ECSE 429 Software Validation
Term Project
Part 1

Group 17
Weige Qian 260763075
Erdong Luo 260778475
Yudong Zhou 260721223
Hanwen Wang 260778557

1. Deliverable summary

In part A of this project, our team did exploratory testing in groups of 2 at first. Its aim was to identify documented and undocumented capabilities and give us a general idea about this Todo Manager system. Each group had its own session notes, which can be found in the “Exploratory Testing” folder.

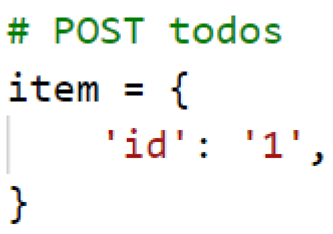
In each session note, we recorded all the scripts of API calls in txt files and we also took screenshots and compressed all the photos into one file. Also, a summary of session findings, a list of concerns and a list of new testing ideas were written at the end of each test session as well.

Then we moved on to unit testing, we wrote some unit tests to confirm if the API was performing as documented. If it was, we continued to the next test, if it was not, we listed them down and checked if it was an undocumented capability. If it was, we classified it as undocumented capability, otherwise it was classified as a bug and was written in the bug summary document. More tests were added later to test some invalid operations, we recorded and compared all results we got for each of them. After all these were done, we merged everything and shared our opinions. At last, we recorded the process of running all unit tests implemented. All files related to unit tests can be found in the “Unit Test” folder. In this report, we will summarize our findings and explain the unit test approaches that we have used.

2. Findings of exploratory testing

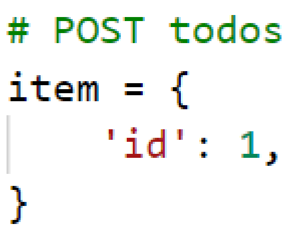
During exploratory testing, we were more familiar with this Todo Manager system. Several potential undocumented capabilities and errors were detected. Those potential capabilities and errors were listed down so that we could pay more attention to them in unit tests. Since we did exploratory testing at a relatively fast pace because of the time budget limitation of each session, we could not confirm those things we found actually existed. Further detailed tests should be implemented to test again, there might be other capabilities or errors we did not distinguish and maybe those we found already were caused by human-made errors. More details of exploratory testing can be found in files under the “Exploratory Testing” folder.

For some tests during exploratory testing, we were not aware of the constraints when sending JSON data to the server, and the result was that some bugs detected during exploratory testing were actually not bugs but rather the outcome of incorrect JSON format. For example, in some POST methods, we need to send the id of the item to create a new relationship between items. Figure 1 shows the JSON data that works, and Figure 2 shows the JSON data that does not work. During exploratory testing, some cases using Python scripts (instead of Postman) were done based on incorrect JSON data, which were concluded as a bug. But actually, those were not bugs because the API could work if we could provide the correct JSON data. We did not realize it until we were implementing the unit test suite. Therefore, one may find that, in our exploratory testing, there are more reported bugs than the actual failed cases in the unit test suite



```
# POST todos
item = {
    'id': '1',
}
```

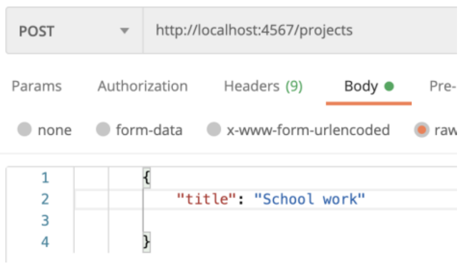
Figure 1. Correct JSON



```
# POST todos
item = {
    'id': 1,
}
```

Figure 2. Incorrect JSON

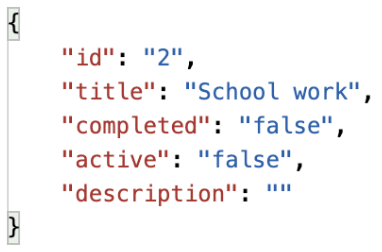
We also find out that while doing the POST api call, if we miss some parameters in the body, the manager will still generate a new item. But the missing parameters will be added and automatically set to default. All the booleans will be set to false and all strings will be set to an empty string.



POST | http://localhost:4567/projects

```
{
    "title": "School work"
}
```

Figure 3. POST Input



```
{
    "id": "2",
    "title": "School work",
    "completed": "false",
    "active": "false",
    "description": ""
}
```

Figure 4. POST Output

From all exploratory testing sessions, there is no undocumented API found.

3. Background environment

For the unit test suite of this project, the team chooses to use python as the only programming language due to its easy syntax. Python has its own library called ‘unittest’ which could provide a pipeline of the unit test process to the users. To send Http requests to the server easily, the team chooses to use a library called ‘requests’. It provides many advanced and useful functions to the users. Due to ‘requests’ being a third-party Python library, before running the unit test script, please make sure you have installed this package by command ‘pip install command’. To deal with the JSON object easily, the script also imports ‘json’ library which could convert the json objects to python objects and vice versa easily. To start the server in the unit test, the library ‘os’ is used to call the command "java -jar runTodoManagerRestAPI-1.5.5.jar" in the code. In addition, ‘from threading import Thread’ is required because, otherwise, the command calling the server will block all the following code, since it will keep running. Using a thread enables the server to start in the background and avoids the blocking.

4. Structure of the test suite

There are four Python scripts for unit testing. The file ‘unit_test_todos.py’ contains all unit tests for APIs starting with ‘/todos’. The file ‘unit_test_projects.py’ contains all unit tests for APIs starting with ‘/projects’. The file ‘unit_test_categories.py’ contains all unit tests for APIs starting with ‘/categories’. There is another file ‘unit_test_general.py’ which has test cases for other APIs such as ‘/docs’ and ‘/shutdown’. The unit tests can run in different orders, and more details are demonstrated in the video.

In the testing script, the team creates a tester class for every API in the document and an extra class for testing the malformed JSON and XML payload. Under each tester class, we wrote test cases for different methods of an API (such as GET, POST and DELETE) under different conditions. This aims to confirm the API does what it is supposed to do. Details will be discussed in section 5. When we are writing the unit test, we generally divide the whole structure into two parts. First, we test the APIs that are documented on the website. Then we try to test the server with invalid inputs and see if the server gives proper error messages.

After we get the output data from the system, we check if the output matches what it is supposed to be. First we check if the return code is correct. The return codes that we encountered are 200 (get success), 201 (post success), 400 (invalid input) and 404 (command not found).

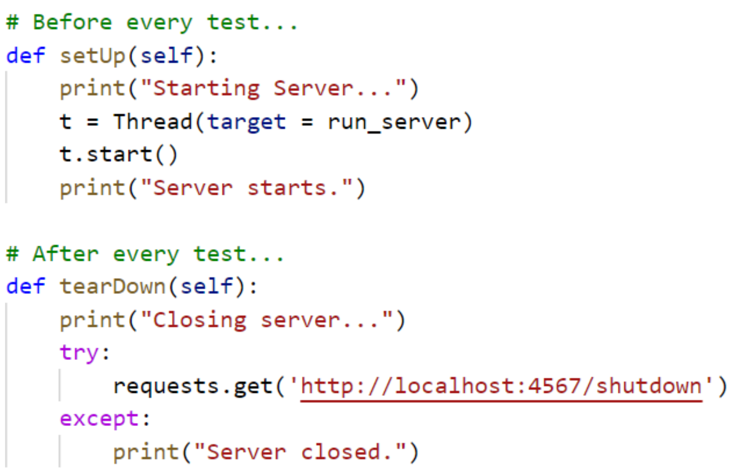
During the implementation of unit tests, we have also tried to explore the existence of undocumented APIs, however, no such API was found. For example, for some APIs where POST method works but PUT is not mentioned, we have tested to run PUT and see if it works. Also, for some APIs such as ‘/categories/:id/projects’, where GET, HEAD and POST can work, we have tried to run DELETE with a message indicating the id number. None of these experiments could find undocumented APIs, hence, we did not include these cases in our unit test suite.

We also test some edge cases. First, we brainstormed some possible edge cases like incomplete input, invalid input and input with mal syntax. Then we wrote corresponding unit tests and ran the tests.

5. Description of source code

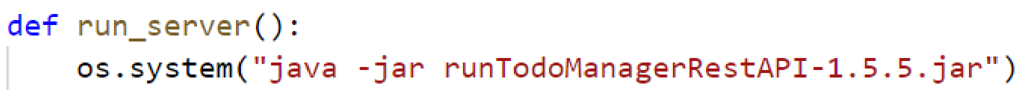
In this project, the test suite needs to ensure the tests fail if the service is not running. To solve this problem, the team has tried two main solutions in the project. The ‘unittest’ library provides a setup and a teardown function. The program will run the setup function and teardown before and after running every test function correspondingly. So in the first solution, the program would try to send a request to the home page of the server in the setup function, if it catches a connection error, it will set the test result to fail. In this case, the program needs to recall all the steps it made in the test function to restore the system to the initial state. The second solution is much more straightforward. The program would boot the server itself in the setup function and shut down it in the teardown function. In this case, the program does not need to recall the steps it made in the test function, since every time it restarts the server, the server would go back to the initial state itself. For the second solution, the team only needs to make sure that the shutdown API is working as expected. Apart from those two main solutions, another extra small solution is also implemented in the last two pieces of API testing (when testing ‘/docs’ and ‘/shutdown’), which is using the “try and except” concept to catch the connection error inside the function.

In this project, the team chooses to use the first solution in the ‘/todos’ and all its related APIs. Because restarting the server is the process that increases the load of the whole system, and it takes longer time. The team has tried many more test cases in ‘/todos’ and all its related APIs compared to others. For the other two main API groups, the team chooses to use the second solution to ease the job. Also, using different approaches to ensure that the server is running can give the team more insight about the application and may reveal more bugs that might not be found if only one of the approaches was used.



```python
# Before every test...
def setUp(self):
    print("Starting Server...")
    t = Thread(target = run_server)
    t.start()
    print("Server starts.")

# After every test...
def tearDown(self):
    print("Closing server...")
    try:
        requests.get('http://localhost:4567/shutdown')
    except:
        print("Server closed.")
```

Figure 5. Server Restart

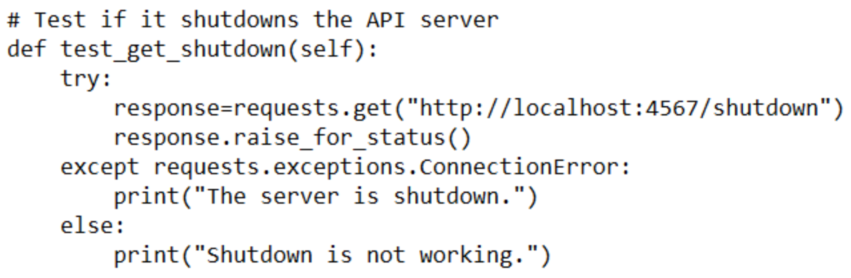


```python
def run_server():
    os.system("java -jar runTodoManagerRestAPI-1.5.5.jar")
```

Figure 6. Start the Server

Figure 5 shows the code implementation that enables the server to restart for every single test case. This approach is used for API groups ‘projects’ and ‘categories’. When running the test script, the unit test library will automatically call setUp() before each test and tearDown() after each test. In setUp(), the function in Figure 6 is called to start the server using the command line. Note that this function is called by thread so that it can run in the background. In tearDown(), the API ‘’shutdown’ is used. The team tested this API first, so that it is ensured to work as expected. Calling this API will result in connection error, which is normal because the server is already closed when the client (the test script) tries to get feedback. Therefore, a ‘try-except’ condition is used to avoid too many error messages showing up when running the entire test script.

For the last two API tests, especially for the shutdown API, since the shutdown API will close the server, this will lead to the situation that we cannot get requests from the server. As a result, if we want to test whether the system is successfully shutdown or not, we should use the try and except concept inside the class and after the shutdown API is called. Then we will catch a connection error exception if the server is shutdown, otherwise it will go to the else function and confirm the shutdown is not working. The snippet of code is shown in the figure below.

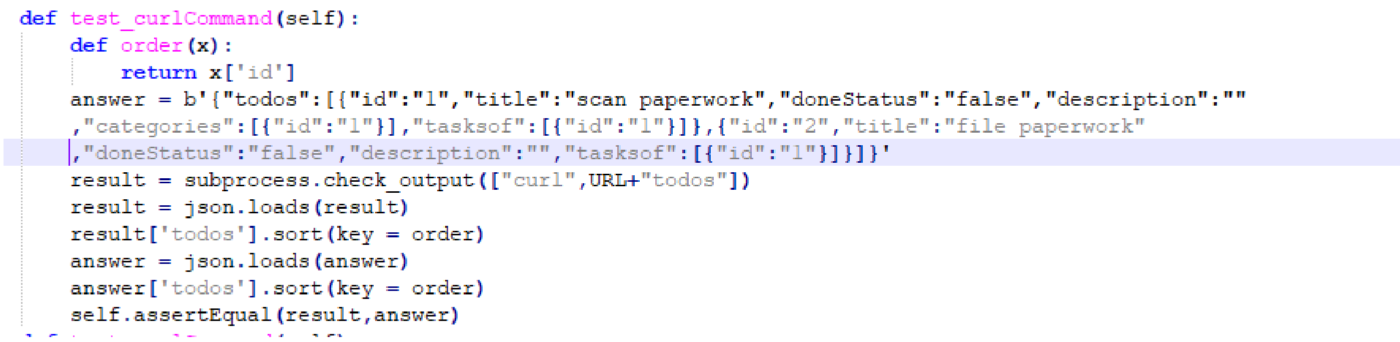


```
# Test if it shutdowns the API server
def test_get_shutdown(self):
    try:
        response=requests.get("http://localhost:4567/shutdown")
        response.raise_for_status()
    except requests.exceptions.ConnectionError:
        print("The server is shutdown.")
    else:
        print("Shutdown is not working.")
```

Figure 7. Test shutdown

Since Python could handle JSON objects much more easily, most test cases are using the JSON object. But there is one test case to show that the server could provide XML messages to show the results.

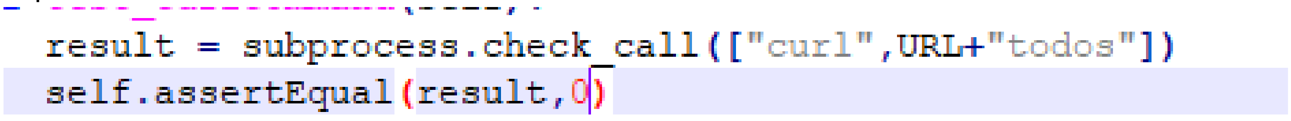
The program also uses the command line queries by using the ‘subprocess.check_output’ function. It executes a curl command in the command line and gets the JSON output that is printed in the command line so that it can compare the output to the right JSON message. Sometimes, due to a previous change in the data item, the received JSON object has a different order in the list which could influence the assertEqual() function’s result, so we need to order the data item by their id. Figure 8 shows the implementation.



```
def test_curlCommand(self):
    def order(x):
        return x['id']
    answer = b'{"todos":[{"id":"1","title":"scan paperwork","doneStatus":"false","description":""
    ,"categories":[{"id":"1"}],"tasksof":[{"id":"1"}]},{"id":"2","title":"file paperwork"
    ,"doneStatus":"false","description":"","tasksof":[{"id":"1"}]}]}'
    result = subprocess.check_output(["curl",URL+"todos"])
    result = json.loads(result)
    result['todos'].sort(key = order)
    answer = json.loads(answer)
    answer['todos'].sort(key = order)
    self.assertEqual(result,answer)
```

Figure 8. Test Curl Command

The program also uses the ‘subprocess.check_call’ function to execute a curl command and get the return code and compare it to 0 which shows no error. Figure 9 shows the implementation.



```
result = subprocess.check_call(["curl",URL+"todos"])
self.assertEqual(result,0)
```

Figure 9. Return Code

When testing the malformed JSON and XML object, we use a malformed static XML and JSON string (such as one lacking a bracket) and post it to the server to see the result.

6. Findings of unit test suite execution

During the unit test suite execution, the team has many interesting findings.
First, we found some bugs.

In the GET method of ‘/todos’, the user could add the URL parameters on the request. The result would be a filtered data item. But when the team uses the non-existing parameter like ‘titles’, the result should be an empty list or the error message to tell us the parameter is wrong, but we received a list without any filtration.

Then, In the GET method of ‘/todos/{id}/categories’, the user would receive the categories that todos with id have a relation with. But if we use a non-existing id in this request, the result should be an error message with error code 404 or 400. But the actual result is 2 copies of categories information. This mistake exists on every relation-related API such as ‘/todos/{id}/categories’,’/projects/{id}/categories

Also, In the post method of ‘/todos/{id}/tasksof’. The user could add relationships between todos and projects by sending the id of the project. When no id is posted, the server should do nothing. But in fact, the server creates a project automatically with the default value and adds a relationship between todo with id and the new project.

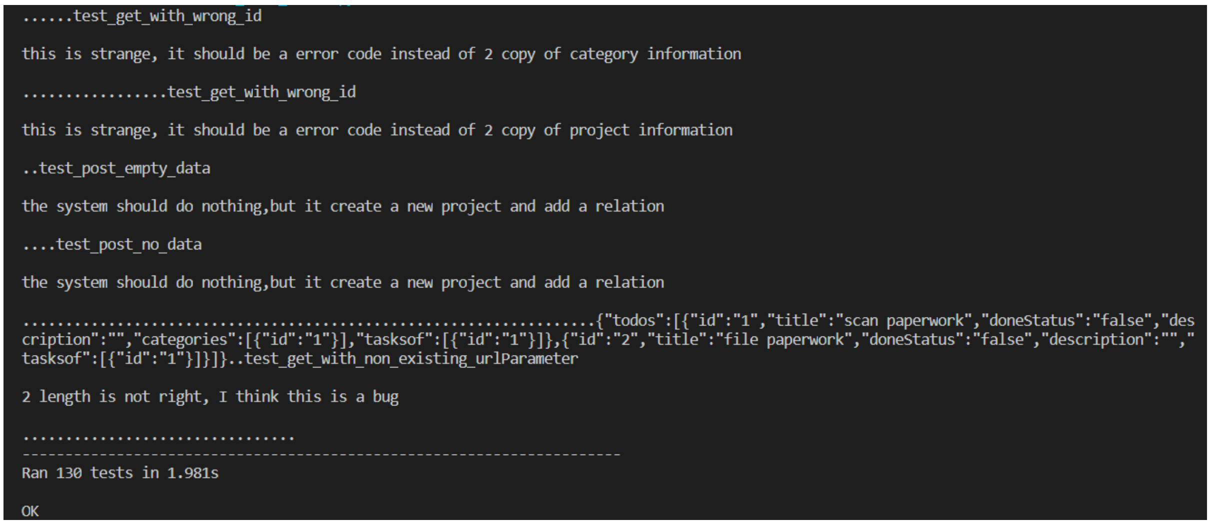
In the POST method of ‘projects’, even if we did not enter anything in the body, it will still create a project when we send the request. All the parameters will be set to default. This is actually not reasonable.

In the GET method of ‘/projects/{id}/categories’, we have tried different situations. In the very beginning, after we have just started the server, there are no categories in the project1 list. So if we try to get the list of categories, the server will return an empty array with no element. But if we try to get the category list of a non-exist project, it works. There is no error message sent by the server saying the project does not exist. Instead, the server returned the whole list of the categories in the first project. Even when there are more than one project in the project list, it will always return the category list of the first project.

Besides the bugs we mentioned, we also have found some undocumented capabilities. Most of them are about the POST methods. When we want to make a post request, The type of data is specified, The team has tried the different types. The result shows that the server does not accept any other type when the required data type is boolean such as ‘active’,’doneStatus’ and ‘completed’, even the post data is ‘true’ and ‘false’ in string type. But other data is different. In POST methods of API ‘/todos’ and ‘todos/{id}’, the team tried other types such as int, float, boolean, list, and dictionary on the title and description field. It is found that the server accepts all except the dictionary type with an item inside it. When the server receives an int type title or description, it will change it to the float and store it in str type. When the server receives a float type title or description, it will store it in string type. Same as boolean type. When the server receives a list type title or description, It will store the last item in the list in string type. If the server receives an empty list or an empty dictionary on the title and description field, it will treat it as nothing.

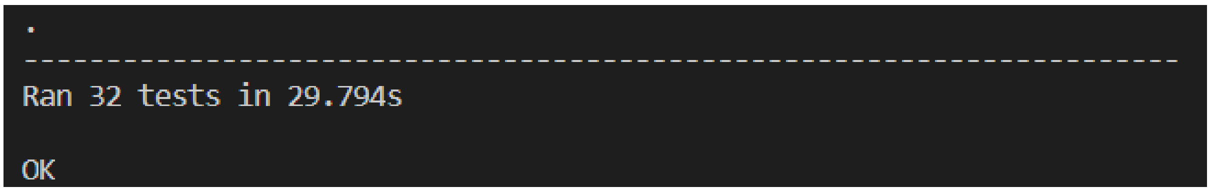
In the docs of API in the server, the description of the PUT method and POST method of ‘/todos/{id}’ is the same. But the actual usage is not the same, in the POST method, if the field is not contained in the request, the server will keep the original value. In contrast, If the user does not include some data field in the request body in the PUT method, The server will change it to the default value or delete the relation.

7. Unit Test Execution Result



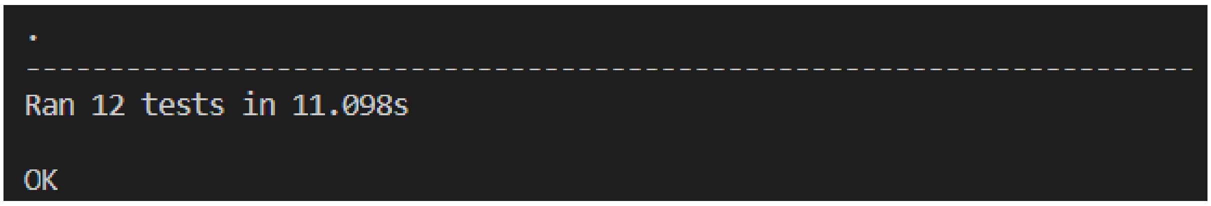
```
......test_get_with_wrong_id

this is strange, it should be a error code instead of 2 copy of category information

................test_get_with_wrong_id

this is strange, it should be a error code instead of 2 copy of project information

..test_post_empty_data

the system should do nothing,but it create a new project and add a relation

....test_post_no_data

the system should do nothing,but it create a new project and add a relation

..................................................................{"todos":[{"id":"1","title":"scan paperwork","doneStatus":"false","des
cription":"","categories":[{"id":"1"}],"tasksof":[{"id":"1"}]},{"id":"2","title":"file paperwork","doneStatus":"false","description":"","
tasksof":[{"id":"1"}]}]}..test_get_with_non_existing_urlParameter

2 length is not right, I think this is a bug

...............................
----------------------------------------------------------------------
Ran 130 tests in 1.981s

OK
```

Figure 10. Screenshot of Testing Result of unit_test_todos.py



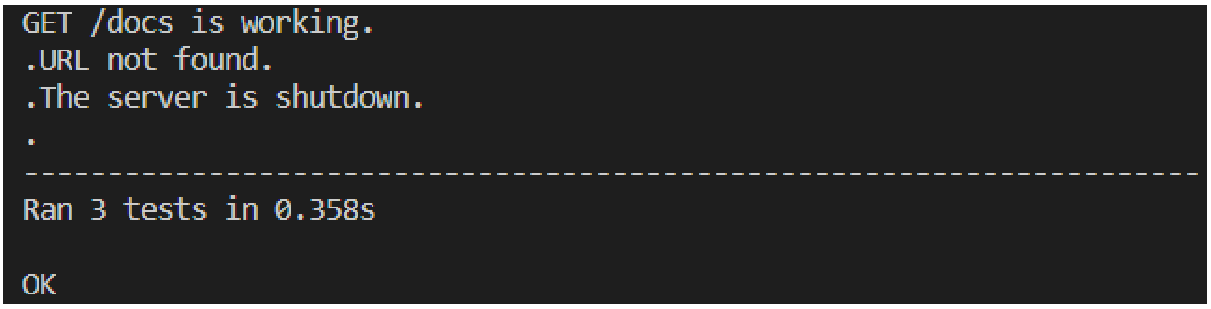
```
.
----------------------------------------------------------------------
Ran 32 tests in 29.794s

OK
```

Figure 11. Screenshot of Testing Result of unit_test_projects.py



```
.
----------------------------------------------------------------------
Ran 12 tests in 11.098s

OK
```

Figure 12. Screenshot of Testing Result of unit_test_categories.py



```
GET /docs is working.
.URL not found.
.The server is shutdown.
.
----------------------------------------------------------------------
Ran 3 tests in 0.358s

OK
```

Figure 13. Screenshot of Testing Result of unit_test_general.py