ECSE 429 – PART A

Exploratory Testing and Unit Testing

Naomie Lo | naomie.lo@mail.mcgill.ca | 261018690

Deniz Emre | deniz.emre@mail.mcgill.ca | 261029931

DELIVERABLES OVERVIEW

The goal of this charter is:
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 deliverables below cover these four points testing the Thingifier software. In the first part Naomie and Deniz focused on Exploratory testing with postman. Following, we prepared extensive test using pytest to further our understanding of Thingifier’s capabilities.

Exploratory testing

1. **Todos**
   - Session Notes
   - Folder with screenshots
2. **Projects**
   - Session Notes
   - Folder with screenshots

Unit Tests

1. **Todos**
   - random_test.py
   - test_documented.py
   - test_undocumented.py
   - test_unexpected.py
   - test_payloads.py

2. **Projects**
   - random_tests.py
   - test_projects_documented.py

- test_projects_undocumented.py
- test_projects_unexpected.py
- test_payloads.py

Other Files

- Report
- Bug Summary Template

FINDINGS OF EXPLORATORY TESTING

Todos

Basic CRUD Operations:

The todos API mostly performed as expected during typical operations.

The GET /todos endpoint successfully returned the list of todos, including proper responses for both existing and non-existing project IDs (E |1.00, E |1.03, E |1.08, E|1.12).

The POST operation allowed for creating new projects without providing an ID, generating the next available ID instead.

The DELETE operation successfully deleted resources as long as the instance existed as expected.

Undocumented Capabilities:

We tested several undocumented capabilities, such as unsupported HTTP methods like DELETE, PUT, PATCH, and OPTIONS:

OPTIONS requests (/todos, /todos/:id) were accepted, and returned 200 OK status codes (U |1.17, U |1.20). While the server seems partially configured to handle these requests, they are not documented, potentially leading to unintentional use.

Other methods like PATCH and PUT on unsupported endpoints (/todos, /todos/:id/categories) returned 405 errors, indicating that they are not

allowed, but these behaviors were not mentioned in the documentation (U |1.17, U |1.18, U |1.16, U|1.21).

Inconsistent ID Generation:

ID values for new instances do not always follow a logical incremental order. The inconsistent ID generation may lead to confusion during data management and retrieval, particularly in systems that rely on sequential ordering of resource IDs.

XML and JSON inputs:

When attempting to specify an ID in either JSON or XML format during a POST request, an error is returned indicating that the syntax is invalid. This prevents the ability to manually set IDs when creating new resources, restricting control over resource identifiers.

Projects

Basic CRUD Operations:

The Projects API mostly performed as expected during typical operations:

The GET /projects endpoint successfully returned the list of projects, including proper responses for both existing and non-existing project IDs (E |1.01 - E |1.06).

The POST /projects endpoint allowed for creating new projects without providing an ID, generating the next available ID instead (E |1.03).

The PUT /projects/:id operation correctly updated project details (E |1.14).

Inconsistent ID Generation:

The ID generation for new projects and categories does not reset, even after deleting existing records. Instead, IDs are incremented

from the last value used across the entire /projects endpoint. For example, when adding categories to a new project, the next available ID is taken, leading to IDs such as 3 being assigned instead of restarting from 1.

IDs need to be specified as strings in JSON format. This behavior was consistent for JSON inputs; however, the API did not support creating IDs using XML, causing potential bugs or failure points (N |1.24, N |1.25). This inconsistency can lead to confusion, especially when attempting to use different formats.

Error Handling and Deletion Messages:

When attempting to delete resources (DELETE /projects/:id or /projects/:id/categories), there was no response message indicating success or failure (E |1.15, E |1.29).

Error messages were inconsistent when trying to link categories or tasks to non-existing projects (N |1.19, N |1.20). For example, attempting to get categories for a non-existing project sometimes returned the categories of another project rather than a proper error response.

Undocumented Capabilities:

We tested several undocumented capabilities, such as unsupported HTTP methods like DELETE, PATCH, and OPTIONS:

OPTIONS requests (/projects, /projects/:id) were accepted, and returned 200 OK status codes (U |1.33, U |1.35). While the server seems partially configured to handle these requests, they are not documented, potentially leading to unintentional use.

Other methods like PATCH and PUT on unsupported endpoints (/projects, /projects/:id/categories) returned 405 errors, indicating that they are not allowed, but these behaviors were not mentioned in the documentation (U |1.31 - U |1.37).

STRUCTURE OF UNIT TEST SUITE

We decided to split the tests we did with postman into 4 categories: documented, undocumented, payload, and unexpected. In the file, test_documented.py has the tests for everything in the documentation as well as tests with data a user would likely enter. The later was to test if the documented data work with edge cases we thought of such as trying to delete an instance that does not exist. Within the test_undocumented.py has the capabilities that are not documented such as PUT, OPTIONS, PATCH, and DELETE. The test_payload.py focus is checking both the returned code and application logic of JSON and XML formats. The test_unexpected has both the tests that returned unexpected values and a test module that will pass with the unexpected behavior.

While creating these tests we had to main focuses, to ensure that the initial state of the API is restored at the end of the tests and that there is an ability to run the tests in random order. For the first point, after each test the resource is immediately deleted. Regarding randomization, this was a two-part task. First, no module should rely on the ordering. For example, when testing a DELETE API call, we need to ensure the instance that we are deleting exists. To avoid dependencies, a resource is created for each module. A helper function, create_todo() will create a todo instance to perform the test on. The second part of randomization is calling all the modules in random order. A python script, random_test.py accomplished this. The logic follows importing all the tests and using random.shuffle() while iteratring through them.

The unit tests for project and todos are done separately but follow the same logic. The todos unit tests are in PART_A/todos/tests and the projects unit tests are located in PART_A/projects/tests.

SOURCE CODE REPOSITORY

This testing was documented on GitHub. We chose to use this as it offers the ability for many users to collaborate on a code base. To best organize the project, we have three main folders names “Part A”, “Part B”, “Part C”. The contents of the exploratory test and unit tests live in Part A. With the Exploratory Test folder is the two testing focuses, todos and projects. These folders have the session notes and the respective screenshots from the timed session. A test folder also resides within the todos and projects subfolders. This contains the test files for the respective focus. Please see layout below.

- ➔ *PART A*
  - ➔ Bug Summary
    - bug_summary_temp.txt
  - ➔ Todos
    - Session_Notes.txt
    - ➔ Screenshots
      - 1.05.png
      - 1.07.png
      - ….
    - ➔ Tests
      - random_test.py
      - test_documented.py
      - test_undocumented.py
      - test_unexpected.py
      - test_payloads.py
  - ➔ Projects
    - Session_Notes.txt
    - ➔ Screenshots
      - 1.05.png
      - 1.07.png
      - ….
    - ➔ Tests
      - random_test.py
      - test_projects_documented.py

- test_projects_undocumented.py
- test_projects_unexpected.py
- test_payloads.py

FINDINGS OF UNIT TEST SUITE

Todos

Documented Capabilities:

The GET, POST, PUT, and DELETE methods for /todos and /todos/:id generally worked as expected with typical inputs.

Undocumented Capabilities:

The results from the unite tests on undocumented capabilities aligned with those seen using postman in the exploratory test.

Unexpected Behavior:

These tests focused on verifying resource creation, ID handling, and response behaviors. They highlighted inconsistencies in ID generation for categories linked to todos and issues with handling numeric and string IDs during POST requests. Additionally, GET requests for nonexistent or invalid todos returned unexpected responses, sometimes allowing a 200-status code instead of the expected 404. The overall findings suggest gaps in error handling, ID management, and response consistency, which can lead to confusion and uncertainty in using the API for reliable data management.

Payload Testing:

The tests validated the API's handling of XML and JSON formats, focusing on both valid and malformed data submissions. Malformed JSON and XML POST requests returned the expected 400 Bad Request responses, while valid XML submissions were successfully processed. Overall, the findings indicate that the API correctly distinguishes

between valid and invalid formats, ensuring robust data validation and integrity.

Projects

Documented Capabilities:

The GET, POST, PUT, and DELETE methods for /projects and /projects/:id generally worked as expected with typical inputs. However, when operations were performed on non-existent resources, responses like 200 OK instead of 404 Not Found were observed, indicating inadequate error handling.

Undocumented Capabilities:

Some HTTP methods, such as PATCH and PUT on /projects/:id/categories, correctly returned 405 Method Not Allowed, but this should be explicitly mentioned in the documentation. The OPTIONS requests returned 200 OK without providing useful information.

Unexpected Behavior:

ID Management: Category IDs increment globally across all projects, rather than resetting per project. This could create difficulties in managing resources.

Invalid IDs: Attempts to access categories or tasks with invalid project IDs often returned 200 OK, indicating improper error handling.

String vs. Numeric IDs: The API accepted string IDs but failed with numeric IDs, showing inconsistency in ID input handling.

Payload Testing:

The API correctly returned 400 Bad Request for malformed JSON or XML, showing basic validation.