Qi Chen – 261053991

**Exploratory Testing**

For the exploratory testing phase, the following Charter was given and followed closely during each session:

* Identify capabilities and areas of potential instability of the “rest
* api todo list manager”.
* Identify documented and undocumented “rest api todo list manager”
* capabilities.
* For each capability create a script or small program to demonstrate
* the capability.
* Exercise each capability identified with data typical to the intended
* use of the application.

The focus was mainly on the capabilities related to the /todos endpoint and everything related to this endpoint, documented or not, like /todos/:id and /todos/:id/categories. There were two sessions of 45 minutes, each focusing on different endpoints and their capabilities with usual data, the concerns that arose during the session and some testing ideas.

**Session 1:**

The endpoints /todos and /todos/:id were tested with the following common Rest APIs methods that were both documented (in the docs website) and undocumented (discovered by the swagger file): GET, PUT, POST, DELETE, OPTIONS, HEAD and PATCH.

Endpoint /todos:

The documented APIs worked as intended. The undocumented OPTIONS was allowed, but was useless since no body was sent from the server. The undocumented methods not allowed were properly blocked from impactful actions. The HEAD method worked properly.

Endpoint /todos/:id:

The behavior was like the previous endpoint for the undocumented methods. However, the PUT API was found to behave differently from the POST method, despite both having the same documented behavior. The POST method modified the fields with the values provided in the request body while the PUT method completely overwrite the old entity with the new one provided in the request body.

Summary:

Overall, beside the POST and PUT problem found previously, the other documented methods worked as expected using usual data in a script that sent HTTP request to the server and printed out the response. The undocumented methods were either blocked from executing useful operations or useless such as OPTIONS. There are also weird quirks found in the server’s behaviors such as the inconsistent ordering of the todos instances after adding one or the fact that the todo’s ids are incremented even if a new todo failed to be created, meaning there will be a gap in the ids sequence for future todos.

There were concerns raised during the session like if the todos are properly discarded when the server shutdown as documented, how the server will behave if malformed bodies are sent to the server, are the inputs properly sanitized by the server and some inconsistencies were found in the error messages returned by the server.

The session led to testing ideas such as how the APIs react to bad inputs and wrong field types (verify validation), if the user inputs are properly sanitized, confirm the behavior discrepancy between POST and PUT and check if all blocked requests return the proper status code.

**Session 2:**

The endpoints /todos/:id/categories, /todos/:id/categories/:id, /todos/:id/tasksof and /todos/:id/ tasksof /:id were tested with the same Rest APIs methods that were both documented (in the docs website) and undocumented (discovered by the swagger file).

Endpoint /todos/:id/categories and /todos/:id/tasksof:

The documented APIs worked as intended. The undocumented OPTIONS was allowed but was useless like the previously tested endpoints. The undocumented methods not allowed were properly blocked from non authorized actions. The undocumented HEAD method worked properly.

Endpoint /todos/:id/categories/:id and /todos/:id/tasksof/:id:

DELETE was the only documented method and worked as intended. The other undocumented methods worked like the previous endpoints except HEAD, which was instead blocked.

Summary:

Overall, there were no flagrant problems with these APIs discovered in this session.

There were minor concerns like how the server will behave with very long inputs and if the non-allowed methods indicated by the swagger file were properly blocked and with the correct status code.

The session led to testing ideas such as how the APIs react to illogical requests (deleting non-existing entries/relations, adding the same relation twice, etc.) or unusual inputs (special characters, long text).

The sessions were complemented by a simple script that sent HTTP requests with the various method and with usual data. The state before and after the operation and the response itself were printed out to assess easily the behavior of the server from the request. Screenshots of the console log were recorded to complement the session notes.

**Unit Test Suite Structure**

Each test suite is testing a specific endpoint. For each API at this endpoint, a unit test that follows the expected path is executed to ensure the normal way is correct. Then, numerous erroneous tests or edge cases are executed and evaluated as if they will behave (usually fail) in the expected way according to the documentation and the swagger file for the undocumented ones (like getting a 404 error when deleting a non-existent relation). In the occasion that an API behave differently from the documented way, “AsDocumented” will be appended to the test name to indicate that the test is failing despite expecting the API to behave the same way as documented. A new test with the actual behavior with “Actual” appended to its name will be created, showing how the actual behavior is working. If the test fails for an undocumented behavior but that is expected (like wrongly getting a 404 not found error instead of a 405 method not allowed method according to the swagger file), the failing test will have “AsExpected” appended to its name and the actual behavior will be displayed in another unit test.

Before each test suite, a simple ping request is done to check if the server is available. If not, an exception is printed out and thrown, preventing the rest of the test suite to execute.

At the start of each test, a test instance of todo is created and it is saved into a variable so that it can be directly accessed by the unit test module (useful when we need to know the id of the test instance for example). If we are testing the relation between todo and another model, a relation will be created automatically so that we are able to test all the available APIs while maintaining the same test instance structure between each test of the same endpoint. The related model instance will then be saved into a variable.

At the end of each test, the todo test instance will be deleted, erasing at the same time all relations it has with the other models and the instance variables will be cleared.

Generally, for each tested API, the state before the executed method will be saved for side effects testing purposes (like keeping a list of category instances under a certain todo instance before deleting one). The saved state will be either the tested instance itself if its fields will be modified (PUT, POST to a specified instance), or a list the instances related to the tested instance (when testing POST to /todos for example).

Then, the test request itself will be executed with the configuration that is under test. The response will be saved in a variable for assertion later.

Afterwards, the state after the request will be requested to the server. If applicable, the before state will be modified internally to reflect the expected behavior of the tested request sent to the server (e.g., modifying the fields accordingly to what was sent in a body in a POST request to the test instance).

After the request is done, the unit test revert the system state to before the impactful operation if applicable (such as deleting a newly created instance of todo after a POST operation at /todos) by sending the proper requests to the server at the correct endpoints.

The unit test finishes with a list of assertions to ensure that the behavior is as expected. The status code and the body of the response given by the method under test are checked to see they are in line with the expected results and the updated before state and after state are compared to check if the new state is as expected and that no side effects was observed.

**Unit Tests Findings**

The biggest issue found from the unit tests is the discrepancy between the behavior of the POST and the PUT methods at /todos/:id despite having the same documented behavior as suspected during the exploratory testing. As proof, the test “putTestAsDocumented” of “todoTest” is failing when expecting the same behavior as “postTestSuccess”. While the POST method allows some fields to be modified by choosing specific fields to send in the request body, the POST method instead treats what is given in the request body as a completely new instance that will replace the old one. This is proved by the test “putTestActual” of “todoTest” that is succeeding when we are expecting this kind of behavior.

The other tests that failed uncovered some issues related to the status code returned by the server. Executing a GET, POST and HEAD request to /todos/:id/categories and /todos/:id/tasksof returned a 404 not found error instead of a 405 method not allowed error indicated by the swagger file. This problem seems systematic as the same APIs of those two endpoints indicated this issue and doesn’t seem intentional according to the swagger file and the fact that the 405 error is properly returned by the other non allowed methods for the same endpoints such as PUT.

**Source Code Repository**

URL: <https://github.com/qiy-chen/ECSE429SoftwareTesting>

The source code is a Maven project directory with the dependencies needed to properly execute the unit tests. The tested application is in the root folder and the script used for the exploratory test also resides here.

The unit tests are in the folder “test” under the parent folder “src” from the root. There is a helper class for common functions used in the unit tests like a function to convert a String Object that represents a JSON Object into a HashMap.

The deliverables are situated in the “Documents” folder. We can also find the swagger file that was used in the exploratory testing and unit tests to test the behavior of documented and undocumented APIs.

The unit tests are separated by the endpoint they are testing and, in each file, the unit tests that test the same method are grouped together with the one that is expected to succeed being the first.