ECSE Software Validation Term Project

Report

Part A Exploratory Testing of Rest API

McGill University
Winter 2023

Rania Ouassif
260861621
rania.ouassif@mail.mcgill.ca

1. Introduction

This report summarizes the deliverables of a project aimed at studying the "rest API todo list manager" application, with a focus on the capabilities related to todos. It includes a description of the findings from the Charter Driven Session Based Exploratory Testing, the structure of the unit test suite implemented, the source code repository used, and the results of the unit test suite execution. The goal of this report is to provide a comprehensive overview of the testing efforts performed on the application.

2. Deliverables summary

This first project includes the following deliverables:

- *Session*_Notes.xlsx, which include two sheets for the two performed exploratory testing sessions. On each page, the session details are provided (participant, session date, session length, testing environment), along with the session notes (explicitly referencing scripts, and screen shots), session summary, concerns and new tests ideas.
- *session1_commands.sh* and *session2_commands.sh,* contain all the curl commands generated by Insomnia during session 1 and session 2 from exploratory testing, respectively.
- Screenshots folder, which includes two folders: Session1 and Session2 include all the referenced screenshots taken during the exploratory sessions.
- *src* folder containing different tests classes, which are further detailed in the Unit Test Suite section.
- *UnitTestSuite_video.mp4,* a video showing all the unit tests running on VSCode.

3. Exploratory Testing

Two 45 minutes sessions were performed following the Charter Driven Session Based Exploratory testing. The 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 exploratory testing was done using Insomnia IDE.

a. Session 1

The first session of exploratory testing revealed that most of the API functions worked as intended, including GET, POST, PUT, and DELETE. For example, sending a GET request with filters endpoint successfully returned filtered todos. POST a new todo successfully created todo without an ID using the field values in the body of the message. PUT successfully modified the amended fields and DELETE successfully deleted the resource with 201 status code. However, some concerns and new testing ideas have arisen from the testing results.

To begin with, when initializing the Java server, the application successfully launched. However, the system failed to log all the classes, which could lead to system instability.

Then, the results showed that sending a POST request for a todo with an ID in endpoint and all the todo fields resulted in a 400 error with the message "not allowed to create with id." This means that the id is automatically generated, but it is not specified in the documentation. This is one of the areas where the API documentation could be improved. The results also showed that the POST method can be used to update a todo, which is not ideal as POST corresponds to "CREATE" in HTTP methods and updating an existing todo should only be done using the PUT method.

The testing results also showed that sending a GET request for nonexisting todo categories resulted in a list of all the categories. This could lead to bad information or even information leaks. From this instability, new testing ideas were formulated which consist of testing APIs that performed with nonexisting todos.

Additionally, the testing results revealed a concern with the status code 200 for a successful delete. A status code 204 would be recommended in this scenario as the action has been performed but the response does not include an entity.

Additionally, the testing results revealed an inconsistent contract between the client and server. The API documentation states that a *doneStatus* field should be a boolean, but the server returns it as a string. Similarly with the *id*, which can be an int or string in the request json body depending on the API request. This can lead to unexpected behavior in the client and make it harder to maintain the code, so it is important to have a well-defined and consistent contract in an API.

In conclusion, the testing has revealed some concerns and new testing ideas. These include performing more tests with nonexistent todos related to the *tasksof* endpoint to see if there is also any instability, performing more tests to visualize the effects on relationships when using the PUT method with id in the body, and performing more tests combining multiple filter conditions.

b. Session 2

In summary, the second exploratory testing confirmed the successful behavior of some API requests, but also revealed several concerns regarding the functionality of the API.

First, an undocumented API (e.g., OPTIONS requests) results in a code 200, but no documentation can be found on this.

Similarly, to the behavior observed in Session 1 when trying to get the categories of nonexisting todo, the GET method for *tasksof* also gives the wrong result when given a nonexistent todo id.

The critical discovery from this session is that the PUT method has a bug since it deletes the relationships with categories and tasks in projects, preventing the correct usage of the system.
After sending a PUT request for a todo, it deletes both categories and tasksof relationships.

Finally, the shutdown command responds with an unknown response code and no data or header, which is not recommended.

4. Unit Test Suite

The tests are written using the JUnit framework in Visual Studio Code (VSCode).

The tests utilize various libraries such as *com.fasterxml.jackson.databind.JsonNode* and *com.fasterxml.jackson.databind.ObjectMapper* to handle JSON data. The tests use the *java.net.http.HttpClient, java.net.http.HttpRequest,* and *java.net.http.HttpResponse* classes to perform HTTP requests and test the responses. The tests also make use of the *java.util.HashMap* classes to manage data used in the tests. The tests use the static methods of *org.junit.jupiter.api.Assertions* to perform assertions and validate the test results. The tests are annotated with the @Test annotation to indicate that they are test cases.

Note that no build automation tools such as Maven or Gradle were used to manage the build process and dependencies of the software project. Instead, the necessary libraries were progressively added to the project after downloading them. It provides greater control over the build process and allow for a simpler setup. This can also help to avoid the overhead of using a build automation tool, which may not be necessary for a project requiring a few dependencies. The added dependencies are: *jackson-databind-2.14.2.jar, jackson-core-2.14.2.jar, jackson-annotations-2.14.2.jar* are part of the Jackson library, used data processing library for Java.

a. Structure of unit test suite

Following the exploratory testing, the test suite is divided into subsections, each corresponding to a different API Endpoint. Within the subsections, tests are performed for the different documented and undocumented requests methods. Four different tests classes are provided:

i. *tests.java,* contains most of the unit tests of all the identified APIs during the exploratory testing. This excludes all tests related to PUT requests
ii. *tests_PUT.java,* includes all unit tests performed which are related to PUT request. As further detailed in the unit test findings section below, the reason for creating a separate file is to clearly demonstrate the abnormal behavior of using PUT to update a todo.
iii. *test_shutdown.java,* includes one unit test demonstrating the failing test of /shutdown GET request.
iv. *invalid_tests.java,* includes additional unit tests considerations, such as attempting to delete an already deleted object

To run the tests in a random order, the @TestMethodOrder annotation is used to specify the order in which tests should be run. By setting the value to *OrderAnnotation.Order.RANDOM*, the tests are run in a random order.

In order to test POST or PUT requests, a HashMap object is used to store key-value pairs, of the amended fields ("title", "doneStatus"…) with their respective values. The ObjectMapper class isaba then used to convert the HashMap object into a JSON string, which is generally stored as the variable "requestBody". This string represents the body of an HTTP request, which can be used to pass data to the POST or PUT APIs.

In some cases, two separate unit tests were created; one showing the expected behavior failing and one showing the actual behavior working. For example, when testing GET categories of nonexisting todos, the expected result should be a status code 404 Not Found. Therefore, a failing test shows the wrong behavior of the test. Because the API does not behave as documented, but still allows the operation to succeed in an undocumented manner, an added test shows the actual result.

b. Findings of unit test suite execution

Generally, through the unit suite execution, most APIs behaved according to the documentation and have the correct response code, request payload type and response payload type (JSON or XML). In addition, the unit tests continued to highlight the areas of instability identified through the exploratory tests and inconsistent behavior with documentation.

More specifically, the tests clearly demonstrate the cases where the API behavior is different from the documentation, by including two modules, one with the expected behavior and one with the actual behavior. This is done for the following APIs :

- */todos* POST of already existing todo: expected to return 409 conflict, instead returns successful request (200) and updates the todo.
- */todos/:id/tasksof* GET of nonexisting todo: expected to return 404 Not Found, instead returns successful request (200) and a list of all projects
- */todos/:id/categories* GET of nonexisting todo: expected to return 404 Not Found, instead returns successful request (200) and a list of all categories

Moreover, unit tests demonstrate that PUT request provoke a bug, which can lead to preventing the user to use the system correctly. In fact, as detailed in the exploratory testing findings, after using a PUT request, the todo tasksof and categories get deleted. Therefore, all the tests related to PUT method are placed in *tests_PUT.java* class, to demonstrates this abnormal behavior.

Additionally, unit tests using malformed payloads in both JSON and XML were performed and confirmed the normal behavior of the API, which is to return 400 status code (Bad Request).

Finally, a different class called *invalid_tests.java* includes some tests of invalid operation:

- Trying to POST a todo with empty request body returns status code 400 (Bad Request).
- Attempting to DELETE an object which has already been deleted returns status code 404 (Not Found).
- Sending a GET request for todo *tasksof* without any id behaves unexpectedly. Instead of returning 404 (Not Found), it returns successful request with status code 200, with a list of all the projects.
- Sending a GET request for todo categories without any id behaves unexpectedly. Instead of returning 404 (Not Found), it returns successful request with status code 200, with a list of all the categories.

5. Source code repository

The source code repository can be found using the following link: https://github.com/raniaouassif/ECSE429-AutoProject