ECSE 429 Software Validation
Term Project
Project Report

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

1. Part A

Collaboration and Implementation

In part A of the project, we first did exploratory testing and then implemented unit test suite to test the given application. Specifically, in exploratory testing, the team followed Charter Driven Session Based Exploratory Testing to study the behavior of the “rest api todo list manager”. In the unit test suite, we wrote the test modules for each API identified in the exploratory testing, using Python as the programming language and the “unittest” library as the main unit test framework.

In the exploratory testing, the testing sessions are timeboxed at around 45 minutes and done in pairs. During a testing session, one team member was exploring the APIs by using different tools, and the other team member was taking session notes based on the findings provided by the first team member. Since our team has four members, we can have two pairs of people. We divided the APIs listed on the document into two groups, so each pair was responsible to explore one group of the APIs. Each pair spent around 2 to 3 sessions to finish exploring one group. Different tools were used in the exploratory testing, including Postman and simple Python scripts. Potential APIs that might not be listed in the document were also explored.

In the unit test suite, we divided the APIs into four groups, so that each team member was responsible for one group of APIs. The tests were written in Python using the ‘unittest’ library, because Python provides a relatively easy way to manipulate API requests. The team also brainstormed some edge cases that could be worth testing in the unit test suite.

Findings and Recommendations

From the exploratory testing, at first we thought that we found some bugs compared to the document. Actually those were not bugs but mistakes made by us during the testing sessions.

For some APIs during the exploratory testing, we were not aware of the JSON data format that we sent to the server, which resulted in wrong conclusions on those APIs. For example,

as illustrated by Figure 1 and Figure 2, when sending an object id as a JSON message to the server, the id field should be in string format instead of integer. If the id number is sent as an integer, then the server will return an error message when the API is called. In this case, if we do not know the issue about the JSON format, then we will mistakenly conclude that the tested API is not working as expected. This issue was found when we were implementing the unit test suite. We should have paid more attention when dealing with the requests with JSON messages. We also learned that some human-made mistakes could lead to completely opposite conclusions during testing, so we have to be very careful and it is necessary to review the tests in order to find human-made mistakes.

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

Figure 1: Correct JSON

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

Figure 2: Incorrect JSON

When implementing the unit test suite, we found some limitations and extra capabilities of the application.

One interesting capability is that, in the JSON messages, although the id field has to be string, and the fields such as “active” need to be boolean, other fields such as the title and description are able to accept multiple data types like int, float and boolean, where all the accepted data will be converted into string by the application and stored in the system.

Another finding is that, when doing PUT and POST requests, although the document mentions that they are the same, these two methods actually have different effects. A POST request will only overwrite the fields contained in the JSON message, while a PUT request will overwrite every field of the existing object including the relations. If some fields are not included in the JSON message in a PUT request, they will be replaced with the default values. This behavior of the system may be caused by the characteristics of PUT and POST methods. PUT and POST can both send data to the server, however, the difference is that PUT requests are idempotent.

One improvement that could be done in the unit test suite is that we can have more common methods and even implement a library, so that we can reduce the amount of repeated code and have a more clear and concise structure of the test suite.

2. Part B

Collaboration and Implementation

In part B of this project, we first need to write story tests for 10 user stories. For each of the 10 stories, at least three acceptance tests should be defined, which are normal flow, alternate flow and error flow. After a group meeting with all members attended, we divided 10 stories into 4 parts, which were 3 3 2 2, and each person was responsible for a part. We followed the example on MyCourses provided by the professor and a tutorial online[1] to write story tests as gherkin scripts, in feature files.

After everyone finished his own part of work, we had another meeting to discuss each one's acceptance tests and then moved on to the story test automation section. During this meeting, we tried to find a way to adjust our acceptance tests so that some of them could share the same function in the step definition library as described in the requirements. We also decided to use behave in python as our story test automation tool and each person was still responsible for the part he did in the previous section. After that we followed the online tutorial[1] and started to implement our tests. During the implementation process, we searched online for help and discussed in groups if we met problems. If someone finished his work early, he would share his implementation to provide others with an example. Before we integrated all four parts together, we did individual tests on each part first. When it was confirmed that all individual tests were passed, we integrated all four parts together. Due to the reason that each person used a slightly different method to write tests, bugs were encountered when doing an integrated test. We then analyzed the part that went wrong, debugged it and reran the integrated test until all tests passed. After all these were done, we revised the code structure so that it followed guidelines of Bob Martin on Clean Code. And then one group member would run this final version of code and record the whole process into a video.

Findings and Recommendations

For the test finding in this part, we found some problems including those already found in part A of this project, which were still affecting further tests because it was the problem of the server.

The first finding is that, for GET API request /projects/:id/tasks used in the 7th feature, when we send the API request using non-existing id numbers, the server should respond us with a message showing errors like 'could not find thing matching value for id' because the documented behavior is 'return all the todo items related to project, with given id, by the relationship named tasks', but the server just sends all the objects of todos in this case. This is not an expected normal behavior. The same situation applies for the request /categories/:id/todos used in the 8th feature, which is the second finding. The documented behavior is 'return all the todo items related to category, with given id, by the relationship named todos'. However, if we send this request with a random id number after adding some relationship between category and todo, all added todos will be returned. This again is not an expected behavior. The API requests we send and responses we get are shown in the following figures.

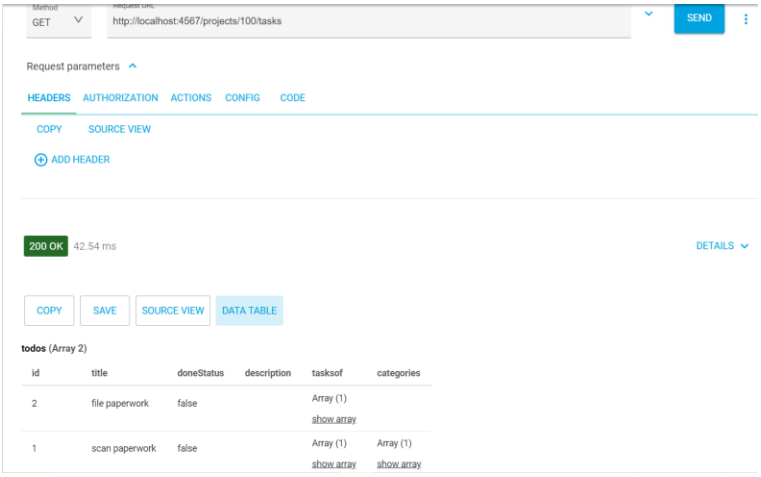

Figure 3: GET /projects/:id/tasks situation

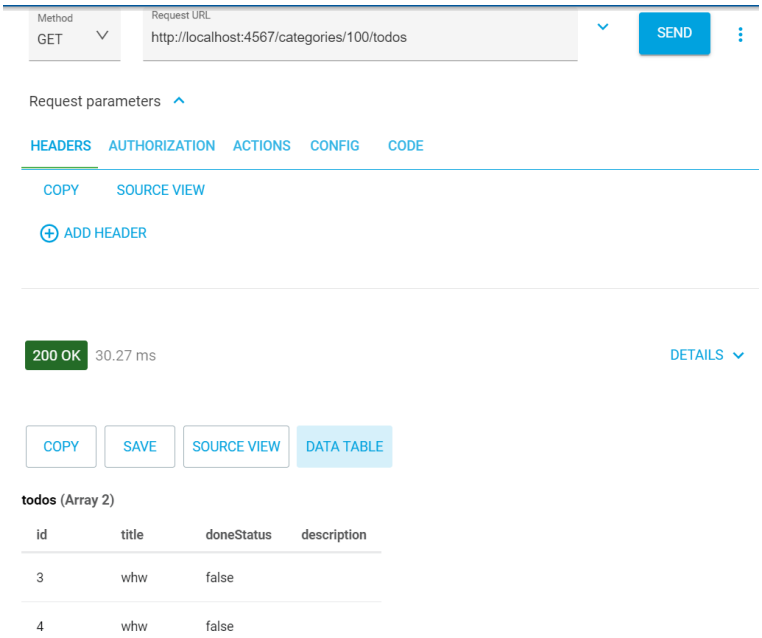

Figure 4: GET /categories/:id/todos situation

What is more, different from unit testing in part A, in story testing, every test execution depends on several methods. Those methods may be in the same file where test execution takes place, however, they may also locate in other files because all tests share a library. As a result, using global variables is very important. Our group decided to use the ‘context’ to store global variables because all step definitions are stored in the different python files separately.

The recommendation for part B of our project is to try to reduce the size of the library a bit more, because some stories are still related to each other, even if they are far apart in the user stories list provided. We should gather all the acceptance tests we write and find similarities between them. After further group discussion, we should conclude some neutral step definitions which more than one acceptance tests can use. In this way, we can efficiently reuse some step definitions and consequently reduce the size of the library.

3. Part C

Collaboration and Implementation

The part c of the whole project series has two parts: performance test and static analysis. Performance test checks how long each transaction api takes against real time and number of objects in the server backend. The testing script of measuring time is written in Python And the memory and cpu use against real time and number of objects in the server backend. Windows perfmon is used to measure the situation of the CPU and memory Static analysis uses a third-party tool Sonarqube to check the code quality of the server. After installing Sonarqube, a few steps would be followed to apply the code check inside the whole code project of the server.

The performance tests have several parts in the delivery: code work, graphs and video. This time, Yudong wrote the whole testing script. He uses Python as his programming language and uses the unittest and requests library as a tool to perform the testing process and send http requests to the server. Hanwen makes the graphs of transaction time, CPU use, Memory use against the real sample time and the number of objects in the server backend. Erdong is responsible for the video recording.

The static analysis has much less workload compared to the performance test. So weige is responsible for the whole static analysis. Use the Sonarqube to general a data report of the code quality of the server implementation. After that, the team will write his part in the report and check each other's work with the rubric.

Findings and Recommendations

For the performance test, check some graphs about transaction time, CPU use and memory use.
Transaction time:

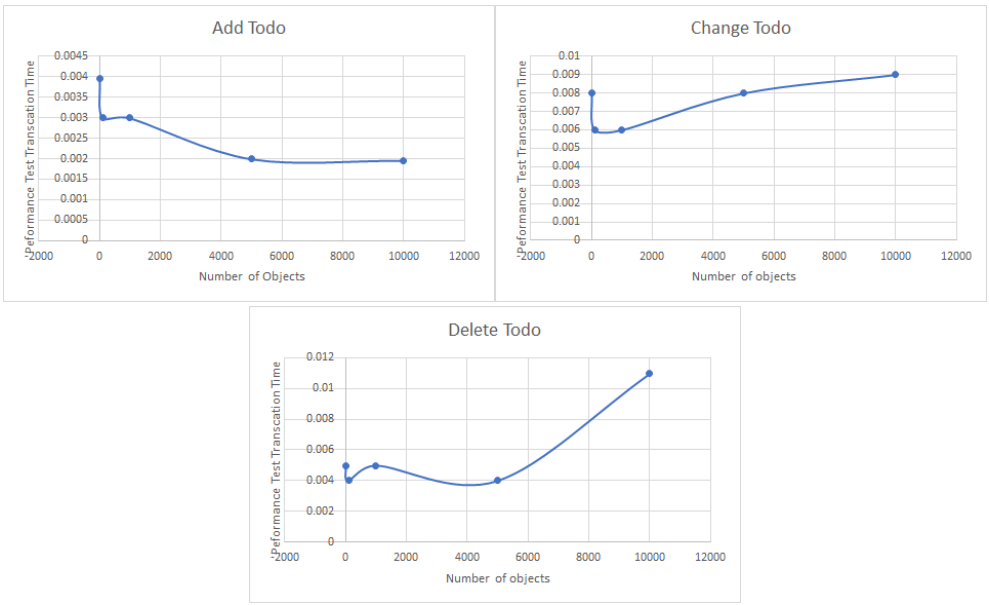

Chart 1: Transaction time vs sample time for Add, Change and Delete Todo

Since the space problem, I only use the graph of todos, but the trend of the other two is quite similar. It's obvious that the transaction time of the add function does not change a lot as the number of objects increases, I guess it is because the server just appends the new object without any logic order in it. In contrast, the transaction time of change and delete functions increase as the number of objects increases, since it needs the search algorithm which heavily depends on the number of objects.

For CPU use

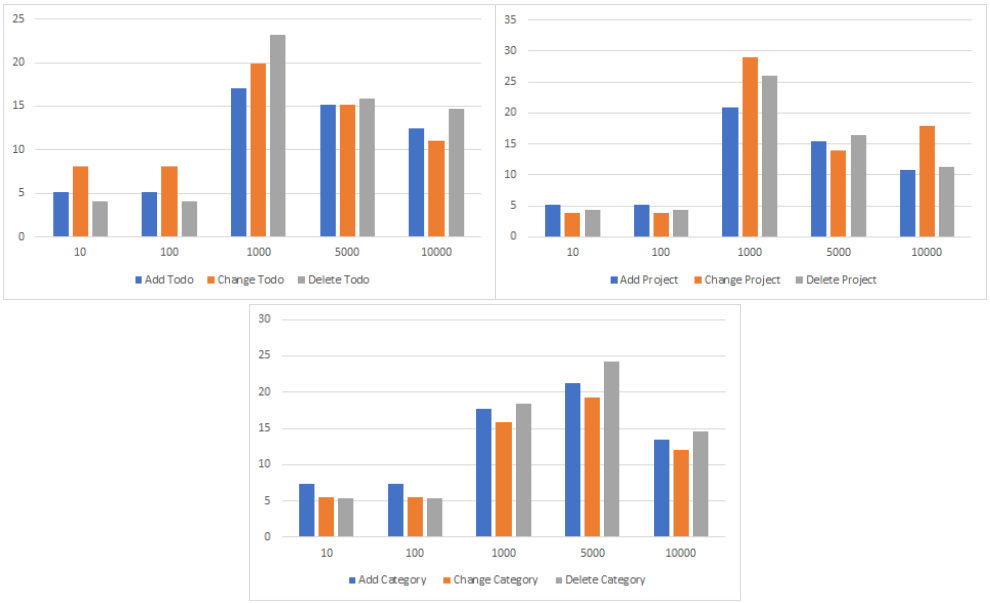

Chart 2: Memory available vs object number for Add, Change and Delete

It is obvious in the graph, the CPU use increases initially as the number of objects increases, but for Todo and project, after the number of objects is larger than 1000, the CPU use of three functions decreases. For the category, this number becomes 5000.

For the memory use

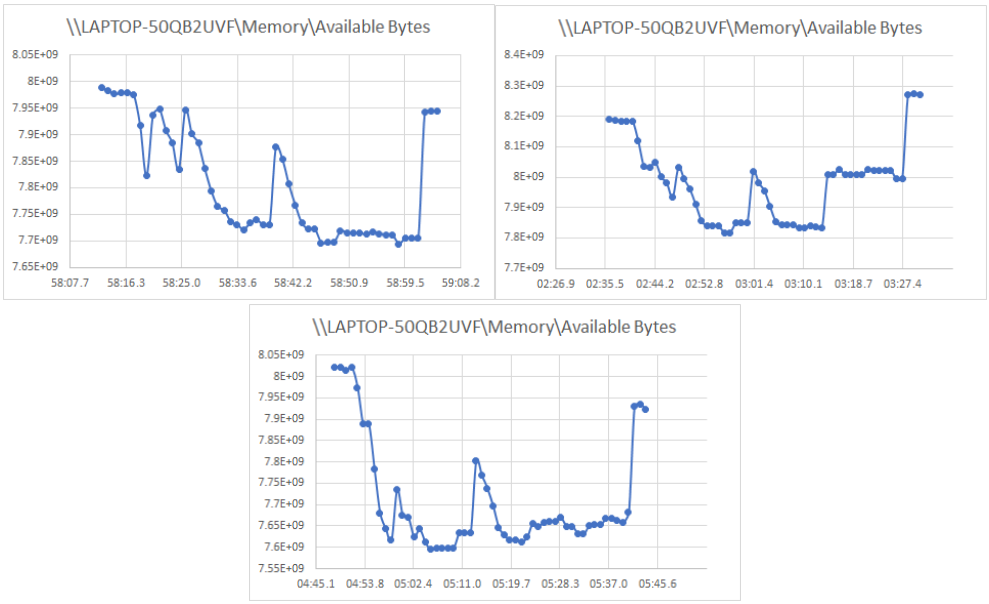

Chart 3: Memory available vs sample time for Add, Change and Delete Category

For the space problem, I just use the graph of the categories, we could find that the trend of this is pretty similar to the cpu use, the available bytes decrease first, but after a point, it will go back to origin again.

Static analysis

Figure 5: Screenshot of SonarQube

From the generated report from the Sonarqube, the code work of the server receives a great score on the maintainability and the security. But on the other hand, it has 8 bugs which is quite a serious problem. Also, it received an E on the Security Review. The code needs to improve on this part. Besides this part, we also need to add more unit test cases on the server side.

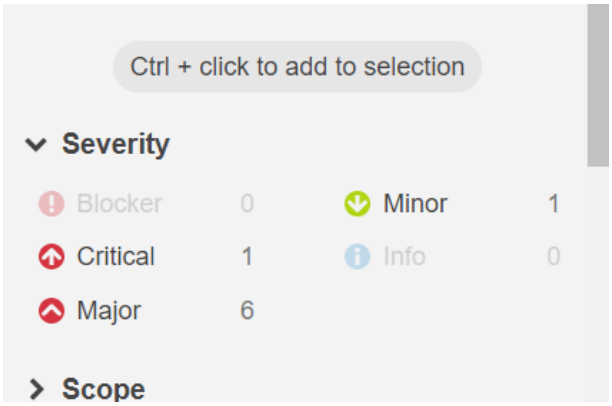

Figure 6: Screenshot of SonarQube

For the bugs' situation, there are 6 major bugs in the code job, most of them are on use == instead of equals to compare objects, and several possible NullPointerExceptions are not gracefully handled.

Reference

1. "Tutorial." behave. https://behave.readthedocs.io/en/latest/tutorial.html (accessed Dec. 6, 2020)