ECSE 429

Software Validation

Team 9

Automation Project - Part A Deliverables

Christopher Cui

Student ID: 260867703

Bo Wen Li

Student ID: 260787692

Dominic Wener

Student ID: 260871052

Glen Xu

Student ID: 260767363

TABLE OF CONTENTS

Summary of Deliverables

Charter Driven Exploratory Testing Session Deliverables

- Detailed Session files highlighting:
    1. Application capabilities discovered and tested during the session
    2. Concerns and potential areas of risks
    3. Short summaries of findings
    4. Session conclusion (new testing ideas and paths to explore in future sessions)
- All other files derived from the test sessions (i.e. screenshots, spreadsheets, programs, etc.)

*The team performed 4 test sessions, one for each of the following:

- ➢ Session 1 - Projects: Christopher and Dominic
- ➢ Session 2 - Todos: Dominic and Bowen
- ➢ Session 3 - Capabilities: Bowen and Glen
- ➢ Session 4 - Interoperability: Glen and Christopher

Unit test suite used for testing the capabilities highlighted during the exploratory testing

- One Unit Test Suite for each instance Model
    1. At least one Unit test for each documented feature
    2. Unit tests for any undocumented capabilities found during test sessions
    3. Unit tests for invalid JSON payloads

Bug Summary Files

- 80 characters or less (per bug)

Video File showing all Unit tests executing

- Unit tests are done at a random order
- All Unit tests are done consecutively

Describes findings of exploratory testing.

Session 1 Projects

In the docs, there was something confusing for the example output. In the example body, the “true” and “false” fields for the json had quotation marks on them. Using these outputs for the body of my post request led me to have many errors during my exploratory testing for the POST endpoint. After fixing this error, the POST requests worked as expected. XML worked fine. Projects have 5 fields: ID, completed, active, description and title. During my exploratory testing, I also noted many times the “HEAD” requests returned nothing as a response. The default behavior for no input in the “completed” and “active” fields is left as “false” if no value for the fields is specified. However, I now believe that this is the intended behavior. The user must simply look at the headers section of the response. All the tests in the suite passed.

Session 2 Todos

All invalid POST requests, i.e.when the payload was missing required fields or had too many fields, returned the code ‘400 Bad request’ along with an error message describing the issue. If an invalid field is given in the payload, the status code returned was ‘400’ along with a message describing which of the given fields is invalid. Since the title field is the only mandatory field for the JSON payload, an error message is returned if it is missing or if it has an empty value.

Error messages were also given when unsupported commands were issued to the API. For example, doing a DELETE request on the endpoint ‘/todos’ returns the status code ‘405 Methods not allowed’. The API handles invalid commands given by the user.

Furthermore, each POST request increments the automatically generated unique IDs. A failed POST request will result with a “400 Bad request” status code, but will still increment the unique IDs. In other words, the unique IDs will not always be reliably incremented if a failed (invalid syntax error) POST request occured in between consecutive valid POST requests. This could be a risk factor, if the client needs to have related todos with consecutives IDs.

In general, ID creation should not be an incremental based approach. Id of 1, then next is ID of 2 etc. If we generate a project with id 1, then a todo object with id 1, how do we differentiate them? We should use GUIDs for the API instead. GUIDs are 38 digit serial numbers which can be used on any item in the universe. There is no more management headache.

Session 3 Categories

During the test session focusing on categories, we tested the “/categories” and “/categories/:id” endpoints extensively. The application was launched on the command line and tested with Postman.

For the “/categories” endpoint, there are three method types - GET, HEAD, and POST. We confirmed that the GET method type contains the capability of returning all of the current categories in the server, and discovered that there are 2 default categories already. We have confirmed the HEAD method type returns “application/json” in content type. We have also confirmed that POST method type contains the capability of adding new categories on the server. The JSON body for POST cannot however contain an id field. We have discovered that the server auto assigns an id for every new category object that increments from its default 2 categories. For example, the default 2 categories have ids 1 and 2 respectively, the new category will have id 3, 4 etc…

For the “/categories/:id” endpoint, there are five method types - GET, HEAD, POST, PUT, and DELETE. We have confirmed the GET method type contains the capability of returning a specific category given an id. This will also return 404 not found when the server does not contain a category with the given id. We have confirmed the HEAD method type returns “application/json” in content type when given an existing id. The POST method type is confirmed to be able to edit title and description of existing ids with the changed title and description in the JSON body. The POST method type however cannot create a category at an id that the server does not contain a category yet. Therefore, the only way to add a new category is with the endpoint “/category” and the method type POST. The PUT method type is similar to the POST method type, and is confirmed to have the capability of editing title and description of existing ids with the changed title and description in the JSON body. The DELETE method type is confirmed to have the capability of deleting an existing id and will return an error message when the id does not exist on the server. An interesting behavior is observed with the combination of the “/categories/:id” endpoint with DELETE method type and “/categories” endpoint with POST method type. After deleting a category at an id, then adding a new category with POST, the unused id is not used, but is incremented from the unused id. For example, the server currently contains 3 categories with ids 1, 2, and 3 respectively. We delete the category with id 3, and add a new category with a POST request. The new category will have an id of 4 rather than 3. This can cause a lot of inconveniences and can be considered a bug because of the unused id and difficulty of keeping track of which id the new category has.

During this test session, we have also tested the endpoint “/shutdown” with a GET request and received the following error: Error: connect ECONNREFUSED 127.0.0.1:4567. This endpoint should be tested more thoroughly to confirm the functionality/dysfunctionality of the endpoint.

Session 4 Interoperability

The PUT and POST commands have the same descriptions in the document, and they behave similarly. This creates confusion of whether these commands should behave exactly the same or not.

If we GET an endpoint, the boolean values such as "completed" will be returned as a string with quotation marks. When we POST an endpoint, however, the boolean values have to be pushed as a boolean, without quotation marks. Such inconsistencies can cause a lot of confusion and inconvenience during the test sessions.

The POST command has a special feature that happens in all cases where a user adds a capability into another. The feature will be described with a sample case, where a user adds a "tasksof" ("project") into "todo" with "id" == "1", and there already exists 2 "project"s with "id" == "1" and "2" respectively.

1. If a user adds a project with an existing id, say "2", to todo, without any other keys such as "title" or "completed", then the project will be added to the todo successfully.
2. If a user adds a project with an existing id, say "2", to todo, with other keys such as "title" or "completed", then the project will be added to the todo successfully, but the updated information will all be ignored, and only the original information will be kept. Note that this behaves differently from the POST command solely in "projects" capability, where the information can be updated.
3. If a user adds a new project to todo, and specifies its id, say "3", then the user will receive an error message of "Could not find thing matching value for id".
4. If a user adds a new project to todo, without specifying its id, then the project will be added successfully with the next available "id", and it will also be added to the todo. Note that the "id"s of the deleted projects are still reserved, and the new "id" will skip over these "id"s.

Source Code Repository

We used GitHub to store our source code repository, and Google Drive to store documents and miscellaneous files. The source code repository consists of Eclipse settings, gradle builds, unit tests, and support classes that help send the HTTP requests. On Google Drive, we have stored the report, test sessions, and demonstration videos. By using version control systems and collaborative storage, it was easier to work on the project simultaneously at our own pace.

Describe findings of unit test suite execution

Projects

For the project's test class, the head requests were easily asserted for correctness by comparing getHeadContentType with “application/json”. There were also PUT requests that tested overwriting fields of a previously created project object, such as the description and title fields. The GET requests were rather straightforward as well.

Todos

GET requests that use invalid endpoints always return a null JSON response along with the Status code ‘400 Bad Request’ and an error message. The API also handles invalid commands, such as DELETE requests on the ‘/todos’ endpoint. It only accepts DELETE requests on the ‘/todos/id’ endpoints and executes them correctly.

Every POST request, whether valid or invalid, increments the unique ID given by the API to each new TODO instance. Deleting a TODO instance will not decrement these unique IDs.

POST requests done on the endpoint ‘/todos/id’ are implemented using a PUT request. The fields of the given JSON payload will change the field values of existing TODO instances. Otherwise, the API throws the appropriate error message and status code ‘400’.

Categories

The capabilities and defects/inconveniences discovered during the test sessions for categories were confirmed by the category unit tests. These categories unit tests focused on the “/categories” and “/categories/:id” endpoints.

The “/categories” endpoint has 3 method types - GET, HEAD, and POST. The GET functionality is confirmed by checking if there are a total of 2 categories in the returned JSON object (default is 2). The HEAD functionality is confirmed by checking if the returned content type is “application/json”. The POST functionality is confirmed by requesting a JSON body of a new category with its title and description fields, and checking with a GET request that the total number of categories is now 3. We discovered that a thread delay of 500 ms is necessary to ensure the server has processed the POST request, and the GET request will return the correct values. Next, to ensure the server category values are the same, we delete the category by requesting a DELETE method to the endpoint “/categories/3” and “/categories/4”. This is because in this test suite, there are two instances where we add categories and we have confirmed during test sessions that for categories, one cannot POST a new category on a null id with the endpoint “/categories/:id”, and that the server keeps its own index that only increments when one POSTs a category with the “/categories” endpoint (does not change after the category is deleted). Thus, we cannot accurately identify the new category as id 3 or 4, so

we attempt to remove both of them to restore the system to its original state. Finally, we test another POST request with an id field in the JSON body (should not work), which we test by checking that the GET request of the total number of categories is still 2.

The “/categories/id” endpoint has 5 method types - GET, HEAD, POST, PUT, and DELETE. The GET functionality is confirmed by checking if the endpoint “categories/1” returns an id of 1 in its JSON body. We also test an invalid id with the endpoint “categories/5” and confirm the return code is 404 not found. The HEAD functionality is confirmed by checking if the returned content type is “application/json”. The POST functionality is tested in 3 parts - by changing the title, description, and using an endpoint with a null id (does not exist yet in categories). The title and descriptions changes are verified by their respective fields with a GET request. They are then changed back to normal after each unit test. We have confirmed in test sessions that one cannot create a POST request to a null id. This is confirmed in the test where we try to run a POST request to “/categories/5” and ensure that the total number of categories is still 2. The PUT functionality is tested similarly to the POST functionality - by changing the title and description independently. Finally, the DELETE functionality is tested by creating a new category (this is the other time we create a new category with POST and endpoint “/categories”), confirming the category is indeed created by checking the size of categories, and deleting the category by sending DELETE requests to both “/categories/3” and “/categories/4” endpoints to ensure it is deleted. We now test that it is deleted by checking the size of categories again. We also test DELETE with an invalid id, “/categories/5” and ensure that the return is null.

Interoperability

1. The inconsistency of capability names.

The same capabilities are often referred to as different names in different capabilities. For example, the "todo" is called "todo" in "category", but it is called "task" in "project". Another example is that, the "project" is called "project" in "category", but it is called "tasksof" in "todo". Such inconsistencies can cause a lot of confusion and inconvenience to the users, it is counter-intuitive that the "tasksof" with "id" == "1" and the "project" with "id" == "1" refer to the same thing but have different names.

2. The inconsistency of key names.

The similar capabilities are often referred to as different names in different capabilities. For example, "completed" in "project" and "doneStatus" in "todo" have essentially similar meanings, but they are referred to as different names. Such inconsistencies can cause a lot of

confusion and inconvenience to the users. With different names, it is natural for the users to assume that the two keys have different meanings.

3. "Todo" with "id" == "1" and "category" with "id" == "1"

After the website was launched initially, before any tests or modifications, the "todo" with "id" == "1" contained a "category" with "id" == "1", but the "category" with "id" == "1" did not contain a "todo" with "id" == "1". This made the following testing process confusing. Intuitively, users would assume that when you post or delete a "category" from a "todo" or vice versa, the relationship should be established or terminated in both "todo" and "category" endpoints concurrently. But the fact that the default status of the product did not obey this, it is hard to determine the expected values during designing the unit tests.

Describes structure of unit test suite

For the structure of our suite, we had a separate java test file for each of the capability types we were testing. We had one for projects, todos, interoperability, and categories.As for the framework, we used JUnit in Java. JUnit already randomizes the unit test order of execution out of the box. We also had an APIInstance class which would act as our shell.

In this class, the REST API is initialized through a command line script. It also has Put, Post, Delete functions which our test cases call. These functions are the skeleton for our API calls. They add the headers, parse the json, and filter the appropriate items to return to the caller. For our test cases, we have a @Before and @After that is run around every test. In the before annotation, we initialize an instance of our rest api. In the after, we kill the instance. This is to ensure we reset the state before every test.

Get Request:

```
@Test
public void Get_Project_Title() throws IOException {
    String project_url = "/projects";
    JSONObject response = TodoInstance.send( type: "GET", project_url);
    String result = response.getJSONArray( key: "projects").getJSONObject( index: 0).getString( key: "title");
    assertEquals( expected: "Office Work", result);
}

@Test
```

Here is an example of a get request. We create a string of the request, call the get method from our ToDoInstance which communicates directly with the API process and returns a JSON object, and we extract the field “title” in this specific test.

```
}

//POST Projects
@Test
public void Create_Valid_Projects() throws IOException, InterruptedException {
    String validID = "/projects";
    String title = "TitleTest";
    String description = "DescriptionTest";

    JSONObject json = new JSONObject();
    json.put("title", title);
    json.put("description", description);
    APIInstance.post(validID,json.toString());
    Thread.sleep( millis: 500);

    JSONObject response = APIInstance.request( type: "GET",  option: "/projects");
    System.out.println(response);
    assertEquals( expected: 2,response.getJSONArray( key: "projects").length());
}

@Test
```

Here is an example of our POST request. In this case, we create a template of the body with iD , title and description. We then make a post request through our APIInstance.

```
//POST projects/id

@Test
public void Update_Description() throws IOException, InterruptedException {
    String validID = "/projects/1";
    String description = "testdescription";

    JSONObject json = new JSONObject();
    json.put("description", description);
    APIInstance.post(validID,json.toString());
    Thread.sleep( millis: 500);

    JSONObject response = APIInstance.request( type: "GET",  option: "/projects/1");
    assertEquals( expected: "DESCRIPTION",response.getJSONArray( key: "projects").getJSONObject( index: 0).get("description"));
}

@Test
```

Here is an example of our Put request. We update the project with is “1”, and update its description through our APIInstance Put method.