2

SYSTEM DESCRIPTION

The system to be designed will have to perform the function of *Login System*, and therefore must allow the user to insert their data and store those user credential in database and allow user to re-login with same credentials. The system will then have to provide the user with the functionalities necessary for carrying out the following operations:

- **System Implementation is given below:**
  1. **SignUp**-Frontend page for User- It collects user's data
  2. SignUp has the fields of **email**, **password**, **usernumber** and **usename**.
  3. **Save** – Save user's data on database
  4. **SignIn**- Here user can re-login with stored credentials.
  5. **Access denied** – If user login with wrong credentials

  This whole project is implemented in *Java* language.The user can interact with the system with a *GUI* (Graphical User Interface) using spring boot technology. This project is implemented with **Test-Driven Development, Build Automation, Continuous Integration**.

To test the whole module I used different frameworks and those formwork were implemented correctly in the project. The next chapter provides more technical information on these frameworks.

3

APPLIED TECHNIQUES, TECHNOLOGIES AND FRAMEWORKS

As easily understood, this report does not place particular emphasis on complexity of the application, but on the use of a set of techniques, technologies and frameworks for the system development, different from traditional cascade development. In particular, the system must be developed through the techniques of *TDD*. The *TDD* is based on development cycles.

In addition to the *TDD*, a variety of technologies, frameworks and tools will be used, the functionalities are briefly described below:.

Technologies, Frameworks:

1. **Operating System**: The project completely developed in *Windows 10* , however it can be implemented other operating system including *Linux*.

2. **Java**: A high-level, object-oriented programming language and a static typing. During the project it will be used as the main language programming for writing the source code.

   -> Version used during development: *Java 8*

3. **Eclipse**: cross-platform and multilingual IDE. During the project it will be used mainly as an editor for writing the largely written source code in *Java* (*Java 8*, in particular).

4. **JUnit**: framework for writing tests. During the project it will be used for it *Unit Test* and *Integration Test*.

   -> Version used during development: *JUnit 4*.

5. **AssertJ**: library for inserting assertions into tests. Will be employed for improve the readability of tests.

6. **JaCoCo**: Code Coverage tool
   -> *EclEmma*: Eclipse plugin for *JaCoCo*.
7. **PIT**:Framework for Mutation Testing.
   -> *Pitclipse*: Eclipse plugin for *PIT*
8. **Maven**: Tool for build automation of *Java* projects. It will be used for automate the dependency management, build, test and report process of the results (i.e. the build process), both locally and remotely.
   -> *M2Eclipse*: *Eclipse* plugin for *Maven*.
9. **Mockito**: Framework for Mocking. It will be used for mock dependencies during the *Unit Test*.
10. **Git**: Distributed VCS
    -> *EclEmma*: *Eclipse* plugin for *JaCoCo*.
11. **Travis CI**: Server CI for Remote Build.
12. **Spring Boot**: The project is carried out using the *Spring Boot framework* Java used for building web applications.
13. **Docker**: Tool for virtualizing services (such as databases) on containers.
14. **sonarcloud**: The *sonarcloud* platform for code inspection and code quality estimation. can he be used in conjunction with *JaCoCo* to get an estimate of *code coverage*.
15. **MongoDB**: Non-relational, NoSQL, document-based DBMS

The technologies mentioned above are implemented during test the development cycle. How we use these tools are more important than the complexity of the project.

All the tools mentioned above available for free on the Internet. To understand the functionality of these tools well, I referenced the book written by *Prof. Lorenzo Bettini*, *Test-Driven Development, Build Automation, Continuous Integration with Java, Eclipse and friends*, Leanpub, 2018.

4

SYSTEM FUNCTIONALITY DESCRIPTION

The main features of the system are described here:

Below features are tested during devloped:

1. SignUp for user
2. Save User profile
3. SignIn
4. Access denied

The “Login System” project can be used by any company to log in to their application safely. All data relating to user information that will be stored and retrieved whenever necessary.

4.1 SIGNUP:

SignUp is a web page where we obtain user data. User data such as *email, password, telephone number* and *username* are visible in this form. For each user profile, there will be an automatically generated id and will be saved in the database along with other user data’s. When a user create all these values will be save in user profile.

Below figure shows *SignUp* page for the "Login System" application. (Figure 1 on the following page)

Figura 1: SignUP Page

4.2 SAVE USER PROFILE:

In this session we can save all the user's data along with unique id of each profile. All the data will be saved in *MongoDB*. Each profile will be saved as a document in *MongoDB*. Below figure shows *Saved* page for the "Login System" application. (Figure 2)

Figura 2: User data Saved Page

4.3 SIGNIN:

SignIn is a web page where the user can log in with their credentials. The user must pass two parameters which are the user's *email ID* and *password*. Once the user has provided the appropriate credentials, the user can successfully log into the system.

Below figure shows *SignIn* page for the "Login System" application. (Figure 3)

Figura 3: User SignIn Page

4.4 ACCESS DENIED

Here we implemented some rules for SignUp and login into the system.

When it comes to SignUp following rules are implemented:

- It is not possible to create multiple users with the same *email ID* and *username*. This means that the *username* and *email ID* must be unique for each user profile.
- With same phone number and password we can create multiple users.

When it comes to SignIn, implemented following rules:

- Access should be denied if user enter wrong *e mail id* .

- Access should be denied if user enter wrong *password*.
- Access must be denied if the user's *email* and *password* are not saved in the database.

Below figure shows One of the *Error* page for the" Login System" application. (Figure 4)

Figura 4: User Login Error Page

The System must notify the user of the outcome of the operations carried out, both in case of success and in case of error. All the functionalities are tested during the development using above mentioned tools in the report.

During development, I made sure that the test had to pass for each module. If the test fails, I've made the code changes accordingly. Then retested and continued with another module. This is what actual purpose of this project. How tests are separated according to each module was explained in the chapter 6.

5

SYSTEM ARCHITECTURE AND DIAGRAM

The brief preliminary analysis phase, it was considered to have a layered architecture, which we can in some ways see as an extension of the *Model-View-Presenter* architecture. The system in fact has the typical components of the *MVP* architecture, namely the *View*, the *Model* and the *Controller / Presenter*, additionally I have added two other layers: The *Service* and the *Repository*.

5.1 VIEW

This layer is responsible for showing the information of interest to the user, taking his own input and notify the user about the outcome of the operations. For this project, it was decided to implement only in *GUI*. There *GUI* will be implemented using Swing (i.e. css and html used for building graphical interfaces).

5.2 CONTROLLER

This layer takes care of taking the information and requests sent by the user, through the view, and to send high-level requests (such as, for example, the creation of a user profile) to the service layer. The controller also takes care of via the view, notify the user about the outcome of the operations and update the status of the View.

5.3 SERVICE

This layer is in charge of providing high-level services to the controller. These services are related the logic of the operation of the application,

such as the one it manages the insertion of a new profile for the user.

5.4 REPOSITORY

The repository layer deals with the execution of low-level operations that involve the reading and writing on the database. Here we use UserRepository() interface and which extends MongoRepository(). We implemented method called *findByEmail ()* , where we can search user by their *e mail id*.

5.5 MODEL

The model contains the classes of the application domain. Model classes describe entities that are manipulated during the operation of the "User Login" application. Using such an architecture, the model classes are independent of the technology and used for collection to save data in the database. In model we have user details and the which has *email; password; username; usernumber;*.

Below figure shows *system architecture* for the "Login System" application. (Figure 5)

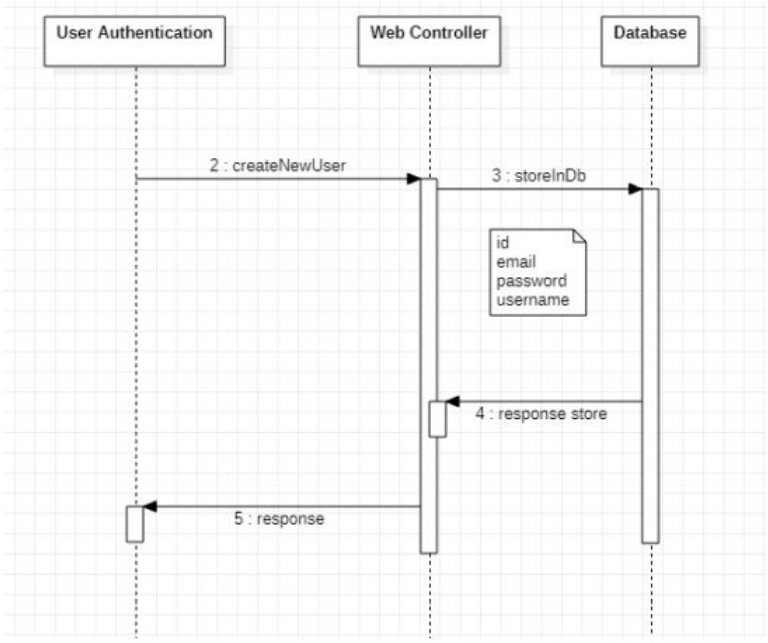

Figura 5: System Architecture

6

TESTING AND DEVELOPMENT

This chapter is important because it describes the details of test phase and its implementation.

6.1 UNIT TESTING

Here we tested each module. In general, we can say that not all classes "deserve" a test. They come in fact test the classes that contain some "logic" in them. A typical example of classes that are not tested are the classes that represent the entities of the domain application (in this case, the class user.java), as the only methods that possess are the getters / setters and the override of equals and toString, which have been generated automatically through Eclipse, and which do not possess enough complexity to render.

6.2 INTEGRATION TESTING

After the individual components have been implemented (and thus also tested in isolation through unit testing), during the integration testing phase it is verified that components of the web application continue to work when they are integrated with each other and with third-party components (such as, for example, a database). Since this type of testing is implicitly slower than unit testing, they are tested those that are the "positive" cases; in other words, not all possibilities are tested scenarios, also because it is often not feasible to easily create tests for "negative" cases render.

A CI server is also used (see Build Local and on *CI* Server) for carry out all tests on a remote server, thus avoiding using local resources. During this phase, the classes that implement the repository interfaces were tested together with “real” *Mongo DB*, which runs on a *Docker* container.

The tests were created and run following a bottom-up approach:
it was tested for first the interaction between the repositories and the db (run on *docker*); subsequently it was tested the interaction between the underlying components; to follow was tests the interaction between the controller and the underlying layers and the interaction between the views (it was created a test for each view implementation) and the underlying layers.

6.3 E2E TESTING

In this phase the entire application is tested. The main difference between an *E2E* test and the integration test is that the interaction takes place exclusively through the user interface.

6.3.1 *E2E test of the GUI*

While testing entire application I focused on following cases. Almost 5 cases are tested and verified. These test result are verified in *CI* build with *docker*.

1. User Login and Logout.
2. User Login with invalid password.
3. User Signup with email already in use.
4. User Signup with User already in use.
5. User Signup with Usernumber already in use.

Number of tools used to implement all tests. The features and use of these tools were explained in Chapter 7.

7

TOOLS FOR TEST DRIVEN DEVELOPMENT

Here in this Chapter 7 now present functionality of various tools used for *TDD*.

7.1 DOCKER

Installing a *MongoDB* server (or other servers, such as those for *MySQL* or servers Apache) on the machines used for development is generally not recommended, as it results in a waste of resources and an increase in the complexity of the configuration between members of the development team. It was therefore considered appropriate to virtualise this service.

For the virtualization of the *MongoDB* server was used *Docker*, which provides the possibility to virtualize through containers. The main advantage of using *Docker* container, in addition to the resource savings given by the fact that *Docker* uses the same kernel of the host operating system, is given by the high reproducibility that an operating environment development has, when compared to other virtualization techniques, such as the use of virtual machines.

We used a particular *Docker* image that allowed us to use the transactions more congenial to the structure of the application than what it would have been possible with a normal *MongoDB* image. During tests performed on user profile ' machines, the *MongoDB* container is configured and launched within *JUnit* tests via Test containers or via Process. For *MongoDB* the Generic Container class is used. During normal application execution the *docker* container containing the server *MongoDB* must be started before the application starts (and eventually closed after the application starts). closing the application).

7.2 LOCAL BUILD AND ON SERVER CI (MAVEN AND TRAVIS CI)

During the entire development of the application, *Maven* was used, to have a tool for build and project management, especially as regards dependency management. When a project build happens, *Maven* takes care of download (if necessary) all dependencies, compile the sources, run the tests, and generate reports. Maven also greatly improves the reproducibility of the environment of development, as all dependencies are managed automatically The project was divided into several modules, in order to increase the modularity and the independence of the application components.

Maven allows you to build your project locally. However, the main advantages of *Maven* emerge when it is placed side by side with a VCS (in our case Git) and a Ci server (in our case *Travis CI*).

Travis CI is used as a server for Continuous Integration. The use of a *CI* server allows you to build the entire application (which could be very expensive when done locally) on a remote server. In the simplest case it comes a build of the app made with each push operation on the repository (*Travis CI* is linked to the use of *GitHub*). The build execution environment on the *Travis CI* server is defined within a *YAML* file (i.e. the *.travis.yml* file). Mutation testing is also performed during the build, and used the plugin *Maven* from *PIT*.

7.3 GITHUB

During the development in group (moreover remotely) of a project, one is required tool that allows the sharing of the code (and related resources) that makes it possible keep track of changes made to the project. These features are provided by *Version Control Systems*. During the development of this project, *GitHub* was used. The use of *GitHub* allows for easier integration with previously mentioned *CI* tools (i.e. *Travis CI* and *SonarCloud*), which go well together with *GitHub*.

7.4 MONGODB

From the point of view of the "structure" of the DB, there is one collection, to store the User profile. Transactions can be implemented using the native APIs provided by Mongo.

7.5 CODE QUALITY WITH SONARCLOUD

The *SonarCloud/SonarQube* is a platform for continuous inspection of code quality, which measures the reliability and maintainability of the application, through *static code analysis* source. Code quality estimation can be done both locally, using a stack bootable via *docker*-compose, or on the cloud. In this project I have used *SonarCloud* and integrated with *Travis CI*.

After the analysis (carried out during the build *Maven*, after the test phase), *Sonar* reports all the "Issues" related to the code, which can make the code less secure, reliable, and maintainable. The most important issues have been corrected by manually editing the code. Using *SonarCloud* together with *Travis CI* and *JaCoCo* (defined as a plugin in the POM), it was measured the code coverage by appropriately excluding the model classes and the class with the main method; these classes are excluded from code coverage as they do not present logic.

7.6 SPRING BOOT

The project was carried out using the *Spring Boot* framework Java used for building web applications. To development all applications based on the Spring framework needed a web server at the base (Tomcat, Jetty, etc.) to be able to be executed; with Spring Boot, however, the execution of an application starts an embedded container (Tomcat by default). Being a dependency injection container, it implements, by means of one library, dependency management mechanism: Spring injects at instead of the programmer, the dependencies necessary for the application classes, in perfect line with the Inversion of Control (IoC) mechanism. In particular, Spring Tool Suite, a development environment, was used integrated derived from Eclipse. Spring Boot takes care of instantiating all the objects (beans) declared in the project, loading the application context when the application is launched: @SpringBootApplication is the annotation that matters @Configuration, @EnableAutoConfiguration and @ComponentScan, annotations that give life to our Springproject.

7.7 COVERALLS

This is an online free tool. Ensure that all code is fully covered and helps us to see coverage trends emerge. It Works with any CI service. Additionality it shows the latest code-coverage statistics on all our repositories including the total percentages covered and the lines covered. This tool also incorporated with Login System application.