Testing Policy

MonoToneID

May 2019

1 Introduction

1.1 Purpose of Document

The purpose of this document is provide details on the testing processes applied in developing EISHMS.

1.2 Purpose of Testing Policy

The MonoToneId testing policy aims to demonstrate the satisfaction of requirements as mentioned in the software requirement specification. This will include the functional and quality requirements of the system. Additionally, it aims to identify errors (logical, semantic and computational) and undesired behaviour. Undesired behaviour refers to system behaviour that is not stated in the requirements which can be exploited to compromise the system.

1.3 Testing Process

1.3.1 Style Check

Our testing process includes linters which are configured to assess the coding style. The code is evaluated against what is specified in the coding standard document, to facilitate uniform and readable code.

1.3.2 Peer Reviews

Furthermore, our team dynamic allows us to practice peer programming. We review each others' code and check if style requirements are met and a certain standard of code is maintained, as mentioned in the coding standards document. (`https://github.com/cos301-2019-se/EISH/blob/admin/Documentation/CodingStandard.pdf`)

1.4 Automated Testing

1.4.1 Travis CI

We use an automated testing tool Travis to automate our testing. Travis builds our backend using Gradle . The Gradle build starts by running the check style file to check coding standards violations, then runs the tests in the testing folder. Travis ten builds the frontend application by running ng lint for checking coding standards violations which is followed by ng test that runs tests before building the project.

1.5 Testing Tools

The EISHMS is based on a custom architecture. The core architecture is made up of an MVC architecture that houses an n-tier component and event-driven component in the controller of the MVC. The testing tools for the different components will be discussed below:

1.5.1 Backend

1. **JUnit:** JUnit serves as a foundation for launching testing frameworks on the JVM(Java Virtual Machine). JUnit uses annotations to easily differentiate between test application and application that will be deployed. JUnit also includes an assertion library which simplifies the writing of unit tests. See (`https://junit.org/junit5/`) for more information.

2. **Mockito:** Mockito is an open source testing framework for Java which allows the creation of test double objects in automated unit tests for the purpose of test-driven development or behaviour-driven development. See (`https://site.mockito.org/`) for more information.

1.5.2 FrontEnd

1. **Karma:** Karma is a task runner for our tests. It uses a web server that executes source code against test code. The results of each test are examined and displayed via the command line to the developer such that they can see which browsers and tests passed or failed. See (`https://karma-runner.github.io/latest/index.html`) for more information.

2. **Jasmine:** Jasmine is an open-source behavior-driven development framework for testing JavaScript and TypeScript. It does not depend on any other JavaScript frameworks. See (`https://jasmine.github.io/`) for more information.

3. **Protractor:** Protractor is an end-to-end test framework for Angular applications. Protractor runs tests against your application in a real browser, interacting with it as a user would. See (`https://www.protractortest.org/#/`) for more information.

1.6   Test Cases

1. **Backend:** All unit and intergration tests are located in the following folder of the system. (`https://github.com/cos301-2019-se/EISH/tree/master/Backend/eishms/src/test/java/com/monotoneid/eishms`)

2. **Frontend:** All frontend unit and intergration tests are located in coverage folder. (`https://github.com/cos301-2019-se/EISH/tree/master/Frontend/coverage/EISHMS/src/app`) to run the test run command ng test.

1.7   Test History

Our test reports for Travis can be view at (`https://travis-ci.org/cos301-2019-se/EISH/builds`)
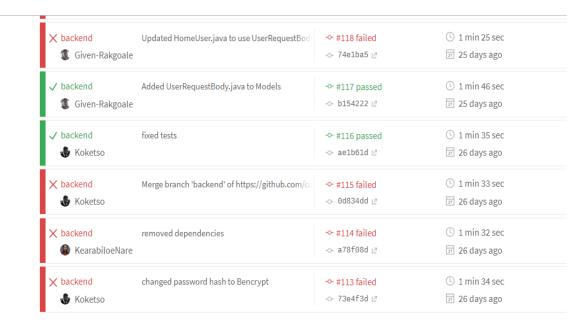

Figure 1: example of build history

```
public ResponseEntity<Object> getCurrentUserPresence(Long userId) {
    try{
        usersRepository.findById(userId).orElseThrow(() -> new ResourceNotFoundException("user does not exi

        HomeUserPresence foundHomeUserPresence =userPresenceRespository.findCurrentHomeUserPresence(userId)
        boolean isPresent = foundHomeUserPresence.getHomeUserPresence();
        JSONObject responseObject = new JSONObject();
        if(isPresent){
            responseObject.put("homeUserPresence",  "User is home!");
            return new ResponseEntity<>(responseObject,HttpStatus.OK);
        }else{
            responseObject.put("message","User is not at home!");
            return new ResponseEntity<>(responseObject,HttpStatus.OK);
        }
    }catch(Exception e){
        System.out.println("Error: " + e.getMessage() + "!");
        throw e;
    }
}
```

Figure 2: example of linter

```
62
63    it('form should call register credentials',()=>{
64        router.navigate['register/Register']
65        sessionStorage.setItem('username','Tom');
66        component.credentialsForm.controls['userName'].setValue('Thomas');
67        component.credentialsForm.controls['userPassword'].setValue('123456789');
68        component.credentialsForm.controls['userDeviceName'].setValue('myIphone6');
69        component.credentialsForm.controls['userEmail'].setValue('test@email.com');
70        fixture.detectChanges();
71        spyOn(component,'registerCredentials');
72        expect(component.registerCredentials).toHaveBeenCalledTimes(1);
73    });
```

Figure 3: example of front end unit test

4

Figure 4: example of front end unit test coverage