**ECSE 429 Software Validation**

**Project Part A**

**Andrew Chirita (261051967)**

**Andrei Sandor (260977451)**

**1.0 Summarizes deliverables.**

**CharterSessionAndreiSandor.docx**

Charter of the exploratory session for the projects

**CharterSessionAndrewChirita.docx**

Charter of the exploratory session for the projects

**POSTMAN\_TODO**

Folder containing screenshots of the POSTMAN app during the exploratory session    for the todos

**POSTMAN\_PROJECTS**

Folder containing screenshots of the POSTMAN app during the exploratory session    for the projects

**Bug Summary Considerations.docx**

Code report with the bugs found during the exploratory sessions.

**testsTodo.java**

Unit tests for todos

**TestProjects.java**

Unit tests for projects

**videoUnitTests.mp4.**

Recording of running the unit tests.

[**runTodoManagerRestAPI-1.5.5.jar**](https://github.com/andrei-sandor/ECSE429/blob/main/PartA/runTodoManagerRestAPI-1.5.5.jar)

API to test

The first task that we performed is exploratory testing on the “rest api todo list manager” app. We both focus our testing for the todos and the projects. During this 45 minute session, we explored many document and undocumented capabilities of the app. We focus our testing on the main CRUD operations (GET, HEAD, POST, PUT, DELETE). For each of the capabilities, we were working with both the ids and the JSON inputs to confirm operations and possible risks. After this session, we summarized our findings, we found new test ideas, we found areas of potential risk and we explained our concerns. During this session, we worked with the POSTMAN app to do the exploratory and to create scripts, with relevant inputs, to generate findings.

Then, after, we completed a bug summary of considerations that are related to PUT interactions for both todos and projects and to some actions for invalid ids for the GET and HEAD calls with invalid ids for the interoperability cases between todos and projects.

Then, we created unit tests with JUnit for the non operability tests for todos and projects. We used the RestAssured framework from the textbook of the classroom to generate tests with GET, HEAD, POST, PUT and DELETE. Before each test, we set up the app by calling it. Then, we ran the tests with a setup variable, call the function and compared the results structure for the tests. Finally, we destroyed the app to go back to the initial state. We ran the tests in a random order to be sure that we are creating good unit tests.

Finally, we recorded a video to show the correctness of our tests and the expected behavior of our bugs to fail.

**2.0 Describes findings of exploratory testing.**

**Charter:**

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, run a script or program to demonstrate the capability

Exercise each capability identified with data typical to the intended use of the application

**Exploratory Testing based on Charter:**

The main goals of the exploratory testing were to identify and validate the correctness of the capabilities related to todos and projects. Each endpoint has been categorized as a functional capability of the api, or an area of potential instability, based on the tested behavior. Endpoints that behave as expected and as the api documentation describes is considered a documented capability. Inversely, endpoints that do not behave according to the api documentation and that allow a new functionality fall into the undocumented capabilities category. For each capability tested, a query with typical data has been executed via Postman and the result has been captured in an image, stored in either the POSTMAN\_TODO folder or the POSTMAN\_PROJECTS folder.

**2.1 Todos**

**2.1.1 /todos**

**GET :** This scenario was always valid. It requires no JSON inputs for this API call. It returns nicely the current todos (i.d. 1 and 2) with all the fields that they have. The status code is 200. This is a documented capability.

**HEAD:** This scenario was always valid in terms of status code. The status code is always 200, but there seems to be no header appearing in Postman. This is a documented capability.

**POST:**

Invalid. Calling POST with no title, but with description and doneStatus set to a variable, the app crashes with a status code of 400. A title seems to be needed. This can be considered an area of potential instability.

Valid. Calling POST with all the possible scenarios of configuration of the JSON by always having a title field, creates a new todos with status code 200. However, the id is incremented by 2 after doing an unsuccessful POST and after a successful POST. This can be considered an area of potential instability.

**2.1.2 /todos/id**

**GET:**

Valid. Calling GET with any possible combination of JSON fields with a valid id 1 which is initially found in the state of the system retrieves the todo 1 after the API call with a status of 200. This is a documented capability.

Invalid. Calling GET with any possible combination of JSON fields with an invalid id of 10 which is not found in the state of the system generates an error "Could not find an instance with todos/10". The status code is 404.

**HEAD:**

Valid. Calling HEAD with any possible combination of JSON fields with a valid id 1 which is initially found in the state of the system generates a status code of 200, but it doesn’t output something in the result section of POSTMAN. This is a documented capability.

Invalid. Calling HEAD with any possible combination of JSON fields with an invalid id of 10 which is not found in the state of the system generates an error "Could not find an instance with todos/10". The status code is 404. This is a documented capability.

**POST:**

Valid. Calling POST with a valid id 1 which is already there since the beginning of the start of the program and all the required fields of the JSON. It amends a todo with the id. The status code is 200. The same output happens regardless which combination of valid JSON inputs are used. This is a documented capability.

Invalid: Calling POST with an invalid id like 10 which doesn’t exist with a JSON call that contains all the required fields, the app crashes with a status code of 400. This is a documented capability.

**PUT:**

Invalid: Calling PUT with an invalid id of 10 which is not found in the current state of the todo with a valid JSON gives an error message with status code of 404 that "Could not find an instance with todos/10".Another issue, which is a bug, is that the id 1 is issued (found in the state of the system) and regardless of the combination of doneStatus and description, but without title, the app crashes with error code 404. It needs a title, but POST doesn’t need one. This constitutes an area of potential instability.

Valid. Calling PUT with a valid id 1 which is already there since the beginning of the start of the program and all the required fields of the JSON (any valid combination with title)  amends a todos with id. However, there seems to be the case that if doneStatus and descriptions are not there, they are set to default. This constitutes an area of potential instability.

**DELETE:**

Valid. Calling DELETE with a valid id like 2 deletes the todos and returns the 200 status code. This is a documented capability.

Invalid. Calling DELETE with an invalid id like 10 generates a 404 status code error since the todo doesn’t exist. This is a documented capability.

**2.2.3 /todos/id/tasks**

**GET:**

Valid.  A GET request with a valid id returns the list of tasks for the corresponding todos. This is a documented capability.

Invalid. A GET request with an invalid id incorrectly returns the tasks of the todos with the lowest valid project id. This constitutes an area of potential instability.

**HEAD:**

Valid. A HEAD request with a valid id returns the header information for the tasks for the corresponding todos. This is a documented capability.

Invalid.A HEAD request with an invalid id incorrectly returns the header information for the tasks of id 1. This constitutes an area of potential instability.

**POST:**

Valid. A POST request with a valid todos id correctly creates a new todo and a task link between the todos and the newly created tasks if the todo id provided in the JSON body input does not correspond to an existing todo. If an existing todo already exists, the POST request correctly links the todo to the project with the corresponding project id. For a non-existing project id, the POST request correctly raises a 404 error, indicating that the project does not exist. This is a documented capability.

**2.2.3 /todos/id/tasks/id**

**DELETE:**

Valid. A DELETE request with valid todos and task id correctly deletes the specified task associated to the corresponding todos. A DELETE request with an invalid project id correctly raises a 404 error, indicating that the project does not exist. A DELETE request with an invalid task id correctly raises a 404 error, indicating that the project does not exist. This is a documented capability.

**2.2 Projects**

**2.2.1 /projects**

**GET:**

Valid. The GET request for /projects successfully returns a list of all created projects. The response status is always 200. This is a documented capability.

**HEAD:**

Valid. The HEAD request for /projects successfully results in a JSON response with status code 200 and only header information. This is a documented capability.

**POST:**

Valid. A POST request without input fields creates a project with default title, completed, active, and description status. Both title and description are strings, and their default value is the empty string “”. Both completed and active are booleans, and their default value is false. A POST request with input fields creates a project with the specified input values if they are valid. Some fields can be left empty, and the project fields for those will be their default value. This is a documented capability.

Invalid. A POST request with invalid input fields correctly raises an error but raises the project id counter, so future valid POST requests have project ids that are not sequential to the projects made before the invalid POST requests. This constitutes an area of potential instability.

**2.2.2 /projects/id**

**GET:**

Valid. A POST request with a valid id returns the correct project’s data. This is a documented capability.

           Invalid. If the id parameter is an invalid project id, the response has a status code of 404, which correctly raises an error that the project does not exist. This is a documented capability.

**HEAD:**

Valid. A HEAD request with a valid id returns the correct project’s header information. This is a documented capability.

                      Invalid. If the id parameter is an invalid project id, the response has a status code of 404, which correctly raises an error that the project does not exist. This is a documented capability.

**POST:**

Valid. A POST request with a valid project id as url parameter and a valid JSON response body correctly amends the inputted fields to the existing project’s fields. If a JSON body field is left empty, the corresponding field’s value is not overwritten.

           Invalid. A POST request with an invalid id correctly raises an error with status 404, indicating that the project does not exist. This is a documented capability.

**PUT:**

           Valid. A PUT request with a valid project id as url parameter and a valid JSON response body incorrectly amends the inputted fields to the existing project’s fields. If a JSON body field is left empty, the corresponding field’s value is overwritten. The API documentation specifies that the PUT operation behaves identically to the POST operation.  This is an area of potential instability.

Invalid.A PUT request with an invalid id correctly raises an error with status 404, indicating that the project does not exist. This is a documented capability.

**DELETE:**

Valid. A DELETE request with a valid project id as url parameter correctly deletes the corresponding project from the project list. This is a documented capability.

Invalid. A DELETE request with an invalid project id correctly raises an error with status code 404, indicating that the project does not exist. This is a documented capability.

**2.2.3 /projects/id/tasks**

**GET:**

Valid.  A GET request with a valid id returns the list of tasks for the corresponding project. This is a documented capability.

Invalid. A GET request with an invalid id incorrectly returns the tasks of the project with the lowest valid project id. This constitutes an area of potential instability.

**HEAD:**

Valid. A HEAD request with a valid id returns the header information for the tasks for the corresponding project. This is a documented capability.

Invalid. A HEAD request with an invalid id incorrectly returns the header information for the tasks for a non-existing project. This constitutes an area of potential instability.

**POST:**

Valid. A POST request with a valid project id correctly creates a new todo and a task link between the project and the newly created todo if the todo id provided in the JSON body input does not correspond to an existing todo. If an existing todo already exists, the POST request correctly links the todo to the project with the corresponding project id. This is a documented capability.

          Invalid. For a non-existing project id, the POST request correctly raises a 404 error, indicating that the project does not exist. This is a documented capability.

**2.2.3 /projects/id/tasks/id**

**DELETE:**

Valid. A DELETE request with valid project and task id correctly deletes the specified task associated to the corresponding project. A DELETE request with an invalid project id correctly raises a 404 error, indicating that the project does not exist. This is a documented capability.

                      Invalid. A DELETE request with an invalid task id correctly raises a 404 error, indicating that the project does not exist. This is a documented capability.

**3.0 Describes structure of unit test suite.**

For the unit tests, we decided to work with the JUnit framework by using the Java programming language.

The file testsTodo.java contains the unit tests for todos and the file testsProjects.java contains the unit tests for projects.

For every test, we will execute before every test the .jar file with the initial state which contains only two default todos and one default project (we assured that the system is ready to be tested). After each test, we will kill the Process containing the .jar file (restore the system to the initial state by killing the process and rexecuting the process). This kill will clear the .jar file at the end of the test. (Complete termination). These lines of code are found at the beginning of the code with the @BeforeEach and @AfterEach annotations. After that, we can find the tests that have the @Test annotations.

Now, for the unit tests themselves, They are annotated with the @Test JUnit annotation. To succeed in running the GET, HEAD, POST, PUT and DELETE operations, we used the RestAssured library by working from the Alan Richardson Book found in the course outline and the official documentation of RestAssured.

For each type of API actions, we set up initial values (id, JSON objects using the fields of todos and projects). Then, we were doing the action by using the .given().body().API call to generate the result. At the end, we were extracting the response to find the status code and the body for asserting the values.

Furthermore, at the end of each file, we have the tests with JSON payload and XML payload malformed.

To run the unit tests in random order, we used the annotation @TestMethodOrder(MethodOrderer.Random.class).

**4. 0 Describes source code repository.**

Inside the main folder, there are two folders. Let’s first analyse the ReportFiles. Inside, we have the CharterSessionAndreiSandor.docx that contains the exploratory session for the todos. We also have the CharterSessionAndrewChirita.docs that contains the exploratory session for the projects. Furthermore, the folder POSTMAN\_TODOS contains the scripts generated during the exploratory session for the todos. Every file inside has the format NUMBER-API QUERY-path in function of the NUMBER in the respective charter session found after the hour of the observation. The same pattern can be found for POSTMAN\_PROJECTS. Finally, in the main folder, we can find the bugSummaryConsiderations that contains the bugs we have found for the todos and the projects. Also, we have the video videoUnitTests.mp4 with the video that contains the unit tests running. Also, we have SummarySectionFindingTodosAndreiSandor.docx and we have SummarySectionFindingTodosAndrewChirita.docx. They contain the findings and concerns that we have obtained for our respective sessions.

Now, for the code, we have to go inside the Part A folder. When we go inside, we find the .jar file of the runTodoManagerRestAPI. Then, to find the unit tests, we go inside /src/test/java. There, we can find the testsTodo.java and the TestsProject.java. The other files are not relevant to the project; they are auto generated by IntelliJ at project initial setup.

**5. 0 Describe findings of unit test suite execution**

The main findings that we found are that all the capabilities that we found during the exploratory section that give positive expected results behave positively again in our tests. These results are related to the main CRUD operations (GET, HEAD, POST, PUT and DELETE). The return codes are also correct.

For the bugs/areas of potential risks, we were able to reproduce with unit tests those cases; they are failing as expected. We also created the corresponding bug with a test with the expected behavior and it was failing. This fact shows that the app does have some testing issues.