Helena Clifford (261104576)
Léanne Ricard (261118883)

Written Report ESCE Project: Part A

Summarizes deliverables.

The deliverables for the first part of this project include exploratory testing session notes and summaries, which document the observations made during manual testing using Postman to analyze API behavior. Additionally, a complete set of unit test modules has been developed for the explored API endpoints, ensuring automated validation of expected functionality and error handling. A bug report form has been created to highlight inconsistencies and unexpected behaviors encountered during testing (https://forms.gle/TR5fZo1DskRfGPLp6). A video demonstration has been created to showcase the testing process and key findings. Finally, this written report compiles all findings, methodology, and analysis, providing a structured overview of the work completed.

Describes findings of exploratory testing.

The exploratory testing of the API Todo List Manager revealed both expected and unexpected behaviors across various endpoints. Documented endpoints mostly functioned as intended, but several exhibited inconsistencies in response messages, error handling, and required inputs. This was the case in both entities explored (Todos and Projects). The testing was conducted using Postman, allowing us to inspect API responses firsthand before implementing unit tests. This approach helped us identify undocumented behaviors, clarify required inputs, and understand the expected structure of responses, before crafting our unit tests.

Todos

Notably, Get/todos/:id endpoint correctly returned specific todo items, the Delete/todos/:id endpoint unexpectedly produced a 404 Bad Request instead of successfully deleting items. Similarly,

the Post/todos request failed due to a missing required field, despite documentation suggesting otherwise. These inconsistencies suggest gaps between expected functionality and actual implementation.

Undocumented endpoints also presented notable issues. Some accepted requests without proper validation or proper error handling, leading to unpredictable outcomes. For instance, certain methods, such as Put and Patch, were neither documented nor explicitly restricted, yet they produced varying results depending on the payload structure. The lack of proper feedback for failed requests, such as missing error messages or unclear responses, further complicated testing and troubleshooting.

Beyond identifying technical inconsistencies, the testing session also highlighted areas for improvement in API usability and robustness. There is a need for better documentation to clarify expected behaviours, input requirements, and supported methods. Additionally, error handling should be more consistent across endpoints to ensure users receive clear and actionable responses when requests fail.

Projects

When exploring the projects api endpoints, we noticed GET requests consistently returned valid data when directed at existing projects, tasks, and categories, while POST requests successfully created new entities and relationships between them. DELETE also worked as intended in most cases, removing resources when properly structured requests were made. However, despite these expected functionalities, the testing process uncovered inconsistencies in error handling, response messaging, and validation mechanisms.

One major inconsistency was the way the API handled errors across different endpoints. Some undocumented endpoints like GET and POST requests returned a 404 Not Found error, while others responded with 405 Method Not Allowed, creating confusion about whether the requested endpoint existed or was merely restricted. Additionally, DELETE did not explicitly validate dependencies, meaning that removing a project did not always confirm whether related tasks and categories were also removed. In some cases,

POST requests succeeded but provided no response message, forcing testers to make a follow-up GET request to verify whether the resource was actually created.

The behavior of responses varied significantly across methods, adding to the difficulty of predicting system behavior. HEAD requests, for example, often returned 200 OK but failed to include meaningful headers or response bodies, making them less useful than expected. OPTIONS requests succeeded even when used on undocumented endpoints, raising concerns about whether the API was unintentionally allowing methods that should have been restricted. Such inconsistencies in response handling make it harder for users to rely on predictable patterns when interacting with the API.

Another area of uncertainty involved the creation and management of relationships between projects, tasks, and categories. Initially, it was unclear what attributes were necessary to link these entities. Further exploration revealed that while projects and categories were assigned incremental IDs, the sequence was not always predictable—some categories appeared to reset their ID count per project, while others continued from the last generated ID system-wide. POST requests with different ID formats also yielded inconsistent results: a numeric ID resulted in a 404, whereas using a string ID was successful, suggesting inconsistent validation rules.

To enhance usability, the API would benefit from improved documentation that clearly defines input requirements, expected responses, and validation rules for all endpoints. Error handling should be standardized to ensure predictable responses, particularly for missing resources and unsupported requests. Strengthening validation checks would help prevent unintended behaviors when requests contain unexpected formats. Finally, response messages should be made more informative to reduce the need for additional requests to verify whether an operation was successful. Addressing these areas would significantly improve the clarity and reliability of the API for developers.

Describes structure of unit test suite.

The unit test suite was developed in a way that maps one method (e.g. /projects or /todos/id) to one file. The tests can be run in order or randomly by using the following commands in the terminal.

- Set ordering: pytest test_todos_id_tasksof.py -v
- Random ordering: pytest test_projects_id_tasks.py -v --randomly-seed=42

todos

The Todos folder has six files containing 1-2 testing codes for each endpoint. They test a success condition and a failure condition for each. The files are:

- tests_todos.py
- tests_todos_id.py
- tests_todos_id_tasksof.py
- tests_todos_id_tasksof_id.py
- tests_todos_id_categories.py
- tests_todos_id_categories_id.py

Each file has between 8 and 14 methods. The setup function at the start of each file ensures that the system and API are ready for testing before we run the tests.

We run tests for both the documented and undocumented endpoints. Then, there are extra tests for unexpected behaviour or edge cases of the API to ensure robustness.

There is also a test_summary function that tracks the success and failure counts of the executed tests. It allows for the option to execute the tests in a randomized order, with a fixed seed for it to be reproduced. After execution, a summary of the results is displayed. When running the tests, the script determines whether to execute them in the default order or randomly. It also provides an option to shut down the API afterward, ensuring that the test environment is managed.

Projects

The projects folder contains six files containing complete testing code for each endpoint. The files are the following:

- test_projects.py
- test_projects_id.py
- test_projects_id_categories.py
- test_projects_id_tasks.py
- test_projects_id_categories_id.py
- test_projects_id_tasks_id.py.

Within one file, the structure is similar, there first is the function ensuring the system is ready, then the methods that do not pertain to tests specifically but are used often within the tests. Then the tests, of which there are between 9 and 16, are split into different categories, the first being the documented endpoint tests, then the undocumented ones and finally the extra tests that pertain to unexpected behaviors from the API.Next, the test_summary function runs a set of predefined tests, tracking their success and failure counts. It includes an option to execute tests in random order using a fixed seed for reproducibility. The script ensures the API is running before execution and provides a summary of results. When run as a standalone program, it determines whether tests should run in a default or randomized order, executes them, and optionally shuts down the API afterward. This structure ensures comprehensive validation while allowing flexibility in test execution.

Describes source code repository.

The repository is divided into two main folders: Todos_tests and Projects_tests, each dedicated to testing their respective entities. Within each folder, every route (/todos, /todos/id, /projects/id/categories/id, etc.) has its own test file to maintain organization and modularity. Each test file contains unit tests written using pytest and requests, ensuring that individual API functionalities are thoroughly validated. Additionally, helper functions are included to create and delete test data dynamically, preventing test dependencies. A summary script is also included to run all tests in a structured or randomized order, providing clear execution results. The

repository follows a structured naming convention for clarity and ease of maintenance.

Describe findings of unit test suite execution.

todos

Running the pytest suite for todos provided valuable feedback regarding the reliability and performance of the API. The tests successfully confirmed that the core operations- such as fetching todos, deleting them, and managing their relationships with categories and tasks- are functioning as intended, with most tests passing without issue.

Some issues surfaced during testing, particularly with the Post method. Several tests that sent seemingly correct data resulted in 400 Bad Request responses. This suggests that there may be inconsistencies in how input data is validated or processed on the backend. Similarly, Patch and Put requests often returned 405 Method Not Allowed instead of 404 Not Found, pointing to a potential misalignment in how unsupported HTTP methods are managed.

The Delete method was another area where unexpected results occurred. While some Delete requests to remove relationships between todos and categories were successful, the responses did not always provide confirmation of these changes. This inconsistency raised concerns about the completeness of the operation.

Requests like Head and Options didn’t behave as anticipated. Instead of returning 405 Method Not Allowed for unsupported actions, they often returned 200 OK. This difference in expected result and actual result can lead to confusion regarding the handling of unsupported methods, making it harder to predict the outcome of such requests.

In summary, while the API for todos mostly functions as expected, there are areas that need refinement. Improving the handling of unsupported methods, standardizing response codes

and ensuring accurate feedback in responses would greatly enhance the API’s usability and reliability.

projects

The execution of the unit test suite provided valuable insights into both the stability of the API and the effectiveness of the test coverage. The suite was structured to validate both documented and undocumented endpoints, ensuring that expected behaviors were met while also identifying inconsistencies. Most tests passed successfully, confirming that core functionalities such as retrieving, creating, and deleting projects, tasks, and categories worked as expected.

However, several tests failed due to discrepancies between expected and actual API behavior. In particular, some requests that were anticipated to return 404 Not Found for missing resources instead returned 200 OK with empty arrays, indicating a difference in how the API handles non-existent entities. Similarly, certain methods, such as DELETE and PATCH, returned 405 Method Not Allowed rather than 404 Not Found, highlighting an inconsistency in how unsupported endpoints are managed.

Another area of concern was the API’s handling of response bodies. Some POST requests succeeded but returned empty response bodies instead of providing confirmation messages or newly created resource details. This lack of feedback required additional GET requests to verify that the operation had succeeded, reducing efficiency. Additionally, HEAD requests, which should ideally return metadata, frequently failed to provide headers, limiting their utility.

The results of the test suite suggest that while the API's fundamental capabilities are functional, there is room for improvement in standardizing responses and error handling. Refining the validation mechanisms and ensuring that failed requests return consistent status codes would enhance predictability. Moreover, improving response messages to include meaningful feedback would reduce the need for redundant verification requests. These refinements would contribute to a more reliable and user-friendly API.