ECSE 429: Software Validation
Fall 2020

Project A

October 26, 2020

Group 11

Steven Luu	260866167
Arneet Singh	260865569
Kevin Li	260862327
Daniel Wu	260863330

Deliverables Summary

In our first exploratory testing session, our goal was to identify the documented and undocumented capabilities of the “rest api todo list manager”. As well as to verify each capability identified with data typical to the intended use of the application. To accomplish this task, we tested the rest api using Postman v7.34.0 on a Windows 64-bit system.

In our second exploratory testing session we seek out to verify capabilities of the rest api by creating a Python script. The python version used here was Python version 3.8.5. As well as finding any areas of instabilities.

Following the exploratory testing sessions, a unit test suite was written to confirm the identified capabilities as well as find other points of instability.

All exploratory testing charters can be found under the folder tab and all Python scripts can be found under the folder pyscripts.

Description of findings of exploratory testing

The “rest api todo list manager” is a tool to help track and manage a project. It is a RESTful api, meaning it uses HTTP protocols to send and receive information. There are 3 total models in this application: todos, projects and categories. All models can be received using the HTTP protocol GET. When creating an instance of a model, IDs do not need to be specified, as they are auto generated. All models can be modified using a POST or PUT and can be deleted using DELETE. All models can have valid relationships as long as their ID is valid. They can also be modified and deleted. All modifications or deletions of relationships are successfully updated on both related objects.

Tasks can be created with a mandatory title. It has 4 fields: ID, title, doneStatus description. If not specified, by default, the todo will be created with a false doneStatus and an empty description. Todos can be linked to be a task of a project and a part of a category. A list of projects can be retrieved from a todo using its ID. A list of categories can also be retrieved from a todo using its ID. The response of a GET request on todos may have a “tasksof” field or a “categories” which indicates the IDs of projects or category they are linked to.

Projects can be created without any specified fields in the body. It has 5 fields: ID, title, completed, active and description. If not specified, by default, the project will be created with the fields completed and active set to false, an empty description and an empty title. Projects can have a relationship with categories and tasks. The response of a GET request on projects may have a “tasks” or a “categories” field which indicates the IDs of todos or categories they are linked to.If the project is not linked to any tasks or todos, those fields will not appear.

Categories can be created with a mandatory title. It has 3 fields: ID, title and description. If not specified, by default, the category will be created with an empty description. A todo can be added to a category through the projects and passing in the request body the ID of the todo. A category can be created and linked to a todo in a single POST. Similarly to the two other models, a list of todos or projects may be retrieved from a category using its ID.

When verifying each documented capability of the “rest api todo list manager”, we noticed in our exploratory test. That every capability worked as documented except when using PUT to amend a specific instance of a model object using an id with a body containing the fields to amend, it would amend the specified instance, but also reset all non mandatory fields. Although this might be the intended purpose of the HTTP protocol, the documentation says otherwise.

By default the request body and response body are in JSON. We can change the response body type by setting “Accept” in the header to “application/xml”. We can also set the “Content-Type” header to “application/xml” if we want to send it in XML.

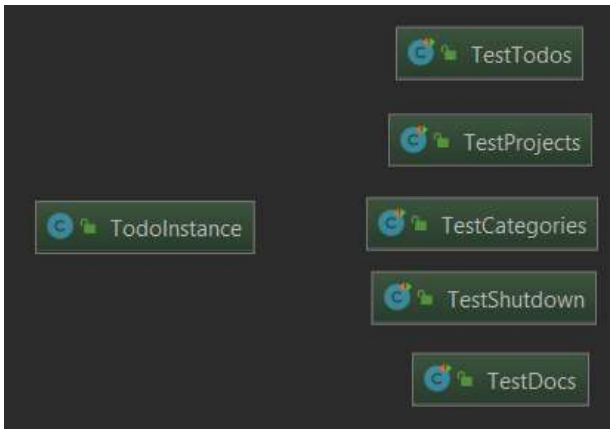
In our second exploratory testing session, we used Python 3.8.5 to create a simple program to test all the capability, but with sample datas. This will allow us to test the application in a similar way to how it would actually be used. To execute HTTP protocols, we had to import Requests into our python scripts, which is a Python HTTP library.

To generate the sample data, we first sent a list of tasks, projects and categories with title fields filled to the “rest api todo list manager”. To create the relationships between the instances of these models, for all newly created projects, we created a random number of relationships with randomly selected newly created tasks and categories.

We then retested every single capability of the application, but this time with the sample data. No difference was found to the original findings. These capabilities include amending specific fields of an instance using both POST and PUT requests, creating relationships between every single instance of all 3 models, removing relationships and deleting instances.

Additionally, while testing for instabilities and capabilities, we discovered two major instabilities. The Todo manager API does not support trailing slashes, which means adding slashes at the end of the url causes an error 404 to occur. The second major instability occurs when we send a faulty creation request. Every time an attempt is made to create an instance of a model, the ID’s is autogenerated and increments by 1 regardless of whether or not the attempt was successful or not. This is an instability as the autoincrement is executed even if the creation of the instance fails.

Description of unit test suite



For unit testing we decided to use JUnit 4 in Java. We have six java classes, five of which is our test suite and the other one is a helper class. Inside our test cases, we test each request methods for todos, categories and projects.

To randomize the order of our unit tests. We imported the package org.junit.runners.* . Which allows us to use the annotation @FixMethodOrder(MethodSorters.JVM). This annotation orders the test execution by JVM enum. In our case, every time the test class runs, the tests will be executed in random order.

In our test suite, the first request method that is tested is the GET method. The test cases assert the contents of the JSON files received with expected contents.

Example:

```
    @Test
     public void testGetByTitle() throws IOException {
         String valid_title = "/todos?title=file%20paperwork";
         JSONObject response = TodoInstance.send("GET",valid_title);
         String result =
response.getJSONArray("todos").getJSONObject(0).getString("title");
         assertEquals(result,"file paperwork");
     }
```

Additionally, our test cases also cover failed GET cases that are expected to fail. We would create faulty requests and assert the request code with error code 404 or an empty json array, in order to ensure that the faulty request was not processed.

Example:

```
    @Test
    public void testGetInvalidID() throws IOException {
        String invalid_request = "/todos/3";
        assertEquals(TodoInstance.getStatusCode(invalid_request),
TodoInstance.SC_NOT_FOUND);
    }
```

Afterwards, we tested the POST methods to test the capabilities. Our test cases would initialize a JSONObject and use the “put” method to identify specific keys and values. We would afterwards send the POST request with our JSONObject and wait 500 milliseconds. Afterwards, we would use the GET method and assert to see if proper changes were made.

Example:

```
    @Test
    public void testUpdateTitle() throws IOException,
InterruptedException {
        String validID = "/todos/1";
        String title = "NEWTITLE";
```

```
        JSONObject json = new JSONObject();
        json.put("title", title);
        TodoInstance.post(validID,json.toString());
        Thread.sleep(500);

        JSONObject response = TodoInstance.send("GET", "/todos/1");

assertEquals(title,response.getJSONArray("todos").getJSONObject(0).get("
title"));
    }
```

As mentioned before, we also have test cases that test what would happen to the system when sending faulty requests. After sending the faulty request, it would assert the request code with error code 404 or an empty json array, in order to ensure that the request was not processed.

Example:

```
    @Test
    public void testCreateInvalidTodo() throws IOException {
        JSONObject json = new JSONObject();
        json.put("description", "DescriptionTest");
        json.put("doneStatus", "TRUE");
        TodoInstance.post("/todos", json.toString());

        JSONObject response = TodoInstance.send("GET", "/todos");
        assertEquals(2,response.getJSONArray("todos").length());
    }
```

We also created test cases for the PUT methods. These test cases have the same format as the POST test cases. The only difference is instead of sending a POST request, we are sending a PUT request.

Example:

```
    @Test
    public void testOverrideTitle() throws IOException,
InterruptedException {
        String validID = "/todos/1";
        String title = "NEWTITLE";

        JSONObject json = new JSONObject();
        json.put("title", title);
        TodoInstance.put(validID,json.toString());
        Thread.sleep(500);
```

```
        JSONObject response = TodoInstance.send("GET", "/todos/1");

assertEquals(title,response.getJSONArray("todos").getJSONObject(0).get("
title"));
        boolean NotExist = false;
        try{

response.getJSONArray("todos").getJSONObject(0).get("categories");

response.getJSONArray("todos").getJSONObject(0).get("tasksof");
        } catch(JSONException e) {
            NotExist = true;
        }

        assertEquals(true,NotExist);
    }
```

The next request method we covered is DELETE. These test cases would send a DELETE request and would wait for 500 milliseconds before sending a GET request and assert to see if the entity was deleted.

Example:

```
    @Test
    public void testDeleteValidTodo() throws IOException {
        String valid_id = "/todos/1";
        assertEquals(TodoInstance.getStatusCode(valid_id),
TodoInstance.SC_SUCCESS);
        TodoInstance.send("DELETE",valid_id);
        assertEquals(TodoInstance.getStatusCode(valid_id),
TodoInstance.SC_NOT_FOUND);
    }
```

Similar to before, we also created faulty test cases for DELETE request methods. We essentially send a faulty DELETE request and assert the status code of the action with the 404 ERROR not found

Example:

```
    @Test
    public void testDeleteInvalidTodo() throws IOException {
        String invalid_id = "/todos/3";

        JSONObject result = TodoInstance.send("DELETE",invalid_id);
        assertEquals(null, result);
```

```
    }
```

We also created test cases for the HEAD request methods. The test cases essentially just get the content type of each response received after sending a request.

Example:

```
    @Test
    public void testHeadTodos() throws IOException {
        String option = "/todos";
        String head = TodoInstance.getHeadContentType(option);
        assertEquals("application/json", head);
    }
```

Finally, we wrote small tests for testing the shutdown feature of the todo manager as well as testing the response code obtained from the API docs, to ensure users can access it at all times.

Example:

```
    @Test
    public void testShutdownServer(){
        String shutdown_url = "/shutdown";
        boolean server_status = true;
        try {
            TodoInstance.getStatusCode(shutdown_url);
        } catch (ConnectException ce){
            server_status = false;
        } catch (IOException e) {
            e.printStackTrace();
        }

        assertEquals(false, server_status);
    }
```

```
    @Test
    public void testGetDocsStatusCode() throws IOException {
        assertEquals(TodoInstance.getStatusCode("/docs"),
TodoInstance.SC_SUCCESS);
    }
```

Description of source code repository

The test suite is split between the models: there is a file containing all the tests for capabilities on todos, another one for projects, another one for categories, another for shutdown and another for docs. The order of the tests are split by the HTTP protocols, with GET at the top, followed by POST, PUT, DELETE and HEAD.

In order to ease the unit testing complexity, a helper class was used. This helper class essentially contains all the functions and public variables that will be used during unit testing. The public variables used throughout the unit test suite are all the common response codes that will be used for assertions, such as status code 404 or status code 200. Additionally, there is a function "runApplication" that creates a process that executes the .jar file. This function will be used in the @Before statement. Also, there is a function "killInstance" that will destroy the process that was created in other functions. This function will be used in the @After statement. The @After and @Before statement will be useful since there will no longer be a need in locally executing the jar file and closing it after each test case and it will allow each test case to start fresh.

The other helper functions includes a "send" function that is used to send GET, DELETE, HEAD requests. For POST and PUT, we created separate functions since these requests require a response body in order to properly send the request. We also have "GetHeadContentType" which returns the content type of a specific request type as well as a "GetStatusCode" which returns the status code of a specific request type.

Description findings of unit test suite execution

All capabilities and instabilities found during the exploratory testing were confirmed. All documented capabilities in the provided doc are tested and confirmed. As was determined in the exploratory testing, when updating or amending, PUT does not have the same behaviour as POST. Also as we found while exploratory testing, a sleep is needed after a POST or PUT, to GET the result.

On requests like GET "/todos/:id/tasksof", where the projects linked to the todo of ID :id, the order or index of a todo is not consistent. For example, while trying to test for the existence of the link to a todo in a project as a "tasksof", the index of the ID of the todo was not always the same.