ECSE 429 Software Validation
Part A: Exploratory Testing of REST API
Written Report

Emile Labrunie (261097953) – emile.labrunie@mail.mcgill.ca
Ethan Wu (261117309) – ethan.wu2@mail.mcgill.ca
Kenny Duy Nguyen (261120429) – kenny.nguyen@mail.mcgill.ca
Yanzhe Zhang (261016377) – yanzhe.zhang@mail.mcgill.ca

Winter 2026

Contents

1 Introduction

This project evaluates the correctness and reliability of a REST-based Todo List Manager application through structured validation techniques. In Part A, we applied charter-driven exploratory testing to identify documented and undocumented behaviors, followed by the development of an automated unit test suite to formalize and verify those findings.

The objective of this phase was not only to confirm expected API behavior, but also to uncover inconsistencies, edge cases, and systemic weaknesses. The insights gained in Part A provide the technical foundation for the story-based behavioral validation performed in Part B.

2 Summary of Deliverables

The deliverables for Part A are as follows.

2.1 Exploratory Testing Session Deliverables

Deliverables from each charter-driven exploratory testing session include:

- Session notes containing: summary of session findings, list of concerns, and list of new testing ideas;
- Any files created during the session (including Bug Summary Reports where applicable).

We performed **5 exploratory testing sessions**: Sessions 1–3 by Kenny Nguyen and Ethan Wu; Sessions 4 and 5 by Emile Labrunie and Yanzhe Zhang.

The session notes are:

- Session 1 – Kenny and Ethan (Projects entity, JSON)
- Session 2 – Ethan and Kenny (Todo–Project relationships)
- Session 3 – Ethan and Kenny (Todo–Category and Project–Category relationships)
- Session 4 – Emile and Peter (Todo entity)
- Session 5 – Emile and Peter (Category entity)

2.2 Unit Test Suite

- At least one unit test module for each documented API (todos, projects, categories in JSON and XML);
- At least one module for each undocumented API (e.g. `/todos/tasksof`, `/todos/categories`, `/projects/tasks`, `/projects/categories`, `/categories/todos`, `/categories/projects`);
- Tests for malformed JSON and XML payloads;
- Tests for invalid operations (e.g. deleting an already-deleted resource);
- A test ensuring the suite fails when the service is not running.

2.3 Bug Summaries

Bug summaries use a form with executive summary (80 characters or less), description, potential impact, and steps to reproduce. Reports are in `partA/bugs/`.

2.4 Unit Test Suite Video

Two videos of all unit tests run in random order are in `partA/documentation/` (Unit Test Run Order 1 and 2).

3 Findings of Exploratory Testing

3.1 Projects Entity (Session 1: Kenny and Ethan)

This session focused on validating the `/projects` endpoints using JSON. We exercised CRUD operations, HEAD support, invalid operations, and a basic relationship query.

Collection Endpoint: `/projects`

- **GET** returned the list of current projects (`200 OK`).
- **HEAD** returned headers successfully (`200 OK`).
- **POST** with an empty body returned `201 CREATED`. Default values were assigned to `id`, `completed`, and `active`, while `title` and `description` were blank.
- **POST** with specified title and description also returned `201 CREATED`.
- Attempting to create a project with an explicit ID returned `400 BAD REQUEST` with the message: *"Invalid creation: Not allowed to create with id."*
- **PUT** `/projects` (no ID specified) returned `405 METHOD NOT ALLOWED`.
- **DELETE** `/projects` (no ID specified) returned `405 METHOD NOT ALLOWED`.

Resource Endpoint: `/projects/:id`

- **GET / HEAD**
  - Valid ID → `200 OK`
  - Non-existent ID → `404 NOT FOUND`
- **POST / PUT** successfully updated title, description, and active status for existing projects (`200 OK`).
- **PUT** with invalid ID → `404 NOT FOUND`.
- **DELETE**
  - Valid ID → `200 OK`
  - Non-existent ID → `404 NOT FOUND`

Relationship Check

- **GET** `/projects/:id/categories` returned an empty list when no categories were linked (`200 OK`).

Session Outcome

All documented CRUD behaviors for the Projects entity were confirmed. No defects were identified during this session. The tested capabilities are reproducible via `projectsCapabilities.sh`.

3.2 Todo–Project Relationships (Session 2: Ethan and Kenny)

This session examined relationship endpoints between Todos and Projects: `/todos/:id/tasksof` and `/projects/:id/tasks`.

Valid JSON Behavior

- **POST** with valid IDs created relationships (`201 CREATED`).
- **GET** and **HEAD** returned expected data for valid IDs (`200 OK`).
- **DELETE** successfully removed relationships.

Invalid-ID Defects Identified

- **GET** with a non-existing ID returned `200 OK` and relationship data instead of `404 NOT FOUND`.
- **HEAD** with invalid IDs also returned `200 OK`.

This behavior was observed for both: `/todos/:id/tasksof` and `/projects/:id/tasks`.

XML Support Issue

- Relationship creation using XML (e.g., `<id>1</id>`) returned `400 BAD REQUEST` due to parsing failure.

Undocumented Endpoint

- **GET** `/projects/tasks` (no project ID) returned relationship data (`200 OK`), though not documented.

Capabilities are demonstrated in `projectsCapabilities.sh` and `todosCapabilities.sh`.

3.3 Todo–Category and Project–Category Relationships (Session 3: Ethan and Kenny)

This session explored interoperability across: `/todos/:id/categories`, `/categories/:id/todos`, `/projects/:id/categories`, and `/categories/:id/projects`.

Valid JSON Behavior

- Relationship creation (POST), retrieval (GET), and deletion (DELETE) worked correctly for valid IDs.

Repeated Invalid-ID Defect

- **GET** and **HEAD** with non-existing IDs returned `200 OK` and relationship data instead of `404 NOT FOUND`.

This defect pattern was consistent across all four relationship types.

XML Limitation

- Relationship creation in XML failed (`400 BAD REQUEST`).

Undocumented Aggregate Endpoints

The following endpoints (without an ID) returned relationship lists:

- `/todos/categories`
- `/categories/todos`
- `/projects/categories`
- `/categories/projects`

All returned `200 OK`.

Capabilities are demonstrated in `todosCapabilities.sh`, `projectsCapabilities.sh`, and `categoriesCapabil`

3.4 Todos Entity (Session 4: Emile and Peter)

This session focused on validating `/todos` endpoints.

Collection Endpoint

- **GET** `/todos` and **HEAD** returned expected results.
- **POST** required a valid body; missing body resulted in `400`.
- Valid JSON payloads successfully created todos.

Resource Endpoint

- **GET** / **HEAD** `/todos/:id` returned `200 OK` for valid IDs and `404 NOT FOUND` for invalid IDs.
- **POST** and **PUT** `/todos/:id` updated existing todos successfully.
- Update operations behaved consistently in both JSON and XML.
- **DELETE** removed valid todos; repeated deletion returned `404`.

Observed Design Concern

- **POST** `/projects/:id` updates existing projects similarly to PUT. This may cause unintended overwrites if clients expect POST to create resources only.

Capabilities are demonstrated in `todosCapabilities.sh`.

3.5 Categories Entity (Session 5: Emile and Peter)

This session validated `/categories` endpoints.

Collection Endpoint

- **GET** and **HEAD** `/categories` behaved as expected.
- **POST** worked in both JSON and XML with required fields.

Resource Endpoint

- **GET / HEAD** `/categories/:id` returned `200 OK` for valid IDs and `404 NOT FOUND` otherwise.
- **POST** and **PUT** updated categories correctly in JSON and XML.
- **PUT** without body returned an error (as expected).
- **POST** with valid ID but no body returned `200 OK` without modification (noted as inconsistent behavior).

Validation Rules

- Empty or whitespace-only title rejected (`400`).
- Single-character title accepted.

Additional Endpoints

- **GET** `/docs` returned API documentation.
- **GET** `/shutdown` terminated the server.

Capabilities are demonstrated in `categoriesCapabilities.sh` and `categoriesXMLCapabilities.sh`.

Cross-Session Defect Summary

Across exploratory sessions, three recurring defect patterns were observed:

- Relationship endpoints return 200 OK for non-existent entity IDs.
- Relationship creation via XML consistently fails with 400 BAD REQUEST.
- POST /projects/:id updates resources similarly to PUT, creating ambiguity.

These defects were later encoded explicitly in the unit test suite.

4 Source Code Repository

We chose to use Google Drive for all documents and notes that are word docs and PDF files, and the a GitHub repository for everything code related. GitHub repository structure:

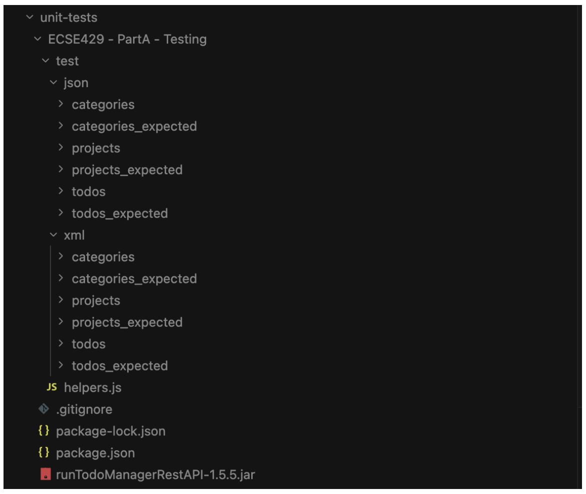
- `partA/exploratory-testing/` – Session notes and session outputs.
- `partA/scripts/` – Capability scripts (todos, projects, categories in JSON and XML).
- `partA/unit-tests/` – Full unit test suite (Mocha/Chai).
- `partA/bugs/` – Bug form and reports.
- `partA/documentation/` – Videos and this report.

Root `README.md` describes objectives and directory-to-requirements mapping.

5 Structure of Unit Test Suite

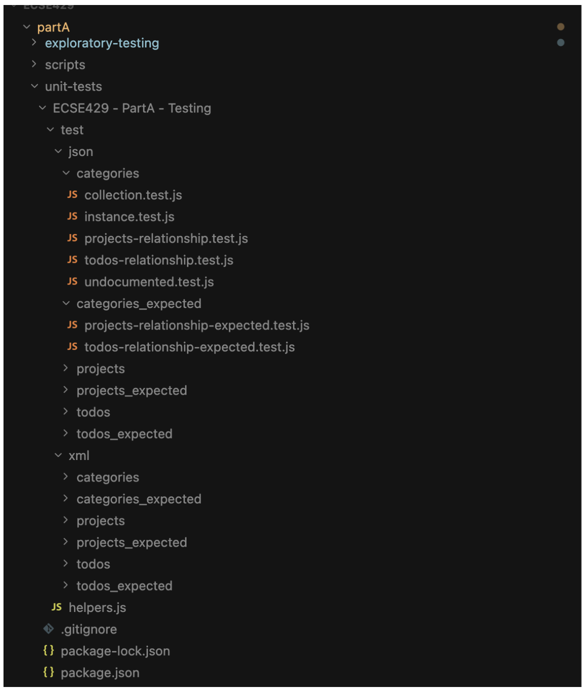
The unit test suite is structured as follows.

Each endpoint or set of endpoints relating to a specific entity is given its own unit test module, or file, under a folder that encapsulates all the endpoints used for that specific entity. Furthermore, the unit tests are broken down by format of requests and responses. JSON unit tests are grouped together under the `test/json/` folder, and the same goes for unit tests in XML format, which are under the `test/xml/` folder. Under the JSON directory (and similarly under XML), there are also modules for undocumented endpoints that were discovered during test sessions. Figure 1 below summarizes our directory structure.



```
unit-tests
  ECSE429 - PartA - Testing
    test
      json
        categories
        categories_expected
        projects
        projects_expected
        todos
        todos_expected
      xml
        categories
        categories_expected
        projects
        projects_expected
        todos
        todos_expected
      helpers.js
    .gitignore
    package-lock.json
    package.json
    runTodoManagerRestAPI-1.5.5.jar
```

(a) directory structure



```
partA
  exploratory-testing
  scripts
  unit-tests
    ECSE429 - PartA - Testing
      test
        json
          categories
            collection.test.js
            instance.test.js
            projects-relationship.test.js
            todos-relationship.test.js
            undocumented.test.js
          categories_expected
            projects-relationship-expected.test.js
            todos-relationship-expected.test.js
          projects
          projects_expected
          todos
          todos_expected
        xml
          categories
          categories_expected
          projects
          projects_expected
          todos
          todos_expected
        helpers.js
      .gitignore
      package-lock.json
      package.json
```

(b) expanded directory structure

Figure 1: Directory structure of the unit testing project.

Each file confirms the operation of each of the HTTP methods supported by the API by passing in typical information and data in the requests that would yield correct results. Furthermore, invalid and error scenarios are tested such as: malformed JSON/XML bodies which either have an attribute missing or an extra random attribute not supported by the API, requests made with non-existing IDs of entities, and invalid sequence of operations that is meant to throw an error, such as requesting to delete something twice in a row.

The unit tests are independent of each other, as before and after each unit test, the server is restarted and shut down. Our unit tests are coded in JavaScript using two testing frameworks and libraries: Mocha and Chai. Mocha allows a seamless testing process with useful before and after hooks to make our unit tests independent of each other. Chai was used for test assertions to confirm the behaviour of the API is as we expect. Axios was used to make HTTP requests to the API.

As for how to set up the repository, `npm install` must first be run inside the directory on the command line to install all dependencies of our project. Then, using the `npm test` command, unit tests will run; to run them in a random order each time (to showcase their independence), use `npm run test:random`. A seed may be logged to the console when this command is run; it can be saved and used to run the unit tests in the same order again. Otherwise, calling the command again will run the tests in a different randomly generated order. To run only a specific set of unit tests which have been grouped together in a file, the command to run is `mocha ./pathToTestFile` after having navigated to the root directory of the project.

For bugs, our unit tests will pass since we assert the existence of the buggy behaviour. For each unit test, it is described in plain English and can be labelled as "Capability", "Error Case", or "BUG". A short explanation of the labels:

- **Capability:** indicates this unit test is confirming normal API behaviour that is typical usage and will lead to success return codes—no error messages due to invalid data passed through, or buggy behaviour that is not supposed to happen.
- **Error Case:** indicates this unit test is confirming the correct behaviour of the API to return error messages and error codes when a user passes in invalid data or invalid requests.
- **BUG:** indicates this unit test is confirming the presence of a bug—an unhandled error case by the API, or behaviour not conforming to the documentation. This unit test is accompanied by comments at its beginning to summarize expected versus actual behaviour being tested and confirmed.

6 Findings of Unit Test Suite Execution

Our unit test suite confirmed our findings from the exploratory testing sessions. The tests that assert normal API behaviour (CRUD and HEAD on todos, projects, and categories in both JSON and XML) pass when given valid data and typical usage. The suite also confirms that relationship creation via XML body fails as observed during exploratory testing: the API returns 400 and the tests assert this actual behaviour. For relationship endpoints called with non-existent entity IDs, the suite passes by asserting the current (buggy) response: GET and HEAD often return 200 OK and data or headers instead of 404. Separate modules (the "expected" tests) encode the documented, correct behaviour and fail under the current API, making the discrepancy between expected and actual behaviour explicit.

In addition, we tested malformed payloads in JSON and XML, such as an extra attribute not supported by the API or a missing required attribute (e.g. missing title). We found that these error cases are handled by the API with an appropriate error message and a 400 status code. The documentation does not describe this wide range of invalid requests in detail, so the decision during exploratory sessions to cover these cases in the unit tests was useful. Invalid operations were also tested—for example, deleting an already-deleted resource or requesting a relationship for a non-existent entity—and the suite confirms that the API returns the expected error responses (e.g. 404) in these cases. A dedicated test ensures that the suite fails when the REST API service is not running, as required.

The unit tests are run in random order (e.g. via `npm run test:random`) and pass consistently, demonstrating that each test is independent and that the server is correctly started and shut down before and after each run.

We also observed some inconsistency in error message formats and phrasing for similar error cases and methods. For example, when updating a todo entity with a non-existing ID, using POST and using PUT can yield different error messages (e.g. "No such todo entity instance with GUID or ID 25 found" versus "Invalid GUID for 25 entity todo"). The expected error messages for different API requests are not documented verbatim, so this cannot be strictly classified as a functional bug, but it does go against best practices of unifying error messages across the API.

Conclusion

Part A provided a structured validation of the Todo List Manager REST API through exploratory testing and a comprehensive unit test suite. Core CRUD and HEAD operations for todos, projects, and categories were confirmed to function as documented, while recurring issues were identified in relationship endpoints, particularly when handling non-existent entity IDs and XML payloads. These defects were formalized in the unit tests to ensure reproducibility and clarity between documented expectations and observed behavior.

Having validated the API at the endpoint level, we now extend our focus in Part B toward user-centered behavioral testing. Instead of testing isolated operations, we evaluate complete user workflows through story-based acceptance tests. This shift allows us to assess not only correctness of individual endpoints, but also the overall value and consistency delivered to users when interacting with the system as a whole.