QA Test Suite

Part 1: Web UI Testing

First things first, I’d start by setting up a clean environment. I’d create a new GitHub repo so I can keep everything organized and have a clear history of my work. Then, I’d initialize a Node.js project and install Playwright it’s a fantastic tool for browser automation and makes writing tests pretty straightforward.

Once everything’s set up, I’d generate a default Playwright config file. This helps me customize things like the number of parallel tests, retries, and the format of the reports (like HTML or JSON). I’d want my setup to be flexible and easy to run.

Next, I’d dive into planning the actual test cases. For the upload page, I’d cover the basics: making sure the page loads properly, the upload input exists, and the upload button is enabled only when a file is selected. I’d write tests to upload a sample file and verify that the success message appears. Also, I’d test what happens if someone tries to submit without choosing a file it should show an error.

While scripting, I’d make sure to take screenshots if a test fails that way, I can easily see what went wrong. I’d also check the accessibility of the page, making sure labels and buttons are correctly named.

To keep things tidy, I’d organize my tests into separate files, maybe one for setup, one for upload tests, and so on. I’d run my tests locally first, then generate reports so I can review the results visually and share them if needed.

Finally, I’d think about automating this in a CI/CD pipeline later, but for now, I’d focus on making sure all tests are reliable and easy to run.

Part 2: API Testing

For the API part, I’d follow a similar approach. I’d create a dedicated folder for API tests and use Playwright’s API request capabilities, which are super handy. I’d set the base URL to the JSONPlaceholder API and write tests for each endpoint.

For example, I’d start with testing GET /posts — making sure it returns a 200 status and an array of posts. Then, I’d check GET /posts/{id} to verify that fetching a specific post works correctly. For creating a new post with POST /posts, I’d send some sample data and check that the response confirms the creation with the right data and a 201 status.

Updating a post with PUT or PATCH would involve sending new data and verifying the changes are reflected. And for deleting, I’d delete a post and then try to fetch it again to make sure it’s gone.

Throughout, I’d add tests for edge cases like trying to get or delete a post with an invalid ID, or sending invalid data to make sure the API handles errors gracefully.

Like with the UI tests, I’d generate reports, log responses, and take screenshots if anything goes wrong. I’d also keep my code modular, so if the API changes or I need to add more tests later, it’s easy to do.

Wrapping It All Up

In the end, I’d organize everything clearly in my repo, with a README explaining how to set up and run the tests. I’d include instructions for running both the web UI and API tests, and how to generate reports. I’d also add some notes about the architecture why I structured things a certain way and mention any challenges I faced, like handling file uploads or managing flaky tests.

Throughout the process, I’d commit often with meaningful messages, so the history is clear and easy to follow.

Overall, my goal is to build a reliable, maintainable test suite that covers all the key functionality, provides clear feedback, and demonstrates my skills in automation and test design.