POST todo GET todo/id POST todo/id/tasksof	GET todo/id
As a student, I mark a task as done on my course to do list, so I can track my accomplishments.	POST todo GET todo		Put todo/id	GET todo/id
As a student, I remove an unnecessary task from my course to do list, so I can forget about it.	POST project GET project	POST todo GET todo/id	GET todo/id DELETE todo/id GET todo/id	GET todo/id
As a student, I create a to do list for a new class I am taking, so I can manage course work.	POST project GET project		POST project GET project	GET project
As a student, I remove a to do list for a class which I am no longer taking, to declutter my schedule.	POST project GET project		GET project GET project/id DELETE project/id GET project/id	GET project
As a student, I query the incomplete tasks for a class I am taking, to help manage my time.	POST todo POST project Post project/id/tasks	POST todo POST category/id/tasks		GET project/id/tasks
As a student, I query all incomplete HIGH priority tasks from all my classes, to identity my short-term goals.	POST todo POST category POST category/id/tasks	POST todo POST category/id/tasks		GET category/id/todos
As a student, I want to adjust the priority of a task, to help better manage my time.	POST todo POST category POST category/id/tasks	POST category/id/todos	DELETE category/id/todos POST category/id/tasks	GET category/id
As a student, I want to change a task description, to better represent the work to do.	POST todo GET todo		PUT todo/id	GET todo/id



```
Feature: Add a task to a course to do list

    As a student, I add a task to a course to do list, so I can remember it.

    # Normal flow
    Scenario Outline: Add a new task to course to do list
        Given an existing project with title <project_title>, description <project_description>, complete status <project_completed> and active status <project_active>
        When a user adds a task with title <task_title>, description <task_description> and done status <task_doneStatus> to the project
        Then this task should be contained in the course to do list
        And the project should be associated to the task
        Examples:
            | project_title | project_completed | project_active | project_description | task_title  | task_description | task_doneStatus |
            | ECSE 429      | False             | True           | Sofaaaon course     | Hello world | asdf             | False           |
            | COMP 360      | False             | True           | Todss60             | A3          | data3            | False           |

    # Alternate flow
    Scenario Outline: Add a completed task to course to do list
        Given an existing project with title <project_title>, description <project_description>, complete status <project_completed> and active status <project_active>
        When a user adds a task with title <task_title>, description <task_description> and done status <task_doneStatus> to the project
        Then this task should be contained in the course to do list
        And the project should be associated to the task
        Examples:
            | project_title | project_completed | project_active | project_description | task_title  | task_description | task_doneStatus |
            | ECSE 429      | False             | True           | Sofaaaon course     | Hello world | asdf             | True            |
            | COMP 360      | False             | True           | Todss60             | A3          | data3            | True            |

    # Error flow
    Scenario Outline: Add a task to a project that does not exist
        Given an existing project with title <project_title>, description <project_description>, complete status <project_completed> and active status <project_active>
        When a user adds a task that does not exist with title <task_title>, description <task_description> and done status <task_doneStatus> to the project
        Then we should receive an error message
        Examples:
            | project_title | project_completed | project_active | project_description | task_title  | task_description | task_doneStatus |
            | ECSE 429      | False             | True           | Sofaaaon course     | Hello world | asdf             | True            |
            | COMP 360      | False             | True           | Todss60             | A3          | data3            | True            |
```

Figure 1 Example of a Gherkin script file

Structure of acceptance test suite

The acceptance test suite is defined inside the features directory. All the files with extension *.feature* are the Gherkin scripts which together form the acceptance test suite. Figure 1 is a sample Gherkin file to add a task to a todo list. Within the features directory, the step directory contains the *Python* source code which will execute the Gherkin scenarios. Each *.feature* file is mapped to a *.py* file in the steps directory. Their filenames are identical. Within each Gherkin script, a single user story is covered. Three different flows are tested using scenario outline. The scenario outline describes a scenario and let us define placeholders in which we can inject data from the example data table defined after the scenario. Each flow, normal, alternate and error, is tested with all the data in the data table. As mentioned above, each user story has three flows that we need to test.

If we consider the user story “As a student, I mark a task as done on my course to do list, so I can track my accomplishments” as an example, the goal of this story is to change the task status. For the normal flow, the test will first begin by creating a task with a *task_title, task_description,* and *task_status* and a project representing the todo list with *project_title, project_description, project_completed* and *project_active*. The task is part of the project. Then, when the user marks the task as done, we need to send a PUT request in order to update the task status. Finally, we can verify that the task status by sending a GET request and asserting that its status is done. Figure 2 below is a sample output when testing "create todo" user story.

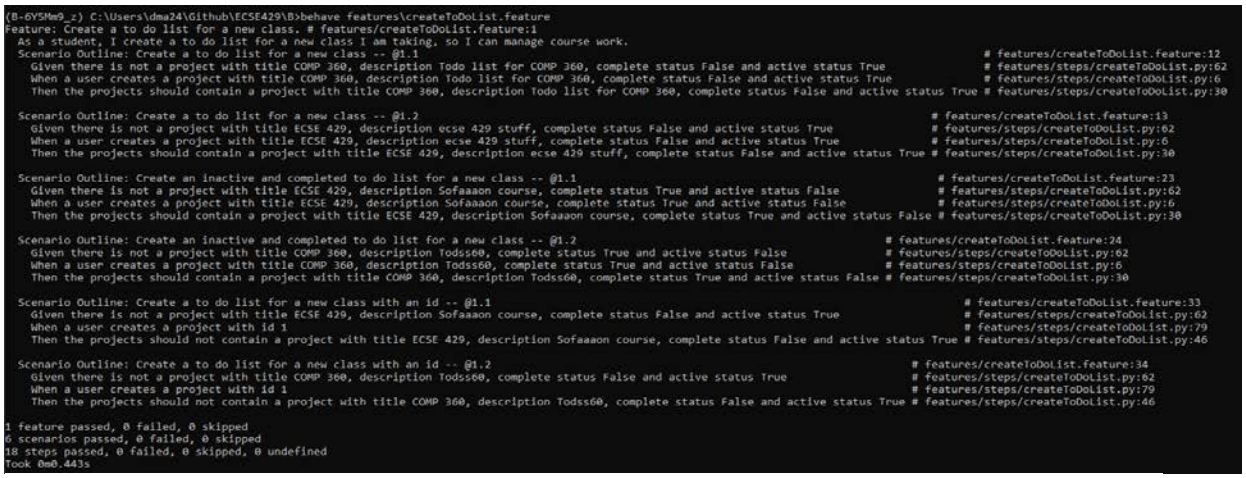


Figure 2: Behave output sample

Similarly, for the alternate flow, we can change back to the status of a completed task to incomplete by performing the same operations. An example of the error flow would be trying to mark a task done when the task does not exist. This should obviously throw an error as the *task_id* does not exist. For this deliverable, the above steps need to be repeated for all the user stories. Each user story has it's own *.feature* file in the feature directory. We simply need to create a Python function that maps to the appropriate sentence with the Gherkin keyword to pair them up with the appropriate HTTP request as show in figure 3. More details on the code repository structure will be discussed in the next section.

```python
@given(u'the a task with title {task_title}, description {task_description} and done status {task_doneStatus}')
def step_impl(context, task_title, task_description, task_doneStatus):
    create_todo = requests.post(url_todo, data=json.dumps({"title": str(task_title), "description": str(task_description),
                                                           "doneStatus": bool(task_doneStatus)_}_), headers=send_json_recv_json_headers)
    todo_res = create_todo.json()
    todos = requests.get(url_todo).json()["todos"]
    context.task_id = todo_res["id"]
    context.task_title = todo_res["title"]
    context.task = todo_res
    assert create_todo.status_code == 201 and todo_res in todos


@when(u'a user marks the task {task_title} as done')
def step_impl(context, task_title):
    task_update = {
        "title": str(task_title),
        "doneStatus": True
    }
    r = requests.put(url_todo_id % int(context.task_id), data=json.dumps(task_update), headers=send_json_recv_json_headers)
    assert r.status_code == 200
```

Figure 3 Sample Python Method

We can run the Gherkin script with the *Behave* package in Python to map Gherkin keywords to Python methods. For instance, when the "Given the following tasks" sentence in the Gherkin script is executed, it will perform the appropriate Python method mapped to the sentences using annotations that are specific to *Behave.* The table within the Gherkin script example is serialized and passed to the Python method where we can simply access the value of each column as string parameters. Then, we can do our request method to validate what the tests want to accomplish.

Source code repository

Our source code repository has a root directory and four subdirectories. The root directory contains our project deliverable A and project deliverable B. The B directory is our interest for this project. If we run the *tree* command by ignoring directory *A/*, we can see its layout in figure 4.

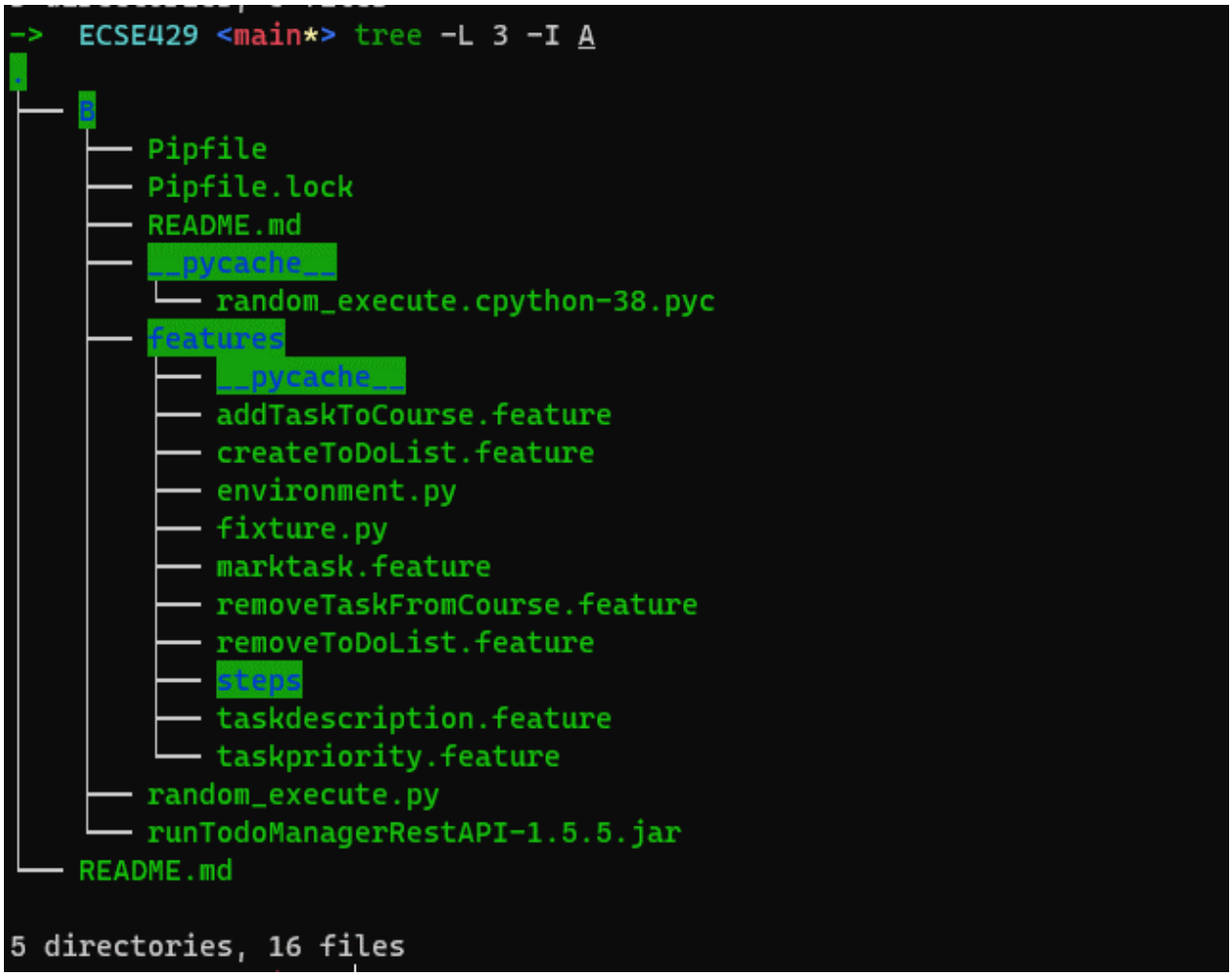


Figure 4: tree of root directory

Within the root directory of *B/*, we can see the application jar, *Pipenv* and *Pipfile.lock* which are Python dependency files, a README file, and the *features* directory. Like last deliverable, we are using *Pipenv* with Python 3.8 to run our tests. *Pipenv* helps us create the virtual environment for this project by running *pipenv shell && pipenv install* in the terminal assuming the user has already installed *Python*, *pip*, and *pipenv*. There is an additional package that allows us to do behavior driven testing which is included in the *Pipfile*.

For each user story that we are testing, there will be a distinct *.feature* file within the *features* directory as well as its associated Python file inside the *steps* directory. All those tests can be run with the *behave* command within the *pipenv shell*. We kept our *fixture.py* file from

last deliverable to restart the backend runToDoManagerRestAPI server between the tests. Hence, the server will stop at the end of each scenario and restart at the beginning of the next scenario to make sure that we restore the initial state upon completion of every test. The e*nvironment.py* file is used to pass this fixture method to the *behave* library. The *__pycache__* folder binaries that helps speed testing up, those can be safely ignored, or deleted.

The application has two running modes, one with the server enabled and one with the server disabled. If we run *python random_execute.py*, it will start the server as expected and execute all the Gherkin scripts in a randomized order. If we add the flag *--disable-server*, the runToDoManagerRestAPI server will not be started at the beginning of the tests since the fixture to shutdown and restart the server will not be passed to the methods. In this case, we can expect all the tests to fail. This proves that if the server is not running, all story tests will fail.

In both cases, it will invoke the appropriate Gherkin script located within *./B/features* directory. The Gherkin script will then call the appropriate Python method that are located in *./B/features/steps* directory. Those files contain our HTTP requests that will make the call to the *runToDoManagerRestAPI* endpoints.

Findings of Story Test Suite Execution

From last deliverable, we knew that there are some endpoints that are not behaving as expected. The table below is a summary of our patches. We either avoided them or made additional HTTP requests to fix them.

test_unexpected.py/Function	Bug	Solution
None	Deleting the last task in a category deletes the category itself	We added a blank task if the last task is deleted.
None	Deleting the last task in a project deletes the project itself	We added a blank task if the last task is deleted.
test_post_todos_id	When sending empty data to POST /todos/:id, we don't get an error mesage and the request successful	We made sure that there we no empty data field before hand
test_post_todos_id_tasksof	Unable to send ids in XML	We are sending JSON instead of XML
test_post_todos_id_tasksof	Unable to create relationship between todo and project since cannot send project id in XML	We are sending JSON instead of XML
test_get_todos_tasksof	This is an undocumented behaviour where /todos/tasksof is redirecting us to /projects	We did not use this endpoint.

test_get_todos_categories	This is an undocumented behaviour where /todos/categories is redirecting us to /categories	We did not use this endpoint.
test_get_project_task_wrong_id	Should return error at endpoint /project/:id/tasks when id doesn't exist.	We made sure that there we no empty data field before hand
test_head_project_task_wrong_id	Should return error at endpoint /project/:id/tasks when id doesn't exist.	We added an additional check before sending the HTTP request
test_post_project_id	Fails with XML body when it shouldn't	We are sending JSON instead of XML
test_connect_categories_tasks	Create connection from task to category, but only shows up in 1 direction	Since the it is unidirectional, we made sure to use the same direction in all our tests.
test_connect_tasks_categories	Create connection from category to task, but only shows up in 1 direction	Since the it is unidirectional, we made sure to use the same direction in all our tests.
test_post_todos_xml_json	Difference between documented input (sample data) and actual accepted input errorMessage	Nothing much we can do
test_post_todos_json_json	potential vulnerability, False in Python and "false" in Java, boolean conversion is vulnerable	we made sure to send "false" instead of False in order to comply with the java backend
test_put_todos_id	potential vulnerability, False in Python and "false" in Java, boolean conversion is vulnerable	we made sure to send "false" instead of False in order to comply with the java backend

In summary, there we no new findings when we wrote our user story tests. The main issue was some end points not behaving properly so we added some additional checks in order to fix them. As for the unidirectional association between task and category, and task and project, we made sure to only use to use the direction we created. Another possibility would be to make two requests when the user makes one to forcefully create the bidirectional link. Some small serialization error between converting from Python diction to JSON then Plain Old Java Objects can be fixed with a string instead of using native Python Boolean.