ECSE 429 Project Part A

Name: Mario Bouzakhm ID: 260954086

Fall 2024

1 Project Deliverables

Cucumber is a behavior-driven development (BDD) tool that will be used to create and automate user stories and the tests that go with them for API improvements. The main thing that needs to be done is the creation of five separate feature files that show how different parts of an API work. Each feature must have a scenario description that lays out a methodical way to test that feature. For example, acts like adding, changing, or removing a resource (like a to-do item) are examples of scenarios. There will be three flows in each scenario: the regular flow, which is the case where everything goes well; an alternative flow, which is used when certain circumstances or inputs make the best way not work; and an error flow, which is used when wrong or invalid inputs are sent.

By automating these tests with Cucumber, the API's operation can be checked quickly and consistently. By using standard language and scenario plans to describe use cases, the project will provide clear, repeatable test cases that cover a wide range of situations. The outputs for the project are five feature files with scenario sketches and three flow types. This makes sure that all standard, option, and mistake situations are fully tested. This structure makes it easier to find problems early on and makes sure that all the important routes are tested to meet user needs and deal with a wide range of real-life situations.

Part A was mostly about exploration testing and unit testing of the API. This project builds on that work to provide a foundation for more in-depth, behavior-driven testing. In Part A, experimental testing let the team interact with the API and learn about its features, limits, and abilities. This real-world study showed likely edge cases and important functional areas that need to be tested more closely. At the same time, unit testing made sure that each part of the API worked correctly when used by itself. This step was necessary to build trust in the API's basic functionality and reliability, and it found early code mistakes before they could affect more complex tests.

Part A was very important because it gave us important information and basic confidence in the API's reliability, which let Part B focus on testing situations that were more focused on the users. After Part A checked the core's security through exploration and unit testing, Part B can now focus on user stories that show how things work in the real world and use Cucumber to automate them to make sure the whole process works. By building on the first testing step, the project makes sure that narrative tests closely match how the API is actually used. These tests cover a wide range of paths, such as standard, alternative, and error flows, while using the knowledge and trustworthiness gained in Part A. This method ensures a thorough, well-organized testing plan that meets both technical and user-focused standards.

2 Story Testing

Story testing is a way to make sure that a system works the way that set user stories say it should. User stories are clear, natural-language descriptions of what the end user wants. This method is very important because it links testing to what users want, which lets you check how well the system works in real-life situations. By organizing tests around user stories, developers and testers work together to reach a single goal that meets real user needs. This makes sure that features meet standards for functionality, usefulness, and dependability. Story testing includes many ways to test each story, such as standard, option, and mistake cases. This helps find bugs early in the development process, which improves software quality and customer satisfaction.

The first step in the story testing method is to write down user stories. Each user story should be a clear need from the user's point of view, like "As a user, I wish to create a new task to manage my to-do list." This simple structure, which is sometimes called the "As a... I want... so that..." approach, is easy to understand and makes sure that user goals are clear. After that, each user story is linked to acceptance criteria, which are the conditions that must be met for the feature to be considered finished. These standards form the basis for the scenario plans, which list separate test cases for three types of flows: normal, option, and mistake. Scenario sketches make generic testing easier by letting you try different sets of inputs within a single scenario. This method not only saves time but also makes sure that all possible use cases are covered completely.

Cucumber is used to simplify story testing, make Behavior-Driven Development (BDD) easier, and connect to Java through the Gherkin syntax for putting together feature files. The user stories and examples in Cucumber feature files are written in Gherkin, which means that both technical and non-technical parties can understand and use the tests. Using words like "Given," "When," and "Then," each feature file describes a scenario. These words describe the situation, what should

be done, and what should happen in each test case. These Gherkin steps are linked to Java code so that they can be automated. Each step in the scenario is linked to a different Java method that does the desired work. The Java code then talks to the API, checks the data, and tells the user whether the scenario worked or not. The team uses Cucumber to make sure that the expected behavior of each feature is fully recorded, can be repeated, and can be checked to see if the results are always the same.

There are many levels to the file format that is needed for effective story testing with Cucumber and Java. The ‘resources‘ section at the top level contains Gherkin feature files that each represent a user story and the events that go with it. The step descriptions that link Gherkin stages to Java methods are in the ‘src/test/java‘ directory. There are also helper classes for API searches, step definitions, and hooks in this area.

3 Explanation of the Five Feature Files

The first feature file, Create a Todo List Item, tests how well the API can make a new to-do item with different qualities. A 201 status number means that the usual process for creating a to-do item with a certain title, state, and description was successful. The alternative method checks that the item is made even though some extra traits are missing. With a 400 security number and a validation error message, the error flow makes sure that creating a todo item without a title fails.

The second feature file, Change Todo Description, checks whether the description of a current to-do item should be changed. The normal process checks that a good change to the description produces a 200 return code and changes the item correctly. The alternative flow checks the update process without changing the description. The error flow shows that trying to change an item that doesn't exist leads to a 404 error, which means that the correct todo object could not be found.

The third feature file, Add Category to Todo, checks that categories are linked to to-do items. The normal process checks to see if a category can be added to a task, confirming the status number and the new category count. The alternative flow checks that adding an extra group to another task can be done correctly. The error flow checks to see if there was a problem when trying to link a category to a todo that doesn't exist. It does this by making sure that the answer has a 404 status and an acceptable error message.

The fourth feature file, Mark Todo as Done/Undone, makes sure that you can change the status of a to-do item. The API changes the state of the task, and the return code shows that this has happened. The alternative process makes sure that the state reports that are given more than once are all the same. The error flow checks for a wrong status update by making sure that trying

to change a status that doesn't exist fails with an error message.

It is possible to remove a category from a to-do item, as shown in the sixth feature file called Remove Category from Todo. The normal process makes sure that getting a 200 success code when a current group is removed from a todo. The other flow tests leave out one more group. The error flow shows that if you try to delete a category from a todo that doesn't exist, you get a 404 answer, which means that you can't find the todo object.

In the story tests, the StepDefinitions class uses Utils methods to check if the system isn't down before starting tests. If the application isn't available, the rest of the tests are stopped. This first system check is done by the $test_application_is_running$ method, which makes sure that all tests only run if the application works as expected. Also, Utils functions like *startApplication* and *stopApplication* return the app to its previous state and control how long it runs before stopping. This reset makes sure that every test starts from scratch and that any data or changes made in earlier tests don't affect later cases. This keeps things consistent between tests. This method makes sure that every user story is correctly validated and increases trustworthiness.

4 Utils and StepDefinitions Classes

The Utils class has utility functions for interacting with the Todo API that handle tasks like starting and ending the app, making to-do lists, changing statuses and descriptions, adding and removing categories, and checking API replies. These functions handle responses for all functions and structure HTTP requests using RestAssured. Request headers are set up and JSON data is sent as the request body. Methods like createTodo, modifyTodoDescription, and addCategoryToTodo are used to send requests to the API. The Utils class also keeps track of important data, which is stored in static variables so that it is easy to get to between tests. Response status numbers, the most recent to-do and group IDs, and problem messages are all examples of this kind of information. This class is an important part of test validation, and it also has helpful methods to see if there are any to-do lists or groups with certain traits. The StepDefinitions class has step definitions for Cucumber feature files. This lets Gherkin steps written in natural language be linked to Java processes. Every method in this class is mapped to a Given, When, or Then step in a Cucumber scenario. These steps are used to do things like make to-do lists, edit descriptions, and check API replies. For example, $create_todo_with_desc$ and $create_todo$ use methods in the Utils class to run API actions, save the most recent answer state, and get error reports when needed. Also, checking methods like checkStatusCode and $check_error_message$ use statements to make sure that expected results are met, making sure that every situation fits the desired user story behavior.

This class helps with thorough, organized testing by clearly linking each Cucumber case to specific API actions and validation checks. It's also easy to keep the test suite up to date and add to it as API needs change.

5 Story Testing Findings

It was possible to run all 15 story test cases across all five feature files, which shows that the API works as expected in a range of situations, such as standard, option, and mistake flows. Each test makes sure that the system works the way the user wants it to, including all of the planned features and common mistakes that users are likely to make. According to the results, the system is sturdy and can handle different types of data and processes without failing in unexpected ways.

The API's "Create Todo List Item" function successfully created items with valid data, dealt with cases where extra elements were missing, and sent appropriate error messages when required values were missing. This proves that the generation endpoints do a good job of validating data and only letting calls that are properly organized pass.

The tests for the Change Todo Description feature made sure that the API can update an item's description, handle calls that don't change the description, find wrong IDs and send a 404 error when needed. This shows that the API can correctly tell the difference between things that exist and ones that don't and can change properties as directed.

All attempts to use the "Add Category to Todo" feature were successful, showing that the API can add categories to tasks, even when there are more than one category. The API gave the right error codes when trying to connect a category to a to-do item that didn't exist, so the error handling worked well.

The Mark Todo as Done/Undone tool showed that the API can change the state of a to-do item correctly, letting users mark things as finished or not finished as expected. The validation showed that when the status is changed incorrectly, clear error messages are sent. This shows that the API can handle both planned status changes and user mistakes.

The tests for the "Remove Category from Todo" feature showed that the API can properly separate categories from tasks and handle situations where the given task or category is not present. This behavior is necessary to keep exact classification free of leftover meanings.

All story tests passed, which means the API works well and handles data, checks for errors, and sends correct messages. The API works the same way in all situations, which shows that it is stable, easy to use, and ready for real-world apps.

6 Conclusion

All story tests for all five feature files were successful, demonstrating that the API is trustworthy and suitable for usage in the real world. All functions, such as task creation, description editing, item categorization, completion marking, and category tracking, were carefully tested in standard, alternative, and error scenarios, and all passed. This result demonstrates that the API matches user expectations and successfully handles a wide variety of inputs and edge situations. The testing suite produced consistent, independent findings by performing basic checks and resetting the system's state prior to each test. This ensured that the API functioned effectively, that the data was valid, and that errors were handled appropriately. These findings demonstrate that the API is robust and user-centric, making it simple to integrate and perform effectively in its intended environment.