ECSE 429

Software Validation

Team 9

Automation Project - Part B 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

Description of the Story Test Suite

- Gherkin feature files created for each Story test. A feature file contains three Scenarios: Normal flow; Alternate Flow; Error flow. Each scenario will be executed 1 to 3 times as individual tests using different data (examples).
- JAVA Definition files that contain the logic used when Cucumber interpretes the Gherkin files.

Description of the Source Code Repository

Breakdown of the Repository structure used while writing the Features and their definition files.

Findings of Story Test Suite Execution

Short summary of findings and conclusion found after completing the Story Tests, executing them and debugging them.

Video File showing all Unit tests executing

Unit tests are executed in a random order
All Story Tests are done consecutively

Describes structure of story test suite

TODOS

Scenario #1: As a user, I want to create a Todo so that I can track my tasks

Normal Flow: Create a new Todo with a title, a 'doneStatus' and a description

Given the user wants to Add a Todo with a `<title>`, a `<doneStatus>` and a `<description>`, a JSON object is created with the above fields. **When** the user creates the TODO instance, the JSON object is sent via POST request to the API. **Then** the TODO instance with the given `<title>`, `<doneStatus>` and `<description>` is successfully added to the TODO list.

Alternate Flow: Create a new Todo with only a title

Given the user wants to Add a Todo with only a `<title>`, a JSON object is created with a 'title' field. **When** the user creates the TODO instance, the JSON object is sent via POST request to the API. **Then** the TODO instance with the given `<title>` is successfully added to the TODO list.

Error Flow: Create a new Todo with an invalid/missing title

Given the user wants to Add a Todo with a `<doneStatus>` and a `<description>`, but without a `<title>`, a JSON object is created with the `<doneStatus>` and `<description>` fields. **When** the user creates a TODO instance, the JSON object is sent via POST request to the API. **Then** the desired TODO

instance will not be created, because the API will reject the 'Bad Request' that uses an invalid JSON object (i.e. the 'title' field is required when sending a POST request on the endpoint "/todos").

Scenario #2: As a user, I want to retrieve a Todo so that I can view a task

Normal Flow: Retrieve a Todo using its ID

Given the user wants to retrieve a TODO by using its <id>, a Dummy TODO instance will be created and its <id> extracted then stored, such that it can be retrieved using a GET request on the endpoint "/todos/<id>". A JSON object with only a <title> field is used for the POST request. **Given** the user needs to know the <id> of the Dummy TODO instance, the <id> is extracted by filtering the TODO database using the known title (i.e. endpoint "/todos?title=<title>"). **When** the user retrieves the TODO instance by using its <id>, a GET request on the endpoint "/todos/<id>" is sent. **Then** the TODO instance with the given <id> is successfully returned.

Alternate Flow: Retrieve a Todo using its Title

Given the user wants to retrieve a TODO by using its <title>, a Dummy TODO instance will be created, such that it can be retrieved using the URL Query: "/todos?title=<title>". A JSON object with only a <title> field is used for the POST request. **When** the user retrieves the TODO instance by using the <title>, a GET request on the endpoint "/todos?title=<title>" is sent. **Then** the TODO instance with the given <title> is successfully returned.

Error Flow: Retrieve a Todo using an invalid/missing ID

Given the user wants to retrieve a TODO by using an <id> that does not exist, such that it cannot be retrieved using a GET request on the endpoint "/todos/<id>". **When** the user retrieves the TODO instance by using the <id>, a GET request on the endpoint "/todos/<id>" is sent. **Then** no TODO instances are returned with the desired <id>, because the API will reject the 'Bad Request' that uses an invalid endpoint (i.e. the GET request endpoint must exist in the database).

Scenario #3: As a user, I want to filter a Todo list so that I can track specific tasks

Normal Flow: Filter a Todo list using the completion status

Given the user wants to filter all TODO instances using their <doneStatus>, Dummy TODO instances are created with different <doneStatus> values, such that they can be filtered using a GET request on the endpoint "/todos?doneStatus=<doneStatus>". JSON objects with the <title> and <doneStatus> fields are used for the POST requests. **When** the user filters the TODO, a GET request on the endpoint "/todos?doneStatus=<doneStatus>" is sent. **Then** completed TODO instances are returned if <doneStatus> = true or unfinished TODO instances are returned if <doneStatus> = false.

Alternate Flow: Filter a Todo list using a specific description

Given the user wants to retrieve a TODO with a specific <description>, a Dummy TODO instance is made, such that it can be filtered using a GET request on "/todos?description=<description>". A JSON object with the <title> and <description> fields is used for the POST request. **When** the user filters the TODO instances, a GET request on the endpoint "/todos?description=<description>" is sent. **Then** the TODO instance with the searched <description> is successfully returned.

Error Flow: Filter a Todo list using invalid URL Query Parameters
Given the user wants to retrieve a TODO by using its <title>, a Dummy TODO instance will be created, such that it can be retrieved using the valid URL Query "/todos?title=<title>". A JSON object with only a <title> field is used for the POST request. **When** the user wants to retrieve the TODO instance by using the <title>, a GET request using the invalid endpoint "/todos? ti tle=<title>" is sent. **Then** no TODO instances are returned, because the API will reject the 'Bad Request' that uses invalid URL Query Parameter syntax.

Scenario #4: As a user, I want to modify a Todo so that I can update my tasks
Normal Flow: Overwrite the title and description of an existing Todo
Given the user wants to modify the <title> and <description> of an existing TODO, a Dummy TODO will be created and its <id> extracted, such that it can be retrieved (GET) and modified (POST) by using the endpoint "/todos/<id>". A JSON object with only the <title> and <description> fields is used. **Given** the user needs to know the <id> of the TODO instance, the <id> is extracted by filtering the TODOs using the known <title>. **When** the user modifies the TODO instance, a JSON object with the fields <newTitle> and <newDescription> is sent via POST request to the API using the endpoint "/todos/<id>". **Then** the TODO instance fields are changed to <newTitle> and <newDescription>.

Alternate Flow: Change the completion status of an existing Todo
Given the user wants to modify the <doneStatus> of an existing <title> TODO, a Dummy TODO will be created and its <id> extracted, such that it can be retrieved (GET) and modified (POST) by using the endpoint "/todos/<id>". A JSON object with only the <title> and <doneStatus> fields is used. **Given** the user needs to know the <id> of the TODO instance, the <id> is extracted by filtering the TODOs using the known <title>. **When** the user modifies the TODO instance, a JSON object with the field <newDoneStatus> is sent via POST request to the API using the endpoint "/todos/<id>". **Then** the completion status field of the TODO instance is changed to <newDoneStatus>.

Error Flow: Modify the title, doneStatus or description of a Todo that does not exist
Given the user wants to modify the <field> of a TODO by using an <id> that does not exist, such that no TODO instance is returned when a GET request using the endpoint "/todos/<id>" is sent. **When** the user modifies the TODO instance, a JSON object with the new <field> value is sent via POST request to the API using the endpoint "/todos/<id>". **Then** the desired <field> will not be changed, because the API will reject the 'Bad Request' that uses an invalid POST endpoint.

Scenario #5: As a user, I want to remove a Todo so that I do not track unnecessary tasks
Normal Flow: Remove an existing Todo with a valid ID

Given the user wants to remove a <title> TODO that exists, a Dummy TODO instance is made such that it can be filtered using a GET request on "/todos?title=<title>". A JSON object with a <title> field is used for the POST request. **Given** the user needs to know the <id> of the Dummy TODO instance, the <id> is extracted by filtering the TODOs using the <title> (i.e. endpoint "/todos?title=<title>"). **When** the user deletes the desired TODO instance, a DELETE request on the

endpoint "/todos/<id>" is sent. **Then** the TODO instance with <title> and <id> is successfully removed from the database.

Alternate Flow: Remove all existing Todos that have been completed

Given the user wants to remove all completed TODOs, Dummy TODO instances are made such that they can be filtered using a GET request on "/todos?doneStatus=true". JSON objects with the <title> and <doneStatus> fields are used for the POST requests. **When** the user deletes the completed TODO instances, they are extracted by using a GET request on "/todos?doneStatus=true". For each returned TODO, its <id> is extracted and then used to send a DELETE request on the endpoint "/todos/<id>". **Then** all completed TODO instances (<doneStatus> = true) are successfully removed from the database.

Error Flow: Remove a Todo that does not exist

Given the user wants to remove a <title> TODO that does not exist, such that no TODO is returned when the database is filtered using a GET request on "/todos?title=<title>". **Given** the user will use <id> to remove the TODO, such that a GET request on the endpoint "/todos/<id>" returns Null. **When** the user deletes the desired <title> TODO instance using <id>, a DELETE request on the endpoint "/todos/<id>" is sent. **Then** the TODO instance with <title> is not removed, because the API will reject the 'Bad Request' that uses an invalid DELETE endpoint.

PROJECTS

Scenario#6: As a student, I create a project, so that I can organize my work

Normal flow: Create Project that does not exist

In this scenario we create a project using properties such as title description, that are not already present in the database. We assume that this creation works. The user sends a post request to the api, and a response is read.

Alternate Flow: Create existing project

In this scenario we create a project using properties that already exist in the database. In this case, we also assume that it works without issue. The user sends a post request to the api, and a response is read.

Error Flow: Create project using predetermined id

In this scenario, we try to create a project using a predetermined id. This fails, and we assert that no projects were created. Creating a project with id is not allowed because the ids are auto generated by the system.

Scenario#7: As a student, I want to adjust project descriptions so that I can adjust my schedule as needed.

Normal Flow: Modify description

In this scenario, we change the description to a new description. At the start, a project without description is created. Then, we do a modify description command, and test the output of the api to make sure it has been modified.

Alternate Flow: Modify description that it currently has
We create a string with a start description. We then try to change the string to the exact same description. The string remains the same and no error is thrown

Error Flow: Change description for non existent project
We try to change the description of a non-existent project. This throws an error, and a message is displayed from the api.

Scenario#8: As a student, I remove a project that I no longer need to do, to declutter my schedule.

Normal Flow: Remove a project
We create a project normally with no connections, then we remove it. We then verify from the API GET request that the project no longer exists.

Alternate Flow: Remove a project related to a category
We create a project, create a relation between it and a category. We then try to remove the project, and assert that it has been removed from our database.

Error Flow: Remove a non existent project
We try to call the remove projects command, however in this case we did not actually create a project to be removed. This throws errors, with id -1 and id 0

Scenario#9: As a student, I mark a project as done on my course to do list, so that I can track my schoolwork.

Normal flow: Mark non active project as active
First, we create a project with the state already set as non active. Then, we do a post request with the new property active field set to active. We then check that the property was changed after the get request.

Alternate Flow: Mark active task as active
First, we create an active task. We then do a post to try to set the status to active. We check if this works.

Error Flow: Mark non existing task as active
We do not create a project. We then try to mark non existing project id as active, and assert that we get an error.

Scenario#10: As a student, I mark a project as completed on my course to do list, so that I can track my schoolwork.

Normal Flow: Mark a non completed project as done
First, we create a project with the completed set as false. Then, we create a post request that tries to set the completed to true.

Alternate Flow: Mark a completed project as completed
First, we create a project and have it set already as completed. We then try to make a POST request to set it to completed. We witness that this has no change on the project state.

Error Flow: Mark a non existing project as completed
We do not create a project. We then call a POST request to modify a non existing project, and assert that an error is thrown.

CATEGORIES
Scenario#11: As a user I want to add a new category
Normal Flow: Adding a new category title to categories list
In this scenario, we first check that there are the default two categories. When we add a category with the title specified in the examples, then there will be a new category with that title.

Alternate Flow: Adding a new category title and description to categories list
In this scenario, we first check that there are the default two categories. When we add a category with the title and description specified in the examples, then there will be a new category with that title and description.

Error Flow: Adding a new category title to categories list and specifying id
In this scenario, we first check that there are the default two categories. When we add a category with the title and id specified in the examples, then there will still be two categories because one cannot specify id in the JSON body for categories.

Scenario#12: As a user I want to use the POST method to edit a category
Normal Flow: Editing a category's title
In this scenario, we add a new category with a specified title specified in the examples with POST. When we change the title of the category to a new title specified in the examples with POST, then there exists a category with the new specified title.

Alternate Flow: Editing a category's description
In this scenario, we add a new category with a specified title and description specified in the examples with POST. When we change the title and description of the category to a new title and description specified in the examples with POST, then there exists a category with the new specified title and description.

Error Flow: Editing a category's title with invalid id

In this scenario, we first check that there are the default two categories. When we try to change the title of the category to an invalid id specified in examples with POST, then we make sure that the category at the invalid id does not have the new specified title.

Scenario#13: As a user I want to use the PUT method to edit a category
Normal Flow: Editing a category's title
In this scenario, we add a new category with a specified title specified in the examples with POST. When we change the title of the category to a new title specified in the examples with PUT, then there exists a category with the new specified title.

Alternate Flow: Editing a category's description
In this scenario, we add a new category with a specified title and description specified in the examples with POST. When we change the title and description of the category to a new title and description specified in the examples with PUT, then there exists a category with the new specified title and description.

Error Flow: Editing a category's title with invalid id
In this scenario, we first check that there are the default two categories. When we try to change the title of the category to an invalid id specified in examples with PUT, then we make sure that the category at the invalid id does not have the new specified title.

Scenario#14: As a user I want to get a category
Normal Flow: Get a category with category id
In this scenario, we first check that there are the default two categories. When we try to get a category with an id specified by examples, we make sure that the title of the category also matches the title specified by examples (these are the default categories). Then we check again that there are the default two categories.

Alternate Flow: Get a category with category name
In this scenario, we first check that there are the default two categories. When we try to get a category with a title specified by examples, we make sure that the categories exist (these are the default categories). Then we check again that there are the default two categories.

Error Flow: Get a category with invalid category id
In this scenario, we first check that there are the default two categories. When we try to get a category with an invalid id specified by examples, we make sure that the title of the category does not match the title specified by examples (these are the default categories). Then we check again that there are the default two categories.

Senario#15: As a user I want to remove a category
Normal Flow: Removing a category from the categories list with title

In this scenario, we add a new category with a specified title specified in the examples with POST. When we remove the category with the specified title in examples, then we make sure that there are now two categories (default number of categories is two).

Alternate Flow: Removing a category from the categories list with title and description
In this scenario, we add a new category with a specified title and description specified in the examples with POST. When we remove the category with the specified title and description in examples, then we make sure that there are now two categories (default number of categories is two).

Error Flow: Removing a category from categories list with invalid id
In this scenario, we add a new category with a specified title specified in the examples with POST. When we attempt to remove the category with an invalid id number, then we make sure that there are still three categories (default number of categories is two).

INTEROPERABILITY

Scenario 16: As a user, I want to add an existing category to an existing project.

Normal Flow: Adding an existing category to an existing project
In this scenario, we make sure a project and two categories already exist. When we try to add the category with a valid id to a project, there will be a new category with the corresponding id to the project.

Alternate Flow: Adding an existing category that has already been assigned to this project again
In this scenario, we make sure there exists at least 1 category and 1 project, this project already has this category. Then we attempt to add the category with valid id to the project again, then the category should still be assigned to this project.

Error Flow: Adding a non existing category to an existing project
In this scenario, we make sure there exists at least 1 project. When we try to add a category with an invalid id to a project, the system should be aware that this category does not exist.

Scenario 17: As a user, I want to see which category this todo belongs to.

Normal Flow: Getting the existing category of an existing todo
In this scenario, we make sure at least a todo exists, and it has at least one category. When we attempt to get the category id of the todo, then the todo should still have that category.

Alternate Flow: Getting multiple categories of a todo
In this scenario, we make sure at least a todo exists, and it has at least two categories. When we attempt to get the ids of the categories of the todo, then the todo should still have those categories.

Error Flow: Getting non existing category of a todo
In this scenario, we make sure at least a todo exists, and it has at least two categories. When we attempt to get the category with an invalid id of the todo, then the todo should still have those categories.

Scenario 18: As a user, I want to modify the information of a category of an existing todo.

Normal Flow: Modifying an existing category to an existing todo
In this scenario, we make sure an existing category is already assigned to an existing todo. When we try to modify the description of the category of this todo, then the description should change correspondingly.

Alternate Flow: Modifying an existing category that already has a description to an existing todo
In this scenario, we make sure an existing category is already assigned to an existing todo, and the category already has a description. When we try to modify the description of the category of this todo, then the description should change correspondingly.

Error Flow: Modifying a non existing category to an existing todo
In this scenario, we make sure an existing category is already assigned to an existing todo. When we try to modify the description of a category with an invalid id of this todo, the system should be aware that this category does not exist.

Scenario 19: As a user, I want to add an existing category to an existing todo.

Normal Flow: Adding an existing category to an existing todo
In this scenario, we make sure two todos and two categories already exist. When we try to add the category with a valid id to a todo, there will be a new category with the corresponding id to the yodo.

Alternate Flow: Adding an existing category that has already been assigned to this todo again
In this scenario, we make sure there exists at least 1 category and 1 project, this todo already has this category. Then we attempt to add the category with valid id to the todo again, then the category should still be assigned to this todo.

Error Flow: Adding a non existing category to an existing todo
In this scenario, we make sure there exists at least 1 todo. When we try to add a category with an invalid id to a todo, the system should be aware that this category does not exist.

Scenario 20: As a user, I want to see which todo this project has.

Normal Flow: Getting the existing todo of an existing project
In this scenario, we make sure at least one project exists, and it has at least one todo. When we attempt to get the todo id of the todo, then the project should still have that todo.

Alternate Flow: Getting the done status of an existing todo of an existing project
In this scenario, we make sure at least one project exists, and it has at least one todo. When we attempt to get the done status of the todo id of the todo, then the project should still have that todo, and the done status remains the same.

Error Flow: Getting non existing todo of a project
In this scenario, we make sure at least a project exists, and it has at least one todo. When we attempt to get the todo with an invalid id of the project, then the project should still have the original todo.

Description of the source code repository

The source code repository structure for the JUnit tests was designed so that all JUnit Java classes were grouped within the same Test folder named 'test'. Within this folder, the 'CucumberTests' folder contains two new folders that were created for the Story Tests: the 'Feature' folder which contains all the Gherkin files; and the 'Definition' folder which contains the Java Definition files for each of the corresponding feature files.
The framework used was Cucumber in Java, to interpret our Gherkin files and link them to our JAVA definition files. We use a CucumberRunner JAVA file that will run all the tests sequentially.

Findings of story test suite execution

Data persistence between tests is important, because it can affect assertion and API request responses. Our tests assume that the database is reset between each test, but we had errors if data was saved between tests when the Application Instance failed to be killed (and then launched).
In some cases, it was necessary to add a thread.sleep after a post request. This is due to the fact that there is a small delay until the post request completes and the post request data gets added to the database. We do the thread sleep after a post request, to ensure our get request has the proper data.