Testing

Sdmay23-06

Mindset

Our team is going to build the product in a test-driven development environment. Test-drive development is a style of programming which includes three tightly interwoven activities: coding, testing, and design. The benefits of doing this is that it usually reduces the amount of defect rates.

Test-driven development follows a set of rules to work as efficiently as possible. The first rule is to write a single unit test that describes an aspect of the program. Next, run the test. This will obviously fail because the code base doesn't have the feature yet. After the first test, write the minimal and simplest amount of code to make the test pass. After the second test passes, refactor the code so that every line, method, and class has a distinct purpose, idea, or responsibility. The refactored code should also have a minimum number of components, and it should have no duplication. Once the refactored code is complete, keep repeating until the program is finished with all of the features.

The team believes this is the best way to develop the product. The set of rules will keep development in progress without running into poorly written code that can't pass tests. The rules make sure that every single line of code has a purpose. TDD is a great style of programming for this project.

Unit Testing

Front-End

Unit testing for the frontend will consist of utilizing Jest which is a viral library that allows developers to conduct unit tests on React. By using Jest's watch mode, we will be able to generate snapshots of our code which is then compared to previous snapshots. If the snapshots differ, we will be able to trace where in our code the tests failed, or produced unintended results. These snapshots will be vital when conducting unit tests because they help us identify areas of our frontend where components changed where they should not have.

The units that we will be testing are the user login page, generate report page, and the generated report page. These three components must all perform correctly in order for our product to work. By ensuring that these all are thoroughly unit tested, we can deliver a product that is up to the standards of the team and client.

An example of a unit test that we will run would be creating a snapshot of the generated report, then generating the same report again. By comparing the snapshots, we can verify that the components display the correct data and are working as intended.

Back-End

Unit testing for the backend will be pretty simple. ASP.NET has the ability to create unit tests for controller actions. Since we will most likely only have one controller, we will only have to create one test controller. The test controller will check to see if the data is correct and properly formatted. To test the log file, we will just feed mock data to the log file. If the data being sent and the data showing up in the log file match, then we know the log file is complete.

An example of a unit test would be creating a test controller for the Work Order controller. This will consist of making a Work Order test object with specific data. Then we can do a check to see if the API call of the Work Order test object is equal to the expectation.

Interface Testing

For interface testing there are numerous different routes we could take, user testing is going to be our first main form of interface testing. The plan moving forward will be to have both a mix of engineers and people who have never coded before.These groups will comb through the website to test every link and endpoint available. The mixed demographic is important here because engineers are nice for checking edge cases and things that will most likely break the site. Meanwhile, having a non-engineering perspective can be nice for testing the usability of the site and receiving input on how the common user feels about the interface.

Additionally, any formal testing will be completed using a framework called LambdaTest. Lambda test makes automated GUI testing easy and fast. This may not end up being necessary, but additional testing never hurts. With the majority of our code we will test on a regular basis to avoid any foundational errors. However, the majority of the testing will take place after the interface is already made due to the nature of interface testing.

Integration Testing

For integration testing we plan on using Robot Framework, it is a python library that makes automated test cases easy to run with easy to interpret output. One of our group members is also very familiar with RF, so it will reduce the learning curve.

We found our critical integration pathways by looking at our software architecture as well as our object structure. A key integration test would be linking the front end request button to the back endPDF calls.

List of some of our critical integration parts:
(Front End) Request PDF with (Back End) PDF Generation
Pulling Data and PDF Generation
(Front End) Email PDF with (Back End) Email
PDF Generation with Log File

A sample test case for PDF generation and log file integration (written in robot framework style code):

```
PDF Gen Log File Test
    Generate PDF        ${sWorkOrder}
    @{sLogFile}=        Read Log File
    RUN KEYWORD IF  '${sLogFile}' not in @{sLogFile}    Fail    'Work Order not found in file'
```

System Testing

Since system testing involves the entire scope of the system we will have to try “end to end testing”. This typically means that we will be testing different inputs on various parts of the system to make sure that they yield the desired output. For example, a test input would be given to the front end of the system while a desired output would be tested on the back end of the system. Since multiple tests will have already been done in order to make sure that subsystem outputs are correct, this would allow us to check to make sure that an input can travel through the entire system without throwing any errors or producing an incorrect output. While the majority of our tests will be written to test individual components, after we finish our integration we will run end-to-end tests to make sure that the system works correctly.

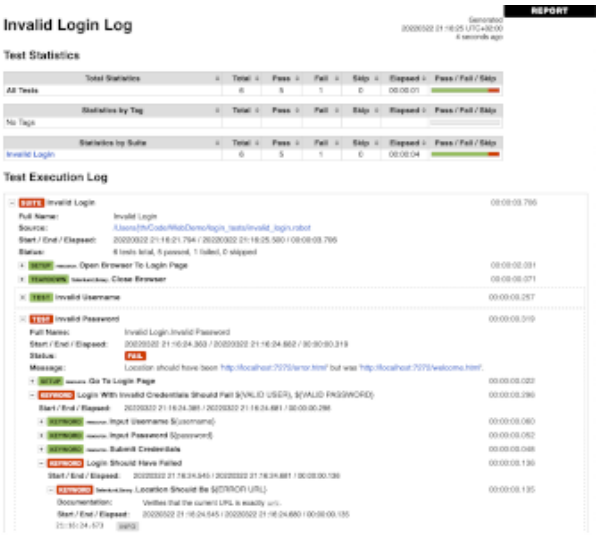
We plan to use the versatile framework “Robot Framework” as mentioned in previous sections. For more information on how we plan to integrate Robot Framework see the integration testing portion of the document. For more information on how we plan to receive and use the output of Robot Framework see the results portion of the document.

Acceptance Testing

This portion of testing will consist of two things. The first form of demonstration will be going through the requirements list. Each functional and nonfunctional requirement will be reviewed to see if it is being met. The last form of demonstration is to show the client a “beta” version of the product. This will ensure that the product has all of the requirements desired by the client. Although most of the product is already done, we might be able to change some small things that the client desires. For the most part it will be demonstrating to the client that the product meets the requirements list.

Results

Our results from Robot Framework will be automatically formatted into a report, log, and output file, below is a picture from Robot Framework's website where they display a test result. We can see that it breaks down test cases into steps, time for each step and case, inputs and outputs, as well as reasons for failures. This makes it very easy to interpret the results, and when we want to show our client our testing results it will not be complicated for them to understand.



Testing will ensure compliance with our requirements as all of our requirements are testable, and if we add any non testable requirements we can add a FIT criteria which will make it testable.

In theory we should be able to have a test suite/case for every individual requirement, and with RF's auto formatting the results will be easy to verify we meet with the specified requirements.