Software Engineering Homework 1: Report

Anlang Tang

September 29, 2024

For unit tests: first, I needed to implement the `register_params_check` function. A key idea here is that I wrote a separate `validate_[param]` function for each parameter that needed validation; this made it easier to unit test the validation of each parameter individually. This also made the main `register_params_check` function very readable to look at and understand. For the validation functions, my implementation approach was to always test each condition separately and make use of early returns. This makes the code more readable, and more maintainable as it will be simple to add/remove specifications in the future. For most of the text validation, I tried to keep it to just Python code. I used regular expressions when it was much simpler than Python code and when regexpr itself was easy to debug, such as in the case of phone numbers or usernames. In the validation of URLs, for example, I first did processing in Python to extract labels in the domain and used regexpr matching only on the labels.

For the test cases, I wrote unit tests for each parameter's validation function as well as `register_params_check`. For the parameters' individual validation function, I tested negative cases by going through each condition in the specification and testing with an example that violated it. Then I tested positive cases with a few random examples that fit the specification. Since I already tested the validation for each parameter individually, testing `register_params_check` was simpler. I only needed to make sure the behaviour was correct when a given parameter was invalid/missing compared to when it is valid. I did this in the test case by creating a `params` object containing correct information for every parameter, and then for each parameter, I created a copy that modified said parameter to be incorrect in some way. This way I can create negative test cases to test `register_params_check`'s behaviour for each possible parameter.

By testing each condition of each parameter's validation, as well as the main function's behaviour for each parameter key, I was able to get over 90% code coverage.

For integration and end-to-end testing: For integration testing, implementing the tests as specified was fairly straight forward. First, I looked at the Swagger docs to understand which routes I need to be using, their expected parameters and method, and what they respond with.

For a positive test case, I will send valid data and assert that the response has the expected HTTP OK code and a message indicating success according to the Swagger docs. For a negative test, I will send invalid/missing data, and assert that the response has the expected ERROR code and a message indicating failure according to the Swagger docs.

For end-to-end testing, implementing the tests were a little bit trickier. First, I launched a development build to understand the HTML layout that Selenium would be looking at, and noting down the elements that a user would need to interact with to complete the given task. For example, a user flow might be: click the 'make post' button, find the input element for the post title, type some text, find the input element for the post body, type some text, click the 'submit' button. I would then make a note of all the elements that we need to interact with, and figure out how Selenium can locate it during automated testing. After doing this, it is simply a matter of writing out the sequence of instructions that we want accomplished, whether that is creating a new post or writing a reply. One thing to note is that an issue that may occur in automated end-to-end testing is timing issues with elements that may take some time to load. To keep it simple for this homework, I have just used `time.sleep` to give elements/the page time to load, though there are other strategies in Selenium that are more robust.