Soen390 - Team 11 - Testing Plan

1. Testing Team

Lead: Sunny Deng
Contact: sunnydeng90@gmail.com

Internal Testers

Elie Sabbagh
Israt Noor Kazi
Kevin Zhong Hao Li
Kim Chheng Heng
Omar Salahddine
Saebom Shin
Shijun Deng
Yu Xiang Zhang

2. Testing Procedures

General Approach

As per requirements, all controllers and subsequent services of the MVC pattern for the ERP system must be tested with a line coverage of at least 50% for the controller classes. In order to test them, each controller and service should have their functionality tested separately. In other words, each controller and service will be unit tested.

The testing should be done with the help of the JUnit library, Mockito library, and Spring JUnit runner if necessary. The reasoning behind the use of JUnit is because the back-end, written with the Spring framework, has an official support of the framework as seen in a guide hosted on the official Spring website. This official support allows developers to start up a mock version of the developed application in order to test any of Spring's functionality that the program may be exploiting. For example, the use of @RequestBody in the controllers.

These tests should be separated into their separate folders mimicking the package levels of the application and test files should be akin to one linked class.

1. Daily Procedure for Dev Team

1. Write the updates and changes into the code
2. Write their JUnit tests
3. Run the tests with the help of Gradle
4. Fix any failures that may occur
5. Update the classes and tests as needed with changes in the class they test

Note that nothing should be merged into the main branch of the project until the tests have successfully passed. Furthermore, the written tests should reflect the functional requirements of the implemented feature. The tests for non-functional requirements are planned to be written at a later Sprint.
Furthermore, the tests should cover not only success cases, but also failure cases. The success and failures is dependent on the implemented feature, thus no standard exists that covers everything. An example is shown here.

At the end of a sprint, all branches that are ready should be merged with corresponding tests. All tests should also be able to pass when run concurrently, thus no tests should be failing by the end of a sprint.

2. Sprint Procedure for Dev Team

1. All branches and requirements that are ready should be merged into the main branch
2. All tests should be run with a success rate of at least 80% of all tests.
3. Any failures must be fixes before end of sprint

3. Testing Requirements Generation

1. Requirements for the tests are generated by their use cases
2. Other requirements are from the written code as it must reach, at minimum, 50% of code written for the controller in the back-end
3. Requirements can also be generated when unintended results are found. These should be reported by the finder to the team. Furthermore, a github issue should be generated with the steps to reproduce said bug in order to confirm whether it is present on the program or just the finder's system.

4. Bug Tracking Software

Zenhub will be the main software used in order to track the bugs. Github, will be the secondary software as Zenhub interacts with Github when any new issues are created. Bugs should be given the "bug" tag, one of the classification shown in the next section, and instructions on how to replicate the issue.

The Zenhub link can be found here.
All bugs found should follow the scheme below when creating an issue for it. The issue's title should be representative of the type of bug it describes

Bug Report Scheme

Description:

Give a short description of the bug. Namely on what it is affecting, how it looks like, and the results of such bugs. Any other pertinent information should also be written here.

Steps to Reproduce:

Give a step by step instruction on how to reproduce the bug that was found. Use short sentences and be very clear about the steps. Furthermore, if there is more than one way to reproduce the bug, mention it and give a separate step by step instruction to reproduce it.

5. Bug Classification

1. **Major**

Any issues classified here would pertain to anything that could potentially crash the system, change the normal process of a request, or leak sensitive information. For example, any bug that could leak the payment numbers or any users password in plain text.

2. **Minor**

Any issues here would be textual errors or exceptions that would not crash the system nor interfere with the normal process of a user's request. For example, a console log that is done on the client's side that was done to test some functionality and that a developer forgot to remove.

3. **Unreproducible**

Any issues here are those that occurred in one system, but could not have been reproduced in someone else's. This classification should only be placed after it was tested on both a similar and different system. Of course, this is a bug that is to be left in the system in order to showcase that there is a possibility of an issue as it was described.

4. **Won't Fix**

Any issues here are those that have been investigated and found to be unable to be fixed because of other issues it may bring. Furthermore, this is also for any bugs that have been in the "Unreproducible" classification for more than one sprint and it had been failed to replicate by two other members of the development team.

6. Bug Tracking

The bugs that are created should be written by the one who finds it and they have the responsibility to find someone else to reproduce their bug. Afterwards, when sufficient information is found, they should classify the bug based on that information and the description of each classification seen in section 5.

When minor bugs have not been fixed near the end of the sprint, then they can be pushed onto the next sprint. This is namely because they cause no issue to the expected output of the service.

However, major bugs should be fixed as soon as they are discovered. If they are not fixed by the end of the sprint, then a workaround should be expected to be given in order to give more time for the developer who found the bug to fix it or for them to find another developer to help them fix it. These bugs should not be present for a longer time than two sprints.

The bug should be assigned to anyone who is capable of investigating the bug and potentially fixed. If it is impossible to fix, then it should be given the “Won’t Fix” classification by the end of a sprint as shown in section 5.

When the bug is fixed, the fix should be tested and merged in order to confirm that it no longer exists. This implies that the finder will be assigned as the one who will be confirming if the applied fix correctly mitigates the bug.

7. Code Convention

The code conventions used for this project will be google’s java style guide.
This convention should be verified upon each Pull Request in order to make sure that it is properly being followed.

The main reason it is chosen is because it is the style that has been taught in previous courses as well as it being made by a professional organization. Thus, the guide will be concise as well as complete.