Project A Report - Group 1

Software Validation (ECSE 429)

Abhijeet Praveen (260985492) – abhijeet.praveen@mail.mcgill.ca
Abhigyan Praveen (261047297) – abhigyan.praveen@mail.mcgill.ca
Sebastien Cantin (260979759) – sebastien.cantin@mail.mcgill.ca
Rooshnie Velautham (260985875) – rooshnie.velautham@mail.mcgill.ca

Department of Electrical, Computer and Software Engineering

Contents

1 Summary of Deliverables

Firstly, we were tasked to perform exploratory testing for the "rest api todo list manager". In the course of three testing sessions, we learned several of its capabilities, potential instabilities, and found ideas for testing in the future. The sessions were performed through the use of Postman programs to examine the capabilities with typical data. All the sessions were conducted by different testers, and it focused on exploring the main functions and capabilities of the application. Each testing session was based on testing requests for one out of three types of objects (todos, categories and projects), as well as the interoperability between that object and the other two. These tests allowed the testers to verify that the application's core functionalities were working as expected. One recurring issue identified across all sessions was related to the behavior of the application when POST requests failed. This could potentially lead to data inconsistencies and challenges in managing records. Another common concern was the absence of a response body in certain scenarios, specifically when deleting instances or relationships. This lack of feedback makes it challenging to confirm the successful execution of delete operations. Testers had to rely on subsequent *get* requests to verify whether the deletions were properly performed.

Several testing ideas were proposed in the sessions to further explore the application's behaviour. These included testing how the application handles the null keyword, working with negative IDs, investigating potential race conditions when sending simultaneous requests, and examining the behaviour of one-sided relationships in different scenarios. This meant the team was able to have a clear vision for the next deliverable of the project, which entailed writing the unit tests.

The team decided to use JUnit as their open-source unit test tool to implement the suite of unit tests. The test suite is divided into 4 files. One focusing on tests related to categories, one for todos, another for projects. The final set of tests focus on the interoperability between those three objects.

Furthermore, the team have also configured the tests to run in any random order as we shuffle the order of the tests before the test suite is executed. This meant the team was required to keep every test independent of each other, as dependency between tests would have resulted in tests failing since the order would have been random. As a side effect of keeping tests independent, the team was also able to achieve another of its targets. We were able to reserve the state of the system after running every test. Since any modifications made during the test were also reversed in that test itself, hence keeping the state of the system unmodified.

Thereafter, a video of the tests being executed in random order was taken and the video can be found here.

2 Findings of Exploratory Testing Sessions

In the first exploratory testing session led by Sebastien Cantin, a thorough examination of “categories” and its interoperability revealed several crucial findings. The session commenced with an exploration of the application’s primary functions and capabilities related to categories, which included tasks like retrieving, modifying and deleting. However, a significant concern came to light regarding ID incrementation. It was observed that failed POST requests led to an increase in the ID counter, which could potentially introduce data inconsistencies over time. Additionally, the exploratory testing unveiled interoperability challenges, particularly in creating relationships between categories and todos. These issues were highlighted due to the creation of one-sided associations. The API was also observed to send unclear error messages.

The second exploratory testing session, conducted by Abhijeet Praveen on the macOS platform focused on the ”todos”. The session asserted many of the findings from the first session while also introducing a few additional insights. The issue of ID incrementation on failed POST requests reemerged as a central concern. It was also noted about the challenges in verifying the success of relationship creations. Creating relationships between entities, such as projects and tasks, returned a 201 status code without a response body, making it challenging to confirm the success of these operations. One important highlight was that there was a security concern. The application was found to be vulnerable to a simple GET request that could shut down the system, highlighting a potential security risk due to the lack of authentication or validation while shutting down the system.

In the third and final exploratory testing session led by Rooshnie Velautham and Abhigyan Praveen, the focus was on ”projects”. Their findings once again conformed to the previous sessions, thereby reinforcing the already identified concerns. In addition, it was observed that both POST and PUT requests behaved similarly, highlighting how it could easily become ambiguous for a user to execute their requests properly. Another finding was related to deletion operations. They resulted in a 200 OK status code without a response body, making it difficult to verify the success of the delete action without sending another request afterwards to verify the deletion had actually occurred.

3 Source Code Repository Structure

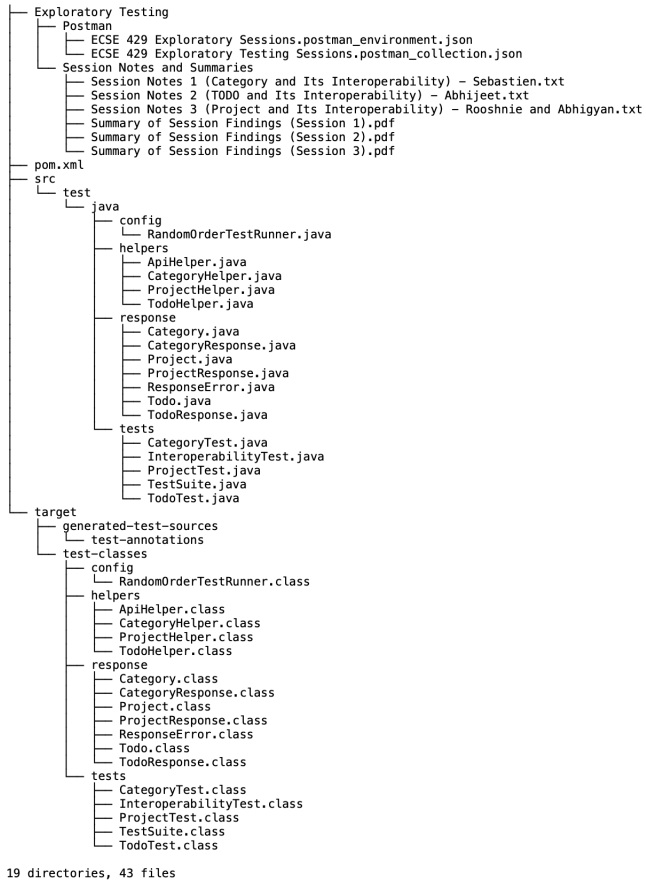

Figure 1: Source Code Repository File Structure

Exploratory Testing: The first top level directory is called **Exploratory Testing**. This directory contains two subdirectories, called **Postman** and **Session Notes and Summaries**.

Postman: Postman has two .json files, containing the postman environment variables and the postman requests collection we used during our testing sessions respectively.

Session Notes and Summaries: This directory contains three .txt files consisted of the testing notes and three .pdf files that summarize their findings.

src/test/java: In the **src/test/java** directory, we have the java code for the JUnit test suite.

config: The **config** directory consists of the code that enables the tests to run in a random order.

helpers: The **helpers** directory consists of the code files that contain helper methods which aid in a smoother execution of the test suite, such as containing a method that sends the request to the API and another method which decodes the response we get from the API.

response: The **response** directory consists of the code that helps us handle the responses we get from the API, such as the objects and the error messages. It also contains classes called Todo, Category and Project which are used as the objects during testing and these classes have the fields that correspond to them.

tests: In the **tests** directory, we have the different files that correspond to the tests for each type of object in the API as well as a file for the interoperability. The **TestSuite,java** file enables us to combine the rest of the four files together and run all the tests at once instead of running the files one at a time.

In addition, the **pom.xml** file is a file typically created for Java Maven projects and the **target** directory contains built artifacts and compiled classes.

4 Structure of Unit Test Suite

The structure of the Unit Test Suite is designed in a way that each type of object created by the API has its own file that contains its own set of distinct tests.

There is a file dedicated to Todos called TodoTest.java. This contains tests that verify that normal CRUD (create, read, update and delete) operations can be performed on Todo objects. Return codes are checked for correctness, the application logic being applied during the tested is also checked using assert calls. Tests also check for unexpected side effects, where this check is performed by comparing the before and after states of the other untouched objects during the test. The file includes tests that check the behaviour of the application when there is malformed input, such as an empty title or description, as well as inputting a non-existent ID during a delete operation.

Furthermore, a file dedicated to Categories called CategoryTest.java and a file dedicated Projects called ProjectTest.java perform the same exact functions in the same manner described for TodoTest.java to confirm that operations on each object from the API are executed properly. However, one notable difference between TodoTest.java and the other two test files, is that the Todo tests have been performed using xml requests while the other two are performed using JSON. This difference is put in place during the setup of sending requests in the helper classes described earlier. This enables us to test that both work properly during testing, and also test what happens if they are malformed.

In interoperabilityTest.java, more CRUD operations are tested on all relationships between the objects in the API. There are a total of 30 tests, 5 for each direction for each of the 6 different relationships that can exist between the 3 objects in the API.

In addition to these tests, there is a test which ensures that the service is on and ready to be tested. This test is performed before any other tests are performed. If this test fails, and we find that the service is not on, the other tests will fail since this one has failed. After all the tests based on the objects have run, there is also another test which attempts to shutdown the system and checks that the system has been shutdown correctly.

5 Findings of Unit Tests Execution

After the execution of the unit tests, the team was able to confirm the capabilities that it had found during the exploratory testing sessions. CRUD operations on the individual objects performed well. We also found that the code works well in completing command line queries as is shown in the figure below.

```
Last login: Fri Oct 20 01:01:48 on ttys000
abhijeetpraveen@Abhijeets-MacBook-Pro ~ % cd Downloads/Application_Being_Tested
abhijeetpraveen@Abhijeets-MacBook-Pro Application_Being_Tested % java -jar runTodoManagerRestAPI-1.5.5.jar
Valid Model Names -model=
todoManager
Model todoManager : Number of app versions available (e.g. -version=2, -versionName=profile1) are: 4
1 - v0 : prototype
2 - v1 : non compressed relationships with guids
3 - v2 : compressed relationships with guids
4 - v3 : compressed relationships with ids
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Using model todoManager
Will configure app as release version v3 : compressed relationships with ids
Running on 4567
 e.g. http://localhost:4567
<todo>  <doneStatus>false</doneStatus>  <description>todo description</description>  <title>todo title</title></todo>
{"todo":{"doneStatus":false,"description":"todo description","title":"todo title"}}
```

Figure 2: Command Line Queries executed properly

However, we did find some bugs during our testing of the interoperability between the objects. When you try to fetch a relationship object, for example, trying to get all todos of a category, the API returns a 200 (OK) response even if we have a "-1" for the project ID corresponding to the request. Instead of a 404 (NOT FOUND) response, since "-1" is not a valid ID, the API fails the expected behaviour and actually sends an OK status response. These example tests can be seen in the image below, and this is also the case for all the other 5 different types of relationships possible between the 3 objects in the API.

```java
/**
 * This test case checks if attempting to retrieve todos associated with a nonexistent category
 * returns the expected result, which is failing (HTTP status code 404 - Not Found).
 *
 * @throws IOException If there's an I/O exception during the test.
 */
no usages  abhijeetpraveen +1
@Test
public void testGetTodosOfNonexistentCategoryExpectedResultFailing() throws IOException {
    // Attempt to retrieve todos associated with a nonexistent category
    HttpResponse response = CategoryHelper.getAssociation( associationType: "todos",  categoryId: "-1", httpClient);
    int statusCode = response.getStatusLine().getStatusCode();

    // Assert that the status code is not equal to the expected value (HTTP status code 404 - Not Found)
    assertNotEquals(String.format("The API behavior for this test is different than expected. Expected %s but got %s.", HttpStatus.SC_NOT_FOUND, statusCode)
}

/**
 * This test case checks the actual behavior when attempting to retrieve todos associated with a
 * nonexistent category. It expects that the API behavior is different from the expected result.
 *
 * @throws IOException If there's an I/O exception during the test.
 */
no usages  abhijeetpraveen +1
@Test
public void testGetTodosOfNonexistentCategoryActualBehaviour() throws IOException {
    // Attempt to retrieve todos associated with a nonexistent category
    HttpResponse response = CategoryHelper.getAssociation( associationType: "todos",  categoryId: "-1", httpClient);
    int statusCode = response.getStatusLine().getStatusCode();

    // Verify that the status code is equal to the expected value (HTTP status code 200 - OK)
    assertEquals(HttpStatus.SC_OK, statusCode);
}
```

Figure 3: Tests that confirm API behaviour is incorrect for getting relationship of nonexistent object