MY BOOKS APP PROJECT

A DETAILED DOCUMENTATION OF TESTING PRACTICES CARRIED ON "MY BOOKS APP" PROJECT.

Prepared by :

1. Anand Nair - 163680
2. Janardhanan SR - 163602
3. Venkatesh - 145152
4. Rajendra Varma - 140631

Submitted to :

Koel Chowdhury,
STACKROUTE TEAM

TABLE OF CONTENTS

ACKNOWLEDGEMENT

We are extremely glad to present our project as a part of our training. We take this opportunity to express our sincere thanks to those who helped us in bringing out the report of my project.

I am deeply grateful to Mr. _____,Stack Route , for their help and suggestions throughout the project.

Express heartiest thanks to **Mrs. Koel Chowdhury**, Project Scrum Master for her encouragement and valuable suggestion. Finally, we express our thanks to all of our team members who gave us their valuable suggestion for successful completion of this project.

Janardhanan S.R(163602)

Venkatesh(145152)

Rajendra Varma(140631)

Anand N Nair (163680)

1.INTRODUCTION

Online Library Management System is a system which maintains the information about the books present in the library, their authors, the members of the library to whom books are issued, library staff and all. This is very difficult to organize manually. Maintenance of all this information manually is a very complex task. Owing to the advancement of technology, organization of an Online Library becomes much simpler. The My Books App has been designed to computerize and automate the operations performed over the information about the members, book search and saving to the favourite list. This computerization of the library helps in many instances of its maintenance. It reduces the workload of management as most of the manual work done is reduced.

1.2 Project Aims and Objectives

The aims and objectives of this project is described below:

- Account creation in the My Books App.
- Login facility for the library.
- Searching functionality made available for the user.
- Users are allowed to save the books to the favourite section.
- Users were allowed to delete the saved books later.

1.3 Testing Aims and Objectives

- Testing the register functionality in both backend and the frontend.
- Testing the login functionality.
- Testing the search functionality in the frontend.
- Testing the add to favourite functionality in frontend and backend.
- Testing the delete functionality in the backend and the frontend.

1.4 Overall description of testing techniques.

For the testing purpose there are two modules provided:

- UserService module.
- FavouriteService module

In the modules mentioned there we are testing in both the frontend and backend section. For the backend purpose we are using technologies like Mockito+J-unit testing and Rest-Assured+TestNg testing. For the frontend testing we are using the Protractor-Jasmine testing framework.

Backend Testing frameworks:

- Mockito+J-unit:

  Mockito is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito greatly simplifies the development of tests for classes with external dependencies.

If we use Mockito in tests we can typically:

- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly
- We can expect the output with the help of assertions.
- The major benefit is that without the help of databases we can directly inject the mocked values to the class we intended to inject.

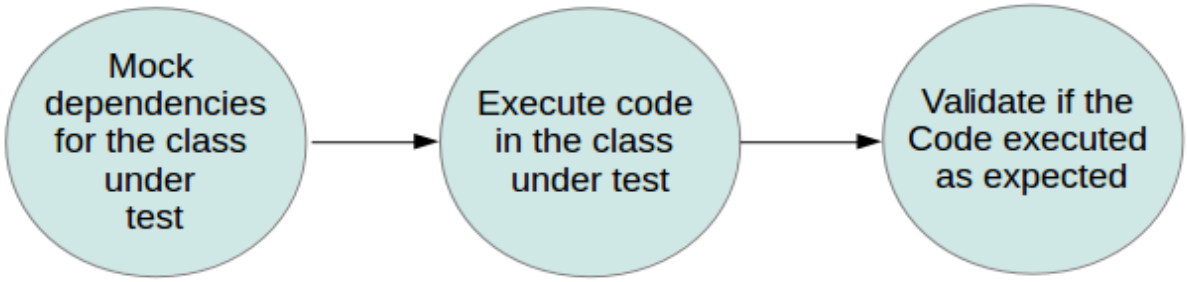


Fig:1.1 Mockito Flow Diagram

JUnit is a Java test framework and an open source project . JUnit 5 (also known as Jupiter) is the latest major release of JUnit. It consists of a number of discrete components:

- JUnit Platform - foundation layer which enables different testing frameworks to be launched on the JVM
- Junit Jupiter - is the JUnit 5 test framework which is launched by the JUnit Platform
- JUnit Vintage - legacy TestEngine which runs older tests

As the usage of JUnit 5 requires multiple libraries to be present, we typically use it with a build system like Maven or Gradle. JUnit 5 needs at least Java 8 to run.

- Rest-Assured+TestNg:

  Rest assured is java library for testing Restful Web services. It can be used to test XML & JSON based web services. It supports GET, POST, PUT, PATCH, DELETE, OPTIONS and HEAD requests and can be used to validate and verify the response of these requests. Also it can be integrated with testing frameworks like JUnit, TestNG etc.

  TestNG is a testing framework developed in the lines of JUnit and NUnit, however it introduces some new functionalities that make it more powerful and easier to use. TestNG is designed to cover all categories of tests: unit, functional, end-to-end, integration, etc.

Frontend Testing frameworks:

- Protractor-Jasmine:

Protractor is an open source end-to-end testing framework for Angular and AngularJS applications. It was built by Google on the top of WebDriver. It also serves as a replacement for the existing AngularJS E2E testing framework called “Angular Scenario Runner”.

It also works as a solution integrator that combines powerful technologies such as NodeJS, Selenium, Jasmine, WebDriver, Cucumber, Mocha etc. Along with testing of AngularJS applications, it also writes automated regression tests for normal web applications. It allows us to test our application just like a real user because it runs the test using an actual browser.

1.5 IDEs used:

- Visual StudioCode
- Spring Tool Suite

1.6 Database Used:

- MySQL WorkBench

1.7 Web page details:

- Registration page.
- Login page.
- Dashboard page(Includes search function).
- Favourites page.

2.PROJECT STRUCTURE

FRONTEND STRUCTURE

Here the portion wise structure of the current project is shown.

2.1 Login Page:

The landing site is the login page:

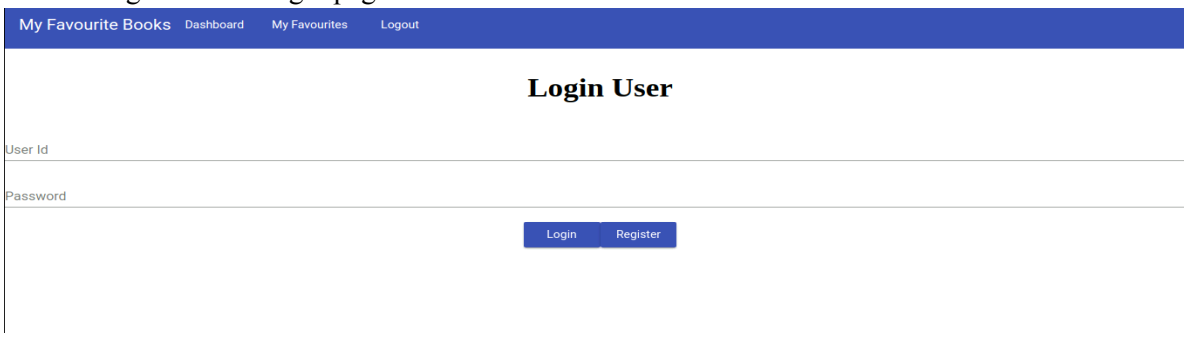


Fig 2.1 Login Page.

2.2 Registration Page:

The registration page is shown here:

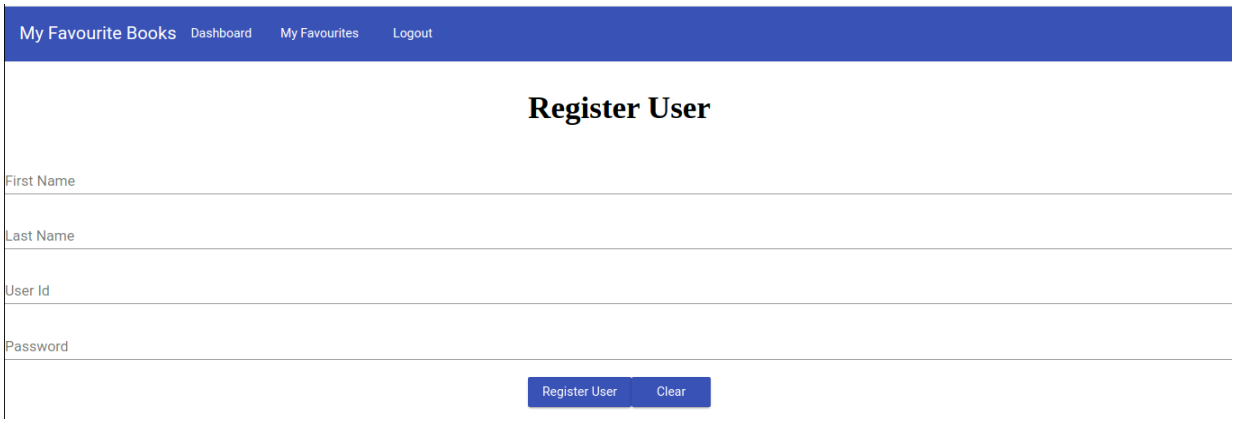


Fig 2.2 Registration Page.

2.3 Dashboard Page:

The dashboard page is shown with search functionality:

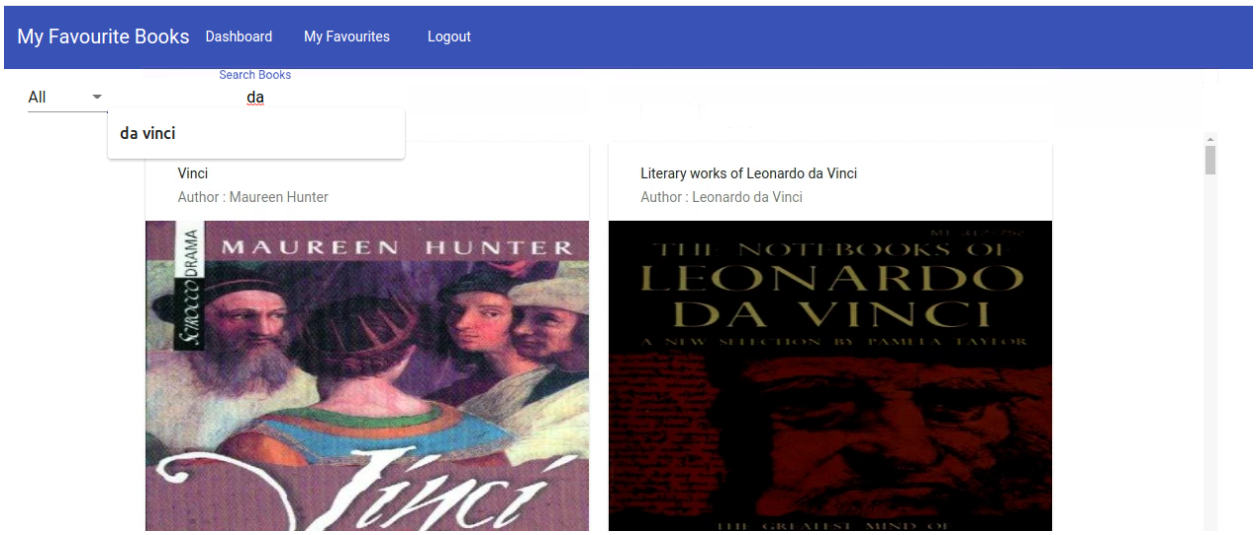


Fig 2.3 Dashboard Page.

2.4 Favourite Page:

The favourite page section is shown here:

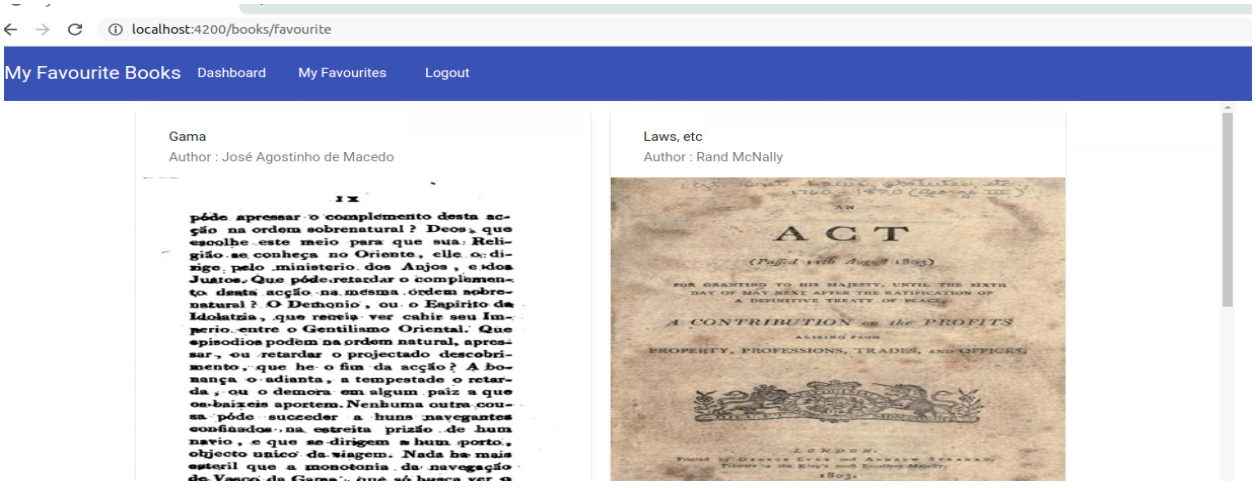


Fig 2.4 Favourite Page.

BACKEND STRUCTURE

2.5 Backend Modules:

USER SERVICE:

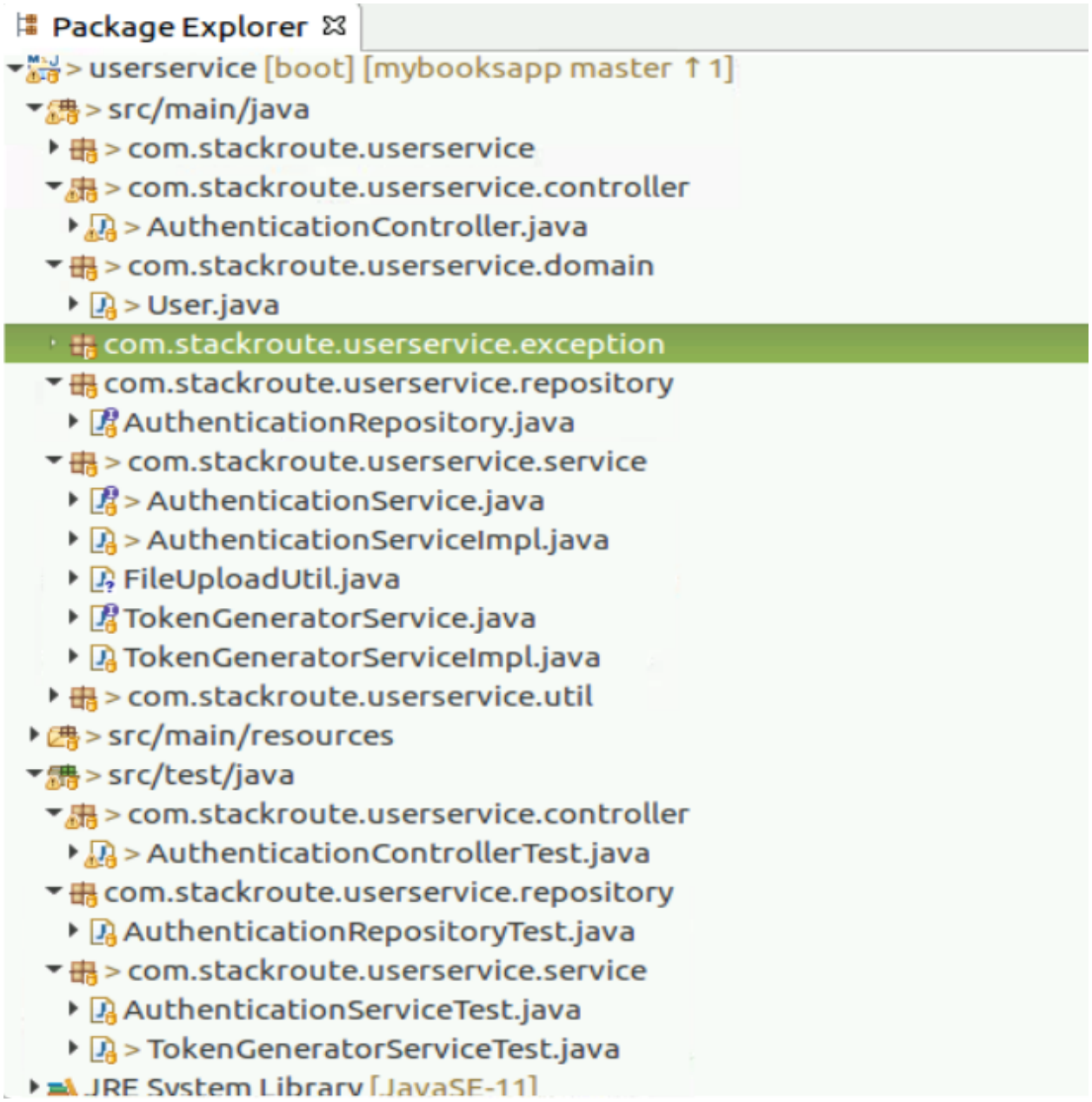


Fig 2.5 Backend-UserService Hierarchy.

FAVOURITE SERVICE:

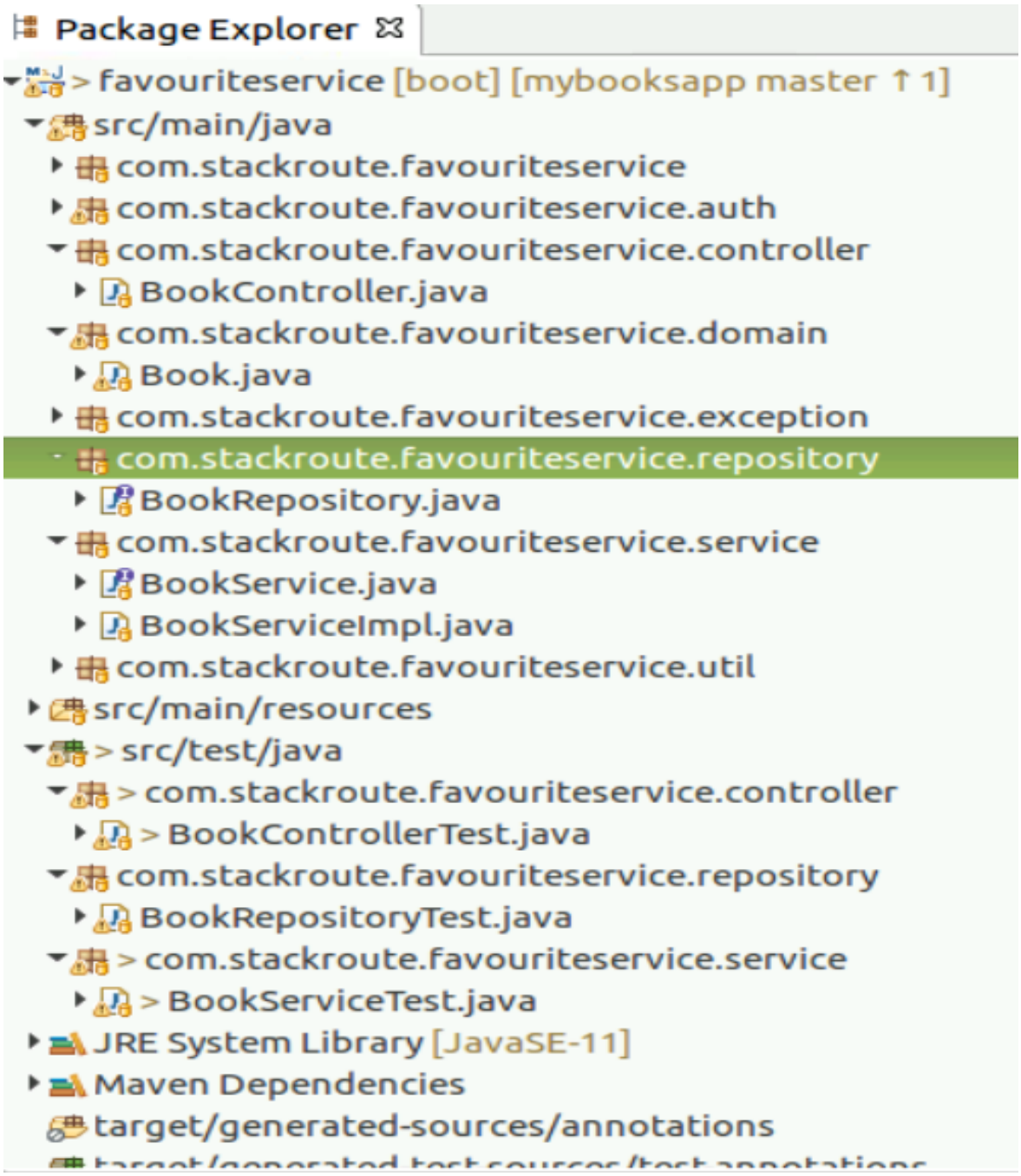


Fig 2.6 Backend-FavouriteService Hierarchy.

FRONTEND STRUCTURE

UI Hierarchy:

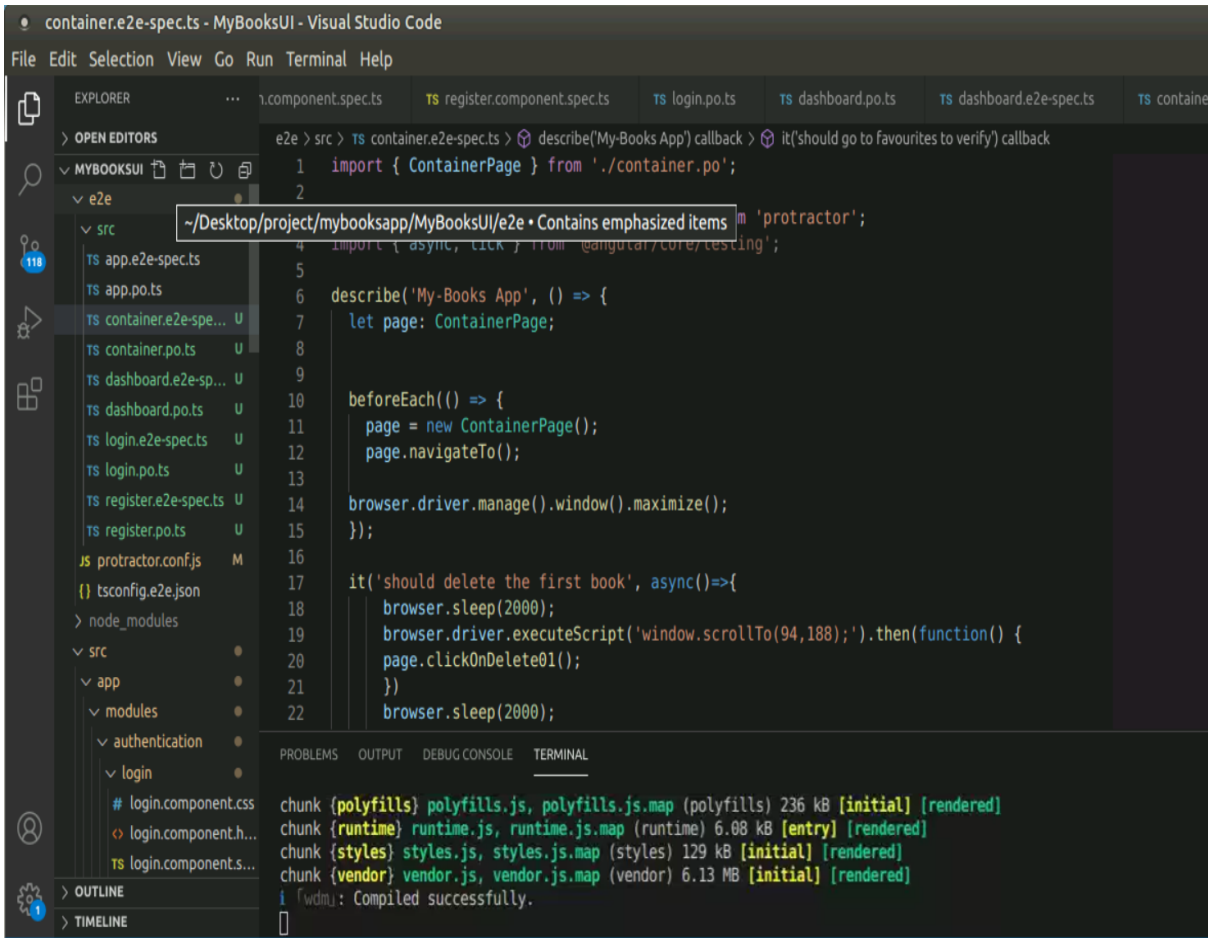


Fig:2.7 Frontend hierarchy

3.TESTING THE PROJECT

BACK-END TESTING

Here we test the different functionalities of the project. This testing is done in order to make sure that all the given functionalities are behaving and responding as per the standards and the client requirements.For the testing purpose, the technologies used are :

- Mockito+J-Unit testing- For back-end testing.
- Rest-Assured+TestNg- For back-end testing.
- Protractor+Jasmine framework- For front-end testing.

As per the hierarchy , the testing is done for the backend initially that consists of two modules.

1. UserService module.
2. FavouriteService module.

From this the first back-end testing is performed on the UserService module.

User Service Module:

UserService module consists testing portions for :

- Authentication Controller Test
- Authentication Service Test

3.1 Authentication Service Controller Testing:

The Authentication Controller part is tested using the Rest-Assured+TestNg framework. The functionalities assigned in the controller part are:

- Register user(Saving the user)
- Logging in the user.

For all the above mentioned functionalities the positive and the negative test cases are being performed. Total test cases runned and the results are also recorded.

Authentication Controller Code Snippet:

*AuthenticationControllerTest.java

```
1  package com.stackroute.userservice.controller;
2  import org.junit.runner.RunWith;
11 @RunWith(SpringRunner.class)
12 @WebMvcTest(AuthenticationController.class)
13 public class AuthenticationControllerTest {
14
15     @Test(priority=1)
16
17     public void test_registerUser() {
18
19  HashMap data = new HashMap();
20  data.put("userId", "104");
21  data.put("firstName", "lavan");
22  data.put("lastName", "pm");
23  data.put("password", "lavan@123");
24  Response res=
25  given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/registerUser")
26  .then().statusCode(201).log().body().extract().response();
27
28     }
29
30     @Test(priority=2)
31     public void test_loginUser() {
32
33          HashMap data = new HashMap();
34          data.put("userId", "104");
35
36          data.put("password", "lavan@123");
37          Response res=
38          given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/login")
```

```
37          Response res=
38          given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/login")
39          .then().statusCode(200).log().body().extract().response();
40
41     }
42     @Test(priority=3)
43
44          public void test_loginUser_with_invalid_userId() {
45      HashMap data = new HashMap();
46      data.put("userId", "logesha");
47      data.put("password", "lavan@123");
48      Response resp=
49      given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/login")
50      .then().statusCode(401).log().body().extract().response();
51      String jString=resp.asString();
52      Assert.assertEquals(jString.contains("User not found"),true);
53      //given().auth().preemptive().basic(username, password).when().get("{yourApiURL}").then().statusCode(200);
54         }
55
56         @Test(priority=4)
57
58             public void test_loginUser_With_invalid_password() {
59          HashMap data = new HashMap();
60          data.put("userId", "104");
61          data.put("password", "lb@123");
62          Response resp=
63          given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/login")
64          .then().statusCode(401).log().body().extract().response();
65          String jString=resp.asString();
```

```
        String jString=resp.asString();
        Assert.assertEquals(jString.contains("User not found"),true);
        //given().auth().preemptive().basic(username, password).when().get("{yourApiURL}").then().statusCode(200);
           }

       @Test(priority=5)

           public void test_loginUser_with_no_userid_or_password() {
        HashMap data = new HashMap();
        data.put("userId", "");
        data.put("password", "");
        Response resp=
        given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/login")
        .then().statusCode(401).log().body().extract().response();
        String jString=resp.asString();
        Assert.assertEquals(jString.contains("User Id and Password are mandatory"),true);
        //given().auth().preemptive().basic(username, password).when().get("{yourApiURL}").then().statusCode(200);
           }

           @Test(priority=6)

           public void test_registerSameUser() {

        HashMap data = new HashMap();
        data.put("userId", "104");
        data.put("firstName", "lavan");
        data.put("lastName", "pm");
        data.put("password", "lavan@123");
        Response res=
    Response res=
    given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/registerUser")
    .then().statusCode(409).log().body().extract().response();

           }


           @Test(priority=7)

               public void test_registerUser_with_no_userid_or_password() {
            HashMap data = new HashMap();
            data.put("userId", "");
            data.put("firstName", "rishi");
            data.put("lastName", "sr");
            data.put("password", "");
            Response resp=
            given().contentType("application/json").body(data).when().post("http://localhost:8081/api/books/registerUser")
            .then().statusCode(400).log().body().extract().response();
            String jString=resp.asString();
            Assert.assertEquals(jString.contains("User Id and Password are mandatory"),true);
            //given().auth().preemptive().basic(username, password).when().get("{yourApiURL}").then().statusCode(200);
               }

}
```

Fig:3.1 Code Snippets for the test cases

3.1.1 Register User Test Case:

Here the user should be able to register themself in the account. Register user contains four fields that are to be filled by the user. For the testing purpose dummy values are given and the save method is initiated.The content types for which the dummy values are given are in Json format. The path variable for initiating the save method is called after "when()." as a "post" method.

After "then()." a status code of **201** is expected as the data move to the database. If datas moves to the database,the expected and computed status code will match and pass the test case. The test cases result is recorded.

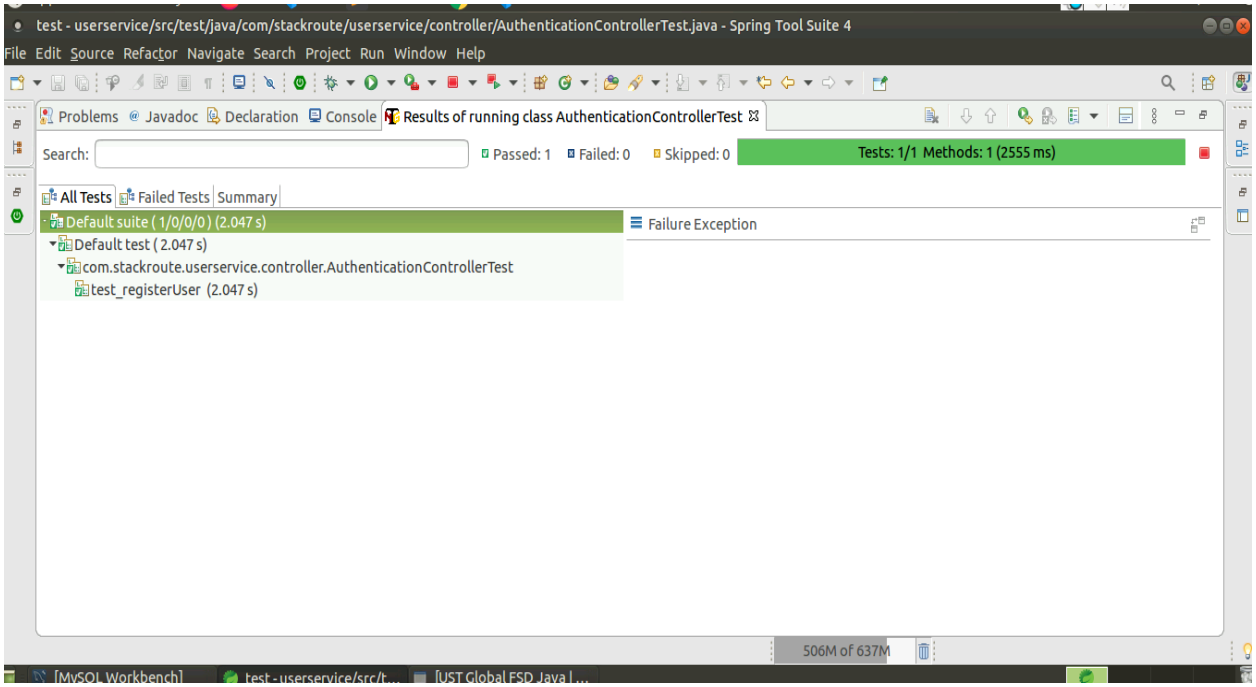


Fig 3.2 Register user result

.

3.1.2 Login_User Test Case with Valid Credentials:

Here the user should be able to login with valid user id and password.Dummy user id and password are given for testing purposes under the test_loginUser() function, and tested using test ng . If a user is able to move to the dashboard page, the expected and the computed status code will match and the test case will be passed.

Result: Expected status code matches the computed status code, hence passing the test case.

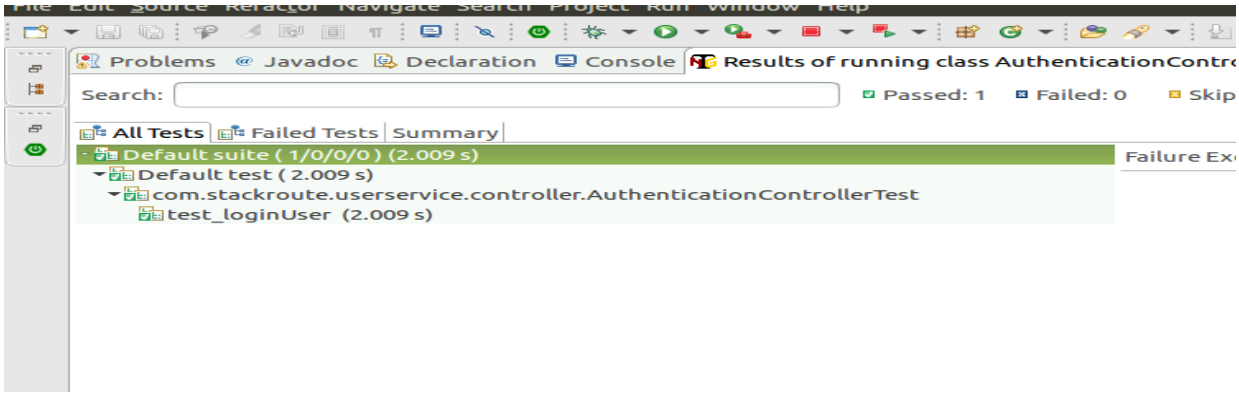


Fig 3.3 Login user result.

3.1.3. Invalid Login with Invalid UserId:

Here the user should not be able to login with invalid user id and valid password.Invalid user id is given for testing purposes under the test_loginUser_With_invalid_userID() function, and tested using TestNg . If a user is not able to move to the dashboard page, the expected and the computed status code will match and the test case will be passed.

Result: Expected status code **401** matches the computed status code, hence passing the test case.

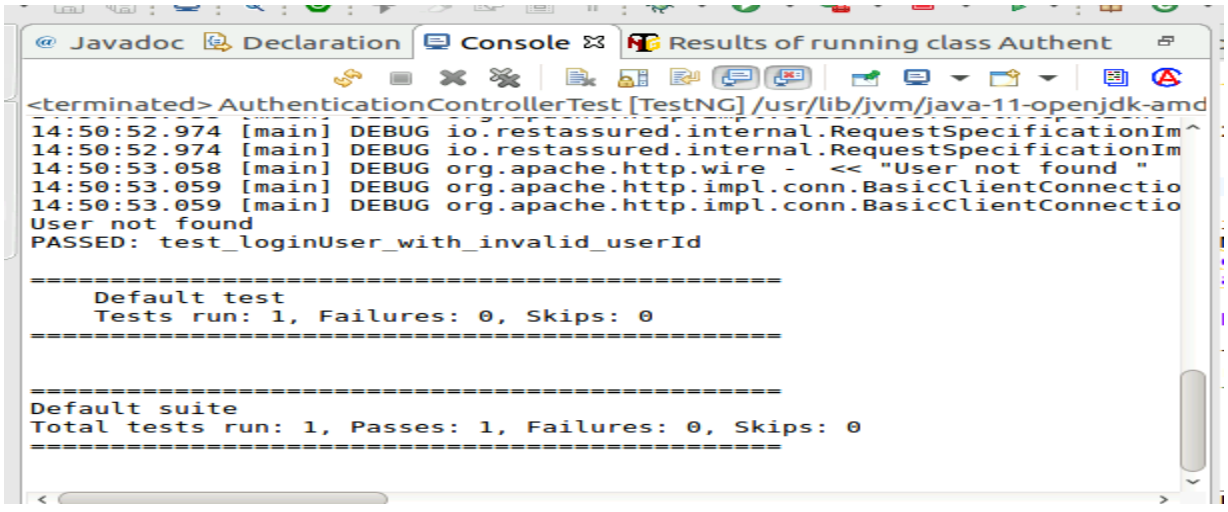


Fig 3.4 Login with invalid user Id

3.1.4. Invalid Login with Invalid Password:

Here the user should not be able to login with valid user id and invalid password.Invalid password is given for testing purposes under the test_loginUser_With_invalid_password() function, and tested using TestNg . If a user is not able to move to the dashboard page, the expected and the computed status code will match and the test case will be passed.

Result: Expected status code **401** matches the computed status code, hence passing the test case.

```
PASSED: test_loginUser_With_invalid_password

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig 3.5 Login with invalid password.

3.1.5. Invalid Login with No User Id or Password:

Here, users should not be able to login with any valid user id or password.UserId and password is given as null for testing purposes under the test_loginUser_With_invalid_password() function, and tested using TestNg . If a user is not able to move to the dashboard page, the expected and the computed status code will match and the test case will be passed.

Result: Expected status code **401** matches the computed status code, hence passing the test case.

```
PASSED: test_loginUser_with_no_userid_or_password

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig 3.6 Login with no user Id and password

3.1.6. Already Existing Register User Test Case:

Here the user should not be able to register themself in the account if it is an already registered or existing account. Register user contains four fields that are to be filled by the user. For the testing purpose dummy values are given and the save method is initiated.The content types for which the dummy values are given are in Json format. The path variable for initiating the save method is called after "when()." as a "post" method.

After "then()." a status code of **409** is expected as the data does not move to the database. If datas doesn't move to the database,the expected and computed status code will match and pass the test case. The test cases result is recorded.

Result: Expected status code matches the computed status code, hence passing the test case.

```
PASSED: test_registerSameUser

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig 3.7 Register already existing users.

3.1.7. Register User Test Case With no UserId or password:

Here the user should not be able to register themself in the account if no userId or password is entered. Register user contains four fields that are to be filled by the user. For the testing purpose dummy values with null user id and password are given and the save method is initiated.The content types for which the dummy values are given are in Json format. The path variable for initiating the save method is called after "when()." as a "post" method.

After "then()." a status code of **400** is expected as the data does not move to the database. If datas doesn't move to the database,the expected and computed status code will match and pass the test case. The test cases result is recorded.

Result: Expected status code matches the computed status code, hence passing the test case.

```
PASSED: test_registerUser_with_no_userid_or_password

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig 3.8 Register with no user Id and password.

3.2 Authentication Service-service Testing:

For the userService-service class testing, Springboot,Mockito,Junit frameworks help in automating the testing portion.

Testing the functionality of UserService .

1. Registering User or Saving User
2. Getting Registered userdetails

```
23 @RunWith(SpringRunner.class)
24 public class AuthenticationServiceTest {
25     @Mock
26     private AuthenticationRepository AR;
27     @InjectMocks
28     private AuthenticationServiceImpl service;
29     User user, user1;
30     Optional<User> optUser;
31
32     @Before
33     public void setUp() throws Exception {
34         user = new User();
35         user.setUserId("123");
36         user.setPassword("group2");
37         user.setFirstName("bookApp");
38         user.setLastName("bookappGroup2");
39         optUser = Optional.of(user);
```

Fig:3.9 Mock values and injection.

3.2.1.Save User Test Success

In this test case the save user functionality is verified by mocking the repository of the service interface and injecting mocks to the service implementation class method "saveUser" using the Mockito framework.Here the creation of user object and passing the object to saveUser method is made.The expected result is to save the specific user if the details don't prevail in the database.For the confirmation the expected message is "`User successfully registered to the App`".

The expected and the actual result is verified using assertion. The same findings are recorded here.Each values are mocked as in the data type mentioned in the pojo class. Assertion is carried to verify whether the particular service is implemented successfully.

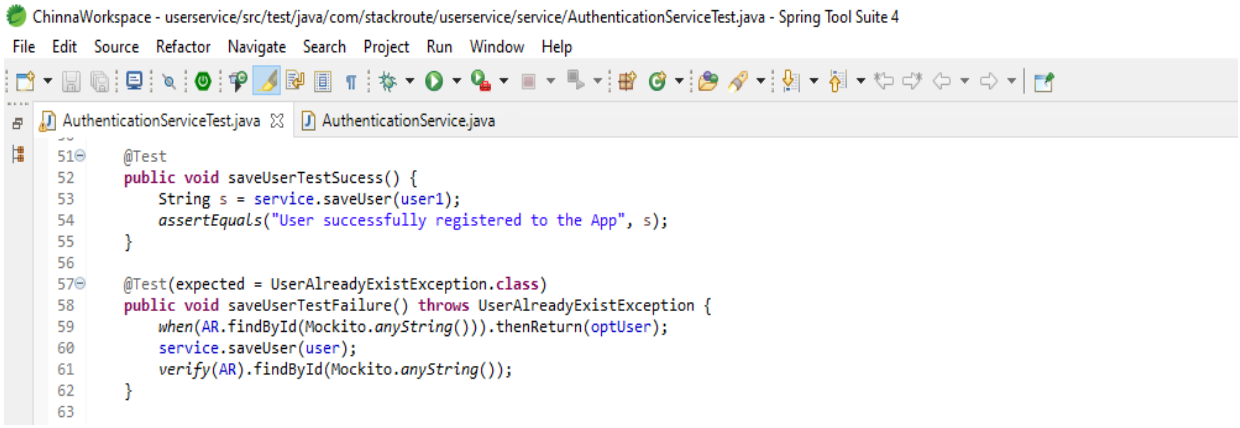


Fig:3.10 Save user test case.

3.2.2.Save User Test Failure:

In this test case the same user is tried to be saved twice by mocking the repository of service interface and injecting mocks to the service implementation class with method saveUser using Mockito framework. The expected result should be equal to "UserAlreadyExistException.**class**" when the call for saveUser method is invoked. Thus the same data referring to the described datatype is given for this process.The expected and computes results are recorded here:

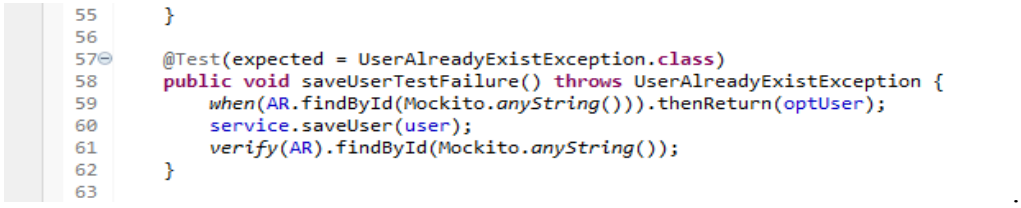


Fig: 3.11 Save user test failure.

3.2.3.Get User Details Success(Passing valid userId ,Password as parameters):

In this test case the getUserDetails functionality is verified by mocking the repository of service interface and injecting mocks to the service implementation class method "GetUserDetails" using Mockito framework

Here creating the valid user is done and calling the "GetUserDetails" method by passing the parameters valid "userId" , "password" expecting to return the user data.

The expected and the actual result is verified using assertion. The same findings are recorded here.

```
63
64    @Test
65    public void getUserDetailsTest() throws UserAlreadyExistException {
66        when(AR.findByUserIdAndPassword(Mockito.anyString(), Mockito.anyString())).thenReturn(user);
67        User u = service.getUserDetails("123", "group2");
68        assertEquals("bookApp", u.getFirstName());
69    }
70
```

Fig:3.12 Get user details success.

3.2.4.Get User Details Failure 1(Passing the null parameters):

In this test case the getUserDetails functionality is verified by mocking the repository of service interface and injecting mocks to the service implementation class method "GetUserDetails" using Mockito frameworkCreating the specific user and calling the "GetUserDetails" method by passing the both parameters as "Null" expecting to return the `UserNotAvailableException.class` is done here. For the invalid data the user shall not login, thus the expected error if matches the computed error passes the test case.

The same findings are recorded here.

```
70
71    @Test(expected = UserNotAvailableException.class)
72    public void getUserDetailsFailure() {
73        User s = service.getUserDetails("null", "null");
74        assertEquals(UserNotAvailableException.class, s.getFirstName());
75    }
76
```

Fig:3.13 Get user details failure.

3.2.5.Get User Details Failure 2(Passing the "invalid userId","Valid password" as parameters):

In this test case the getUserDetails functionality is verified by mocking the repository of service interface and injecting mocks to the service implementation class method "GetUserDetails" using Mockito frameworkBy creating the valid user object and calling the "GetUserDetails" method by passing the invalid parameters i.e invalid userId" and valid "password" it is expecting to return the

`UserNotAvailableException.class` . For the invalid data the user shall not login, thus the expected error if matches the computed error passes the test case.

The exceptions occurred are captured at the specific moment. With the test case expecting an exception regarding no presence of the user is validated for being true or false.

The same findings are recorded here.

```
76
77    @Test(expected = UserNotAvailableException.class)
78    public void getUserDetailsFailure1() {
79        when(AR.findByUserIdAndPassword("123", "group2")).thenReturn(user);
80        User s = service.getUserDetails("1234", "passw");
81        assertEquals(user.getFirstName(), s.getFirstName());
82
83    }
```

Fig:3.14 Get user details failure 02

3.2.6.Get User Details Failure 3(Passing “valid userId” and “null” as parameters):

In this test case the getUserDetails functionality is verified by mocking the repository of service interface and injecting mocks to the service implementation class method “GetUserDetails” using Mockito framework

By creating the valid user object and calling the “GetUserDetails” method by passing the parameter valid “userId”, “null” for password it is expecting to return the `UserNotAvailableException.class` . For the invalid data the user shall not login, thus the expected error if matches the computed error passes the test case.

The exceptions occurred are captured at the specific moment. With the test case expecting an exception regarding no presence of the user is validated for being true or false.

The same findings are recorded here.

```
84
85    @Test(expected = UserNotAvailableException.class)
86    public void getUserDetailsFailure2() {
87        when(AR.findByUserIdAndPassword("123", "group2")).thenReturn(user);
88        User s = service.getUserDetails("123", "null");
89        assertEquals(user.getFirstName(), s.getFirstName());
90
91    }
92
```

Fig:3.15 Get user details failure 03

3.2.7.Get User Details Failure 4(Passing “null” and “valid password” as parameters):

In this test case the getUserDetails functionality is verified by mocking the repository of service interface and injecting mocks to the service implementation class method “GetUserDetails” using Mockito frameworkBy creating the valid user object and calling the “GetUserDetails” method by passing the parameter “null” for user id and a valid ,”password” it is expecting to return the `UserNotAvailableException.class` .

For the invalid data the user shall not login, thus the expected error if matches the computed error passes the test case.The exceptions occurred are captured at the specific moment. With the test case expecting an exception regarding no presence of the user is validated for being true or false.

The same findings are recorded here.

```
92
93    @Test(expected = UserNotAvailableException.class)
94    public void getUserDetailsNullTest3() {
95
96        when(AR.findByUserIdAndPassword("123", "group2")).thenReturn(user);
97        User s = service.getUserDetails("null", "group2");
98        assertEquals(user.getFirstName(), s.getFirstName());
99
100   }
101
```

Fig:3.16 Get user details failure 04.

Overall User Service-Services Test Case Result:

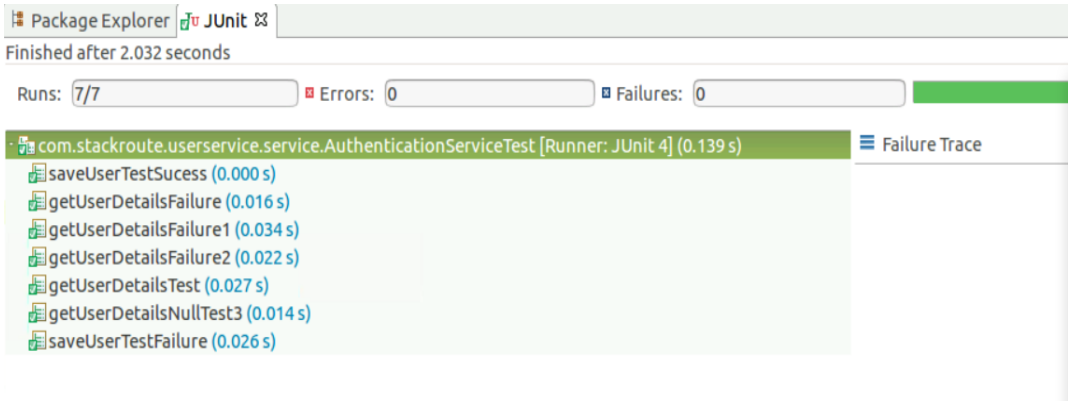


Fig: 3.17 UserService -Service test case result.

Favourite Service Module: FavouriterService module consists testing portions for :

- FavouriteService Controller Test
- FavouriteService Service Test

3.3 Favourite Service Controller Testing:

The FavouriteService Controller part is tested using the Rest-Assured+TestNg framework. The functionalities assigned in the controller part are:

- Saving the book to favourites.
- Listing the saved books.
- Deletion of books from favourite.

For all the above mentioned functionalities the positive and the negative test cases are being performed.

Favourite Controller Code Snippet:

```
@Test(priority=1)
public void saveBookTest() {
    String token = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJuYXNoIiwiaWF0IjoxNjEzODE0NDQzfQ.cqm8yMhgRlK8Jj2rhx4ZJGZllByh601156A1f-q8ov4";


    HashMap data = new HashMap();
    data.put("id", 119);
    data.put("userId", "fredy");
    data.put("coverImage", "pic.jpg");
    data.put("title", "mello");
    data.put("authorName", "Gerrard");
    data.put("firstPublisYear",1896);
    data.put("editionCount", 21);

     Response res=

            given().header("Authorization", "Bearer " + token).contentType("application/json").with().body(data).when().
            post("http://localhost:8082/api/books/saveBookToMyFavourites")
            .then().statusCode(201).log().body().extract().response();
}

 //Book already added test case(Negative Test Case.
 @Test(priority = 2)
 public void test_BookAlreadyExists() {

     String token = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJuYXNoIiwiaWF0IjoxNjEzODE0NDQzfQ.cqm8yMhgRlK8Jj2rhx4ZJGZllByh601156A1f-q8ov4";

     HashMap data = new HashMap();
        data.put("id", 330);
        data.put("userId", "Hanua");
        data.put("coverImage", "Hanu");
        data.put("title", "Hanua Mejo");
        data.put("authorName", "Amber");
        data.put("firstPublisYear",1559);
        data.put("editionCount", 13);
```

```
        Response res=

                given().header("Authorization", "Bearer " + token).contentType("application/json").with().body(data).when().
                post("http://localhost:8082/api/books/saveBookToMyFavourites")
                .then().statusCode(409).log().body().extract().response();
}

 //Get the list of all favourite books.
@Test(priority = 3)
       public void test_getMyFavouriteBooks() {
        String token = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJuYXNoIiwiaWF0IjoxNjEzODE0NDQzfQ.cqm8yMhgRlK8Jj2rhx4ZJGZllByh601156A1f-q8ov4";

    given().header("Authorization", "Bearer " + token).contentType("application/json").when().get("http://localhost:8082/api/books/getMyFavouriteBooks")
    .then().statusCode(200);
       }

//Delete books from favourites.
 @Test(priority =4)
    public void test_deleteBooksFromFavourite() {
     String token = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJuYXNoIiwiaWF0IjoxNjEzODE0NDQzfQ.cqm8yMhgRlK8Jj2rhx4ZJGZllByh601156A1f-q8ov4";

 Response res=
        given().header("Authorization", "Bearer " + token).contentType("application/json").when().
        delete("http://localhost:8082/api/books/deleteBookFromMyFavourites/119")
        .then().statusCode(200).log().body().extract().response();


    }



    }
 //Testing with  ID
 @Test(priority =5)
    public void test_deleteBooksFailure() {
     String token = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJuYXNoIiwiaWF0IjoxNjEzODE0NDQzfQ.cqm8yMhgRlK8Jj2rhx4ZJGZllByh601156A1f-q8ov4";

 Response res=
        given().header("Authorization", "Bearer " + token).contentType("application/json").when().
        delete("http://localhost:8082/api/books/deleteBookFromMyFavourites/54")
        .then().statusCode(409).log().body().extract().response();


    }
 //Get my books failure
    @Test(priority = 6)
    public void test_getMyFavouriteBooksFailure() {
     String token = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJuYXNoIiwiaWF0IjoxNjEzODE0NDQzfQ.cqm8yMhgRlK8Jj2rhx4ZJGZllByh601156A1f-q8ov4";

given().header("Authorization", "Bearer " + token).contentType("application/json").when().get("http://localhost:8082/api/books/getMyFavouriteBooks/662")
.then().statusCode(404);
    }
```

Fig 3.18: Favourite Controller Code Snippet

3.3.1 Save Books to Favourite Test Case:

Here the test case for saving the books to favourite is performed. For the event to occur the user identity(user token) has to be given. The JWT filter is responsible for the token generation for each unique user. Providing the appropriate token after the login of users instantiates the process of saving the books. The tokens are a unique string of ID that is unique every time a user logs in.This token is being passed over each test case to make sure that a particular user has logged in the session.Thus the identity

verification is complete.

In saving the books, dummy details of the book are provided in Json type as body towards the test case. If the provided dummy book details reflect the database a status code **201** is instantiated.After the saving the expected and computed status are verified thus passing the test case. The test cases are being recorded.

```
15:19:40.995 [main] DEBUG org.apache.http.impl.conn.Ba
Book successfully added to your favourites
PASSED: saveBookTest

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig:3.19 Save book success test case.

3.3.2. Save Books to Favourite Failure Test Case:

Here the test case for saving the books to favourite is performed. For the event to occur the user identity(user token) has to be given. The JWT filter is responsible for the token generation for each unique user. Providing the appropriate token after the login of users instantiates the process of saving the books. The tokens are a unique string of ID that is unique every time a user logs in.

This token is being passed over each test case to make sure that a particular user has logged in the session.Thus the identity verification is complete.When trying to save an already existing book must pose an error or conflict. This is given as a status code of **409** which is a conflict status code. The expected status code and the computed code is verified thus passing the test case where the book saving fails. The test cases are recorded.

```
Book already available in your Favourites
PASSED: test_BookAlreadyExists

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig:3.20 Save book failure test case.

3.3.3. Listing of Books in Favourite Test Case:

Here the test case for listing the books to favourite is performed. For the event to occur the user identity(user token) has to be given. The JWT filter is responsible for the token generation for each unique user.

Providing the appropriate token after the login of users instantiates the process of listing the books. The tokens are a unique string of ID that is unique every time a user logs in.This token is being passed over each test case to make sure that a particular user has logged in the session.Thus the identity verification is

complete.When trying to list the books no new Json body has to be sent as dummy figures. Instead all existing books are listed as the "getMyBookFromFavourites" path variable is provided.When the listing is successful an status code of **200** is generated. The expected status code and the computed status code is verified thus passing the test case.The test cases are recorded.

```
PASSED: test_getMyFavouriteBooks

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig:3.21 Get book success test case.

3.3.4. Delete Books from Favourite Test Case:

Here the test case for deleting the books to favourite is performed. For the event to occur the user identity(user token) has to be given. The JWT filter is responsible for the token generation for each unique user. Providing the appropriate token after the login of users instantiates the process of deleting the books. The tokens are a unique string of ID that is unique every time a user logs in.This token is being passed over each test case to make sure that a particular user has logged in the session.Thus the identity verification is complete.

Deletion process is carried on with the help of the auto generated ID created during the saving of books to favourite.Thus the path variable is called for the delete process which instantiates the delete method in the controller. No Json body is to be given here.After deletion is successful a status code of **200** is generated. The expected and computed status code are verified thus passing the test case.

```
Book successfully deleted from your favourites
PASSED: test_deleteBooksFromFavourite

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig:3.22 Book deletion success test case.

3.3.5. Delete Books from Favourite Failure Test Case:

Here the test case for deleting the books to favourite is performed. For the event to occur the user identity(user token) has to be given. The JWT filter is responsible for the token generation for each unique user.

Providing the appropriate token after the login of users instantiates the process of deleting the books. The tokens are a unique string of ID that is unique every time a user logs in.This token is being passed over each test case to make sure that a particular user has logged in the session.Thus the identity verification is complete.

Deletion process is carried on by detecting the auto generated value during the book saving process. But a book which is not saved is out of scope for the delete process. The path variable when instantiated searches for the specific ID in the specific user account.If no book for deletion is found a status code of **409** is generated . The expected and computed status are verified thus passing the test case for failure of book deletion.

```
Book not found in your Favourites
PASSED: test_deleteBooksFailure

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig:3.23 Book deletion failure test case.

3.3.6. Listing Books from Favourite Failure Test Case:

Here the test case for listing the books to favourite is performed. For the event to occur the user identity(user token) has to be given. The JWT filter is responsible for the token generation for each unique user.

Providing the appropriate token after the login of users instantiates the process of listing the books. The tokens are a unique string of ID that is unique every time a user logs in.This token is being passed over each test case to make sure that a particular user has logged in the session.Thus the identity verification is complete.When the variable path for the listing is instantiated ,here it refers to the specific ID of the book. If the ID of the book is not present it should return a status code of **404** .Thus if not found the expected status code and computed status code is verified thus passing the test case.

```
PASSED: test_getMyFavouriteBooksFailure

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================
```

Fig:3.24 Get book failure test case.

3.4 Favourite Service Services Testing

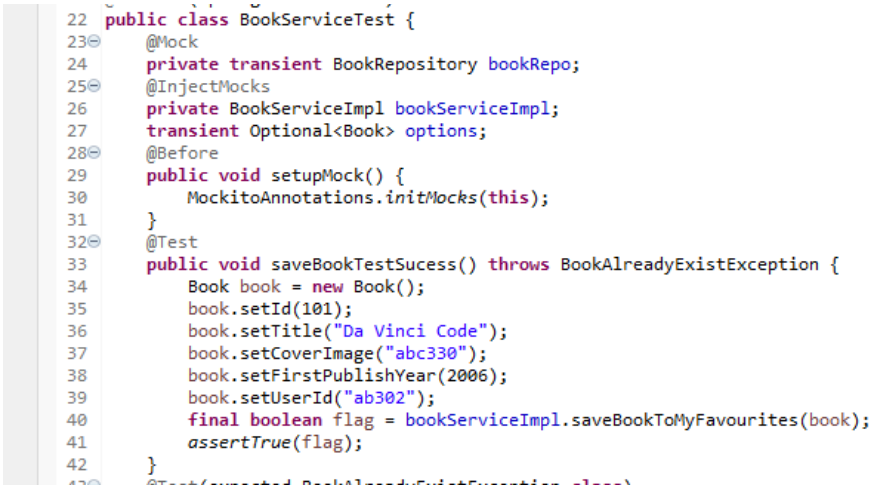
Testing the functionality of FavouriteService are: .

1. Saving Book in the favourites
2. Get the favourite book
3. Deleting Book from the favourites list.

3.4.1.Save Book Success Test Case:

Here the functionality of saving the book to the favourites is carried on. Using **Mockito** the mock values consisting of book details are injected to the favourite service implementation class. The save method in the repository is instantiated . The expected boolean result is set to true if the **save** method works as per the requirement.

Asserting the boolean result with the computed result will pass the test case if the verification is a success.

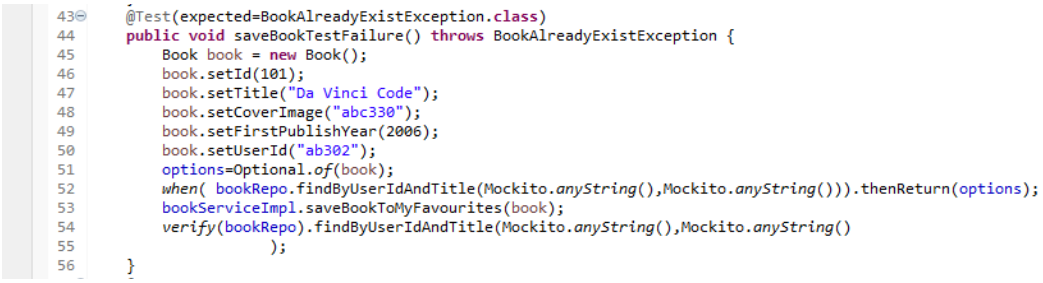


```
22 public class BookServiceTest {
23     @Mock
24     private transient BookRepository bookRepo;
25     @InjectMocks
26     private BookServiceImpl bookServiceImpl;
27     transient Optional<Book> options;
28     @Before
29     public void setupMock() {
30         MockitoAnnotations.initMocks(this);
31     }
32     @Test
33     public void saveBookTestSucess() throws BookAlreadyExistException {
34         Book book = new Book();
35         book.setId(101);
36         book.setTitle("Da Vinci Code");
37         book.setCoverImage("abc330");
38         book.setFirstPublishYear(2006);
39         book.setUserId("ab302");
40         final boolean flag = bookServiceImpl.saveBookToMyFavourites(book);
41         assertTrue(flag);
42     }
```

Fig: 3.25 Save book success.

3.4.2.Save Book Failure Test Case:

Here the functionality of saving the book to the favourites is carried on. Using **Mockito** the mock values consisting of book details are injected to the favourite service implementation class. The **save** method in the repository is instantiated . The expected boolean result is set to true if the save method works as per the requirement.When the same book is added then an error must occur with exception of **"BookAlreadyExistException"**.The test case expects the exception for adding the same book. To make the test more versatile the given book is verified if present in the repository. If the expected result matches the computed result the test case gets passed.



```
43     @Test(expected=BookAlreadyExistException.class)
44     public void saveBookTestFailure() throws BookAlreadyExistException {
45         Book book = new Book();
46         book.setId(101);
47         book.setTitle("Da Vinci Code");
48         book.setCoverImage("abc330");
49         book.setFirstPublishYear(2006);
50         book.setUserId("ab302");
51         options=Optional.of(book);
52         when( bookRepo.findByUserIdAndTitle(Mockito.anyString(),Mockito.anyString())).thenReturn(options);
53         bookServiceImpl.saveBookToMyFavourites(book);
54         verify(bookRepo).findByUserIdAndTitle(Mockito.anyString(),Mockito.anyString()
55                 );
56     }
```

Fig:3.26 Save book failure.

3.4.3.Get All Books Success:

Here the books that have been saved are to be listed. For this the **"getMyFavouriteBooks()"** is called. This invokes the repository and lists the books that have been saved in the repository. Thus the expected result of the book is verified by the title and the name. If the computed result matches the expected result, the test case gets passed.

AuthenticationServiceTest.java | AuthenticationService.java | BookServiceTest.java | BookSe

```
56      }
57      @Test
58      public void getAllBookTest() {
59          final List<Book> bookList = new ArrayList<>();
60          Book book = new Book();
61          book.setId(1);
62          book.setTitle("venkey");
63          book.setCoverImage("abc33");
64          book.setFirstPublishYear(2001);
65          book.setUserId("ab30");
66          bookList.add( book );
67          when(bookRepo.findByUserId("ab30")).thenReturn(bookList);
68          List<Book> fetchedBook = bookServiceImpl.getMyFavouriteBooks("ab30");
69          assertEquals(bookList,fetchedBook);
70
71      }
72
```

Fig:3.27 Get all books success.

3.4.4.Delete Book by Id Success:

Here deletion of specific books by the Id is carried on. The mock values of a book with specific Id is passed. The specific book is initially tracked by the Id and then **"deleteBookFromMyFavourites()"** is initiated and the method gets invoked. The service implementation for the delete process is given to the boolean function which in return if the deletion is successful passes the **boolean flag as true**. If the boolean flag turns true, the test case passes for the delete functionality.

```
73      @Test
74      public void deleteBookByIdSucessTest() throws BookNotFoundException {
75          Book book = new Book();
76          book.setId(101);
77          book.setUserId("abcd");
78          book.setCoverImage("bookImage.jpg");
79          book.setTitle("Harry Potter");
80          book.setFirstPublishYear(2004);
81          book.setEditionCount(5);
82          options=Optional.of(book);
83          when(bookRepo.findById(101)).thenReturn(options);
84          final boolean flag = bookServiceImpl.deleteBookFromMyFavourites(101);
85          assertTrue(flag);
86      }
```

Fig:3.28 Delete book success.

3.4.5.Delete Book by Id Failure:

Here the deletion is carried on a non-existing book. Thus for this test case it expects a **"BookNotFoundException**". A mock value of book details are given. A specific book is tracked by its Id. Then the **"deleteBookFromMyFavourites()**" method is called which invokes the delete function.

But here,the deletion is carried for a non-existing Id, thus the boolean flag assigned for the service implementation must turn false. If the expected flag status matches the computed flag status, the test case gets passed.

```
88    @Test(expected=BookNotFoundException.class)
89    public void deleteBookByIdTestException() throws BookNotFoundException {
90
91        Book book = new Book();
92        book.setId(101);
93        book.setUserId("abcd");
94        book.setCoverImage("bookImage.jpg");
95        book.setTitle("Harry Potter");
96        book.setFirstPublishYear(2004);
97        book.setEditionCount(5);
98        options=Optional.of(book);
99        when(bookRepo.findById(101)).thenReturn(options);
100        bookServiceImpl.deleteBookFromMyFavourites(102);
101       }
102   }
```

Fig:3.29 Book delete failure.

Overall Test Result:

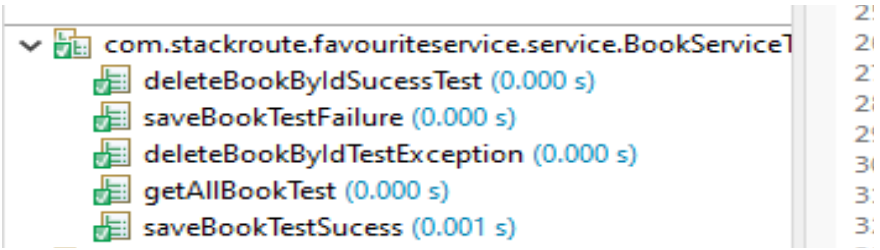


Fig:3.30 Overall test result.

FRONT-END TESTING

For the front-end testing we use Protractor+Jasmine framework which helps in automating the front-end portion. For the testing purpose there are four fields present in the front-end:

1. Login page/Landing site page.
2. Register page(including the resetting of data).
3. Dashboard Page including search functionality.
4. Container/Favourite page including deletion functionality.

Every testing related to the frontend is performed in Visual Studio Code and run using "protractor.protractor.conf.js" command in e2e terminal.

3.5 Registration Page:

In registration page an user should be able to register his/her details in the register user page.The register user page consists of 4 fields such as firstName, lastName, userId, password.a new user should enter firstName, lastName, and a new userId and password which is not already registered.

Once the user enters the details and clicks the register button the values are stored in the database and a pop up message is shown such as "registration is successful" and it will be redirected to the login page.

3.5.1 Register New User Success:

Here the user must be able to register with the correct register details. In return to the successful register, the user must be directed to the login url, and a popup message, "Registration Successful" is shown.

Thus the expected url is that of the login. The expected url and the computed url will be verified. If it matches it will pass the test case. The same findings are being recorded here.

Here the computed url is login page which is the expected url, hence passing the test case.

```
it('should register new user', () => {
    page.clickOnRegisterButton();
    page.setFirstname("Amrutha");
    page.setLastname("M");
    page.setuserid("amrutha");
page.setPassword("amrutha123");
page.clickOnRegisterUserButton();
browser.sleep(3000);
let val = element(by.xpath("/html/body/div[1]")).getText();
browser.sleep(2000);
expect(val).toEqual("Registration successful");

});
```

Fig:3.31 Code for register new user

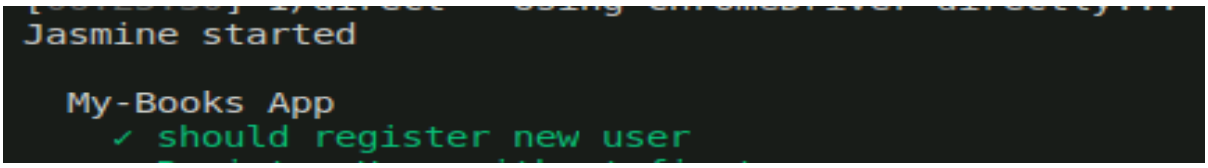


Fig:3.32 Register new user result

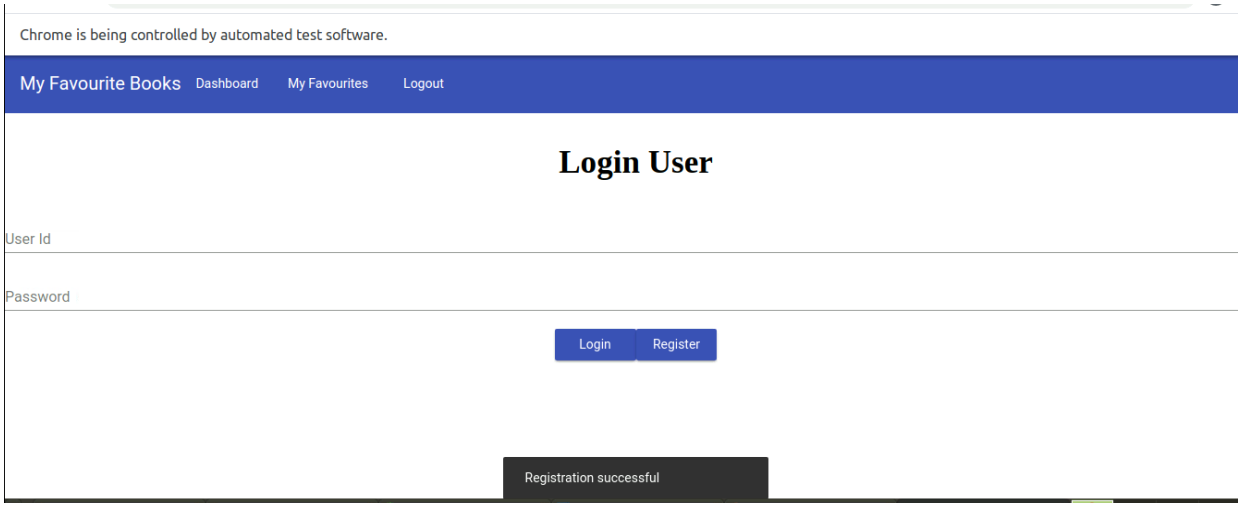


Fig: 3.33 Screenshot of register new user popup message.

3.5.2 Register User without first name:

Here Register credentials are given without a first name. This test case is done by only passing the last name, user Id and password The first name field is left null. In reality it should pose some error with a specific pop up message. The expected message is “First name is mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

Here the computed message passes the expected message hence passing the test case.

```
it('Register User without first name', () => {
  page.clickOnRegisterButton();
  page.setLastname("Patel");
  page.setuserid("kish123");
  page.setPassword("jkush");
  page.clickOnRegisterUserButton();
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(3000);
  expect(val).toEqual("First name is mandatory");
  //page.setPassword("password");
});
```

Fig:3.34 Code for register user without first name.

Fig:3.35 Register User without first name

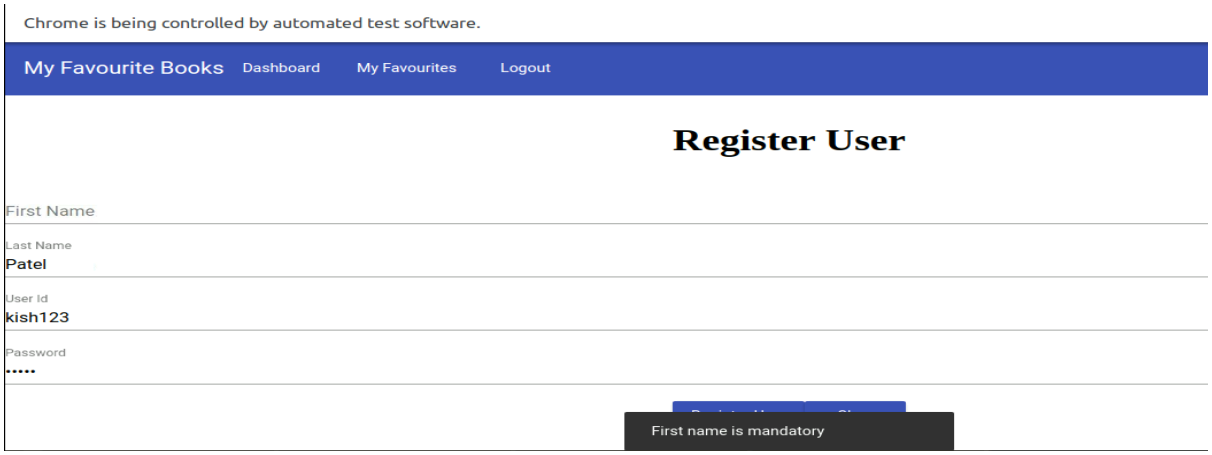


Fig:3.36 Screenshot of register user without first name popup message.

3.5.3 Register User without last name:

Here Register credentials are given without a last name. This test case is done by only passing the first name, user Id and password The last name field is left null. In reality it should pose some error with a specific pop up message. The expected message is “Last name is mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

Here the computed message passes the expected message hence passing the test case.

```
it('Register User without last name', () => {
  page.clickOnRegisterButton();
  page.setFirstname("Kishan");
  page.setuserid("kish123");
  page.setPassword("jkush");
  page.clickOnRegisterUserButton();
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(3000);
  expect(val).toEqual("Last name is mandatory");
  //page.setPassword("password");
});
```

Fig:3.37 Code for register user without last name.

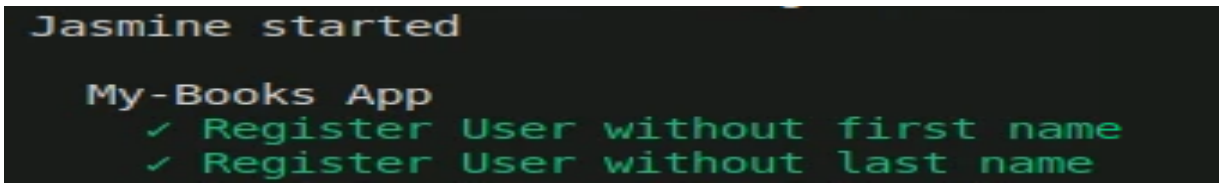


Fig:3.38 Register User without first name result.

3.5.4 Register User without User Id:

Here Register credentials are given without an User Id. This test case is done by only passing the first name, last name and password The User Id field is left null. In reality it should pose some error with a specific pop up message.

The expected message is “User Id and password are mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

Here the computed message passes the expected message hence passing the test case.

```
it('Register User without UserId', () => {
  page.clickOnRegisterButton();
  page.setFirstname("Kishan");
  page.setLastname("Patel");
  page.setPassword("jkush");
  page.clickOnRegisterUserButton();
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(3000);
  expect(val).toEqual("User Id and Password are mandatory");
  //page.setPassword("password");
});
```

Fig:3.39 Code for register user without User Id.

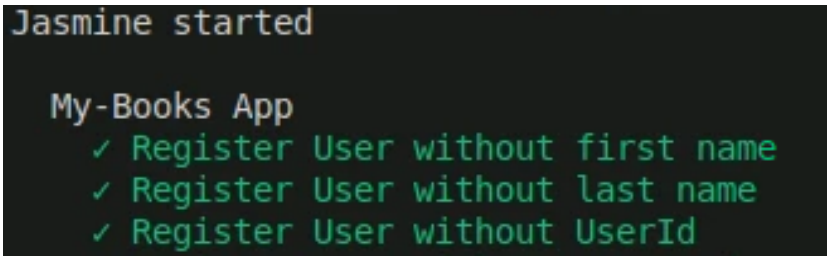


Fig:3.40 Register User without User Id result.

3.5.5 Register User without Password:

Here Register credentials are given without a password. This test case is done by only passing the first name, last name and User Id and the Password field is left null. In reality it should pose some error with a specific pop up message.

The expected message is “User Id and password are mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

Here the computed message passes the expected message hence passing the test case.

```
it('Register User without password', () => {
  page.clickOnRegisterButton();
  page.setFirstname("Kishan");
  page.setLastname("Patel");
  page.setuserid("kish123");
  page.clickOnRegisterUserButton();
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(3000);
  expect(val).toEqual("User Id and Password are mandatory");
  //page.setPassword("password");
});
```

Fig:3.41 Code for register user without Password.

```
Jasmine started

  My-Books App
    ✓ Register User without first name
    ✓ Register User without last name
    ✓ Register User without UserId
    ✓ Register User without password
```

Fig:3.42 Register User without Password result.

3.5.6 Register User without both UserId and Password:

Here Register credentials are given without User Id and password. This test case is done by only passing the first name and last name. The User Id and Password fields are left null. In reality it should pose some error with a specific pop up message.

The expected message is “User Id and password are mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

Here the computed message passes the expected message hence passing the test case.

```
it('Register User without UserID and password', () => {
  page.clickOnRegisterButton();
  page.setFirstname("Kishan");
  page.setLastname("Patel");
  page.clickOnRegisterUserButton();
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(3000);
  expect(val).toEqual("User Id and Password are mandatory");
  //page.setPassword("password");
});
```

Fig:3.43 Code for register user without User Id and Password.

```
My-Books App
  ✓ Register User without first name
  ✓ Register User without last name
  ✓ Register User without UserId
  ✓ Register User without password
  ✓ Register User without UserID and password
```

Fig:3.44 Register User without User Id and Password result.

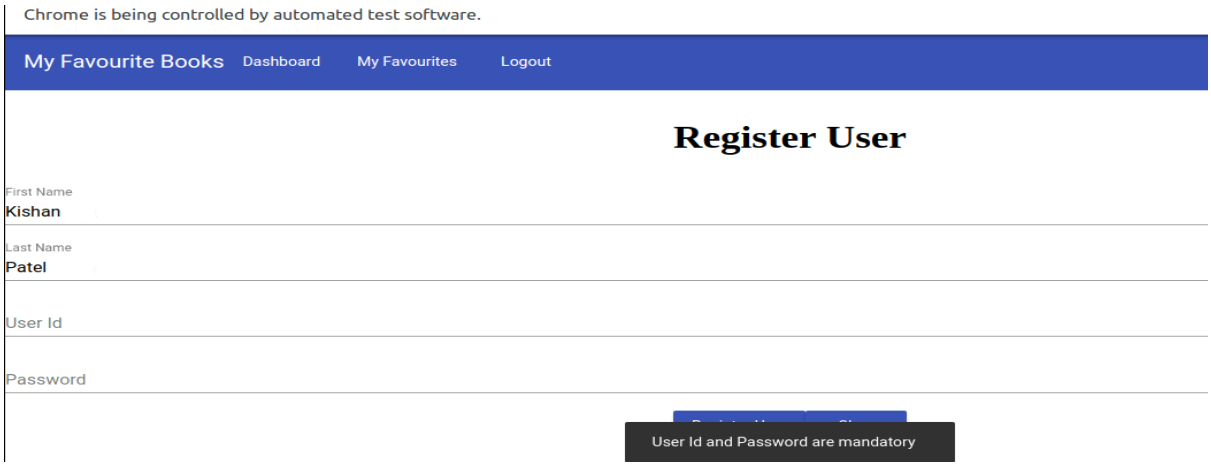


Fig3.45: Screenshot of register user without User Id and Password pop-up message.

3.5.7 Register User with already existing UserId:

Here all the Register credentials of an already existing user are given . This test case is done by passing all the fields of an existing user which includes first name, last name, user Id and password. In reality it should pose some error with a specific pop up message, if we are entering the same details with existing user id, since the user id should be unique.

The expected message is “User Id already exists”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

Here the computed message passes the expected message hence passing the test case.

```
it('Error while registering existing user', () => {
  page.clickOnRegisterButton();
  page.setFirstname("Kishan");
  page.setLastname("Patel");
  page.setuserid("kish123");
  page.setPassword("jkush");
  browser.sleep(2000);
  page.clickOnRegisterUserButton();
  //browser.sleep(3000);
  //page.clickOnClearButton();
  browser.sleep(3000);
  expect(browser.getCurrentUrl()).toEqual("http://localhost:4200/register");
  //page.setPassword("password");
});
```

Fig:3.46 Code for register user with already existing credentials.

```
Jasmine started

 My-Books App
   ✓ Register User without first name
   ✓ Register User without last name
   ✓ Register User without UserId
   ✓ Register User without password
   ✓ Register User without UserID and password
   ✓ Error while registering existing user
```

Fig:3.47 Register User already existing credentials result.

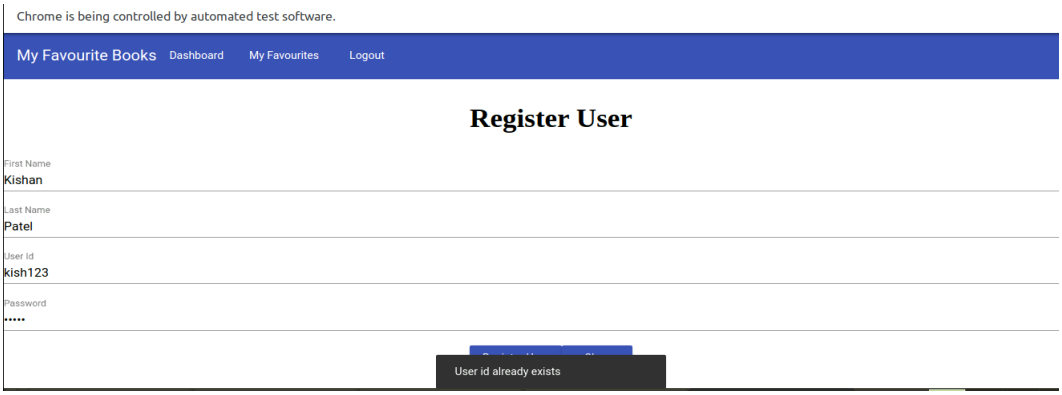


Fig:3.48 Screenshot of already existing user.

3.5.8 Clear Button Test:

After entering the registration details an user must be able to click the clear button and a fresh register user url page with no values on all the fields should be obtained.Here in this test case the user is able to click the clear button and a fresh register User page with null values on all the fields are obtained, hence passing the test case.

```
it('Test for clear button', () => {
  page.clickOnRegisterButton();
  page.setFirstname("Kishan");
  page.setLastname("Patel");
  page.setuserid("kish123");
  page.setPassword("jkush");
  //browser.sleep(2000);
  //page.clickOnRegisterUserButton();
  browser.sleep(3000);
  page.clickOnClearButton();
  browser.sleep(3000);
  expect(browser.getCurrentUrl()).toEqual("http://localhost:4200/register");
  //page.setPassword("password");
});
```

Fig:3.49 Code for clear button test

```
Jasmine started

  My-Books App
    ✓ Register User without first name
    ✓ Register User without last name
    ✓ Register User without UserId
    ✓ Register User without password
    ✓ Register User without UserID and password
    ✓ Error while registering existing user
    ✓ Test for clear button
```

Fig:3.50 Clear button test result.

3.6 Login Page:

In the login page the main functionality is for the user to be able to login after the specific registration has done.

3.6.1 Get Title Page:

In this test case the landing site is verified by checking the title of the browser. Initially navigating the browser to the landing page, the title of “My Favourite Books” is expected .

The expected and the computed title are verified and if matching, it passes the test case. The same findings are recorded here.

```
it('should display My Favourite Books as application title', () => {


  let spanElement = element(by.css('span')).getText();
  expect(spanElement).toEqual('My Favourite Books');
});
```

Fig 3.51: Get title page.

```
ubuntu@ip-172-31-33-156:~/Desktop/project/mybooksapp/MyBooksUI/e2e$ protractor protractor.conf.js
[11:20:39] I/launcher - Running 1 instances of WebDriver
[11:20:39] I/direct - Using ChromeDriver directly...
Jasmine started
[11:20:41] W/element - more than one element found for locator By(css selector, span) - the first result will be used

  My-Books App
    ✓ should display My Favourite Books as application title
```

Fig 3.52: Get title page result.

3.6.2 Login User Success:

Here the user must be able to login with the correct authentication details. In return to the successful login, the user must be directed to the dashboard url.

Thus the expected url is that of the dashboard. The expected url and the computed url will be verified. If it

matches it will pass the test case. The same findings are being recorded here.

```
it('Valid Login Test01', async()=>
  {
   page.setusername("vishnu");
   page.setPassword("vishnu1234");
   page.clickOnLoginButton();
   browser.sleep(3000);
   expect<any>(browser.getCurrentUrl()).toEqual('http://localhost:4200/books/home');
  });
```

Fig 3.53: User valid login.

```
  ✓ Valid Login Test01
```

Fig 3.54: User valid login result.

3.6.3 Login User Failure with only User Id:

Here an invalid login credentials are given. This test case is done by only passing the user Id and the password field is left null. In reality it should pose some error with a specific pop up message.

The expected message is “User Id and password are mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

```
it('Invalid Login test 01', async()=>{
  page.setusername('nash');
  browser.sleep(2000);
  page.clickOnLoginButton();
  browser.sleep(2000);
  let val = element(by.xpath('/html/body/div[1]')).getText();
  browser.sleep(2000);
  expect(val).toEqual('User Id and Password are mandatory');
})
```

Fig 3.55: Login user failure 01.

```
My-Books App
  ✓ should display My Favourite Books as application title
  ✓ Invalid Login test 01
```

Fig 3.56: Login user failure 01 result.

3.6.4 Login User Failure with Password only:

Here an invalid login credentials are given. This test case is done by only passing the password and the user Id field is left null. In reality it should pose some error with a specific pop-up message.

The expected message is “User Id and password are mandatory”. The inspect path of the span element is taken and using the “getText()” function, the text is taken. The expected and the computed message is verified and if matching, the test case is passed. The findings are recorded here.

```
it('Invalid Login test 02', async()=>{
  page.setPassword("nash123");
  browser.sleep(2000);
  page.clickOnLoginButton();
  browser.sleep(2000);
  let val = element(by.xpath('/html/body/div[1]')).getText();
  browser.sleep(2000);
  expect(val).toEqual('User Id and Password are mandatory');
})
```

Fig 3.57: Login user failure 02.

```
ubuntu@ip-172-31-33-156:~/Desktop/project/mybooksapp/MyBooksUI/e2e$ protractor protractor.conf.js
[11:20:39] I/launcher - Running 1 instances of WebDriver
[11:20:39] I/direct - Using ChromeDriver directly...
Jasmine started
[11:20:41] W/element - more than one element found for locator By(css selector, span) - the first result will be used

  My-Books App
    ✓ should display My Favourite Books as application title
    ✓ Invalid Login test 01
    ✓ Invalid Login test 02
```

Fig 3.58: Login user failure 02 result.

3.6.5 Login User Failure(Invalid Credentials):

Here a non existing user id and password are given. The test case gets initiated. If the specific user is not found it shall pose an error with some error message in it and return to the same login page.

The error message is “User not found” and the expected url is the login page itself as a non-existing user is not allowed to access further pages. The expected result and the computed result are verified. If it matches, the test case will be passed. The findings are recorded here.

```
it('Invalid Login Test 03', async()=>
{
 page.setusername("govindan");
 page.setPassword("gd123");
 page.clickOnLoginButton();
 browser.sleep(3000);
 expect<any>(browser.getCurrentUrl()).toEqual('http://localhost:4200/login');
});
```

Fig 3.59: Login user failure 03.

Fig 3.60: Login user failure 03 result.

3.7 Dashboard Page:

In the dashboard page the main functionality is the addition of the book after the search is done. For that the name of the book or the author is given for which the result will appear.

3.7.1 Saving The Book(Success) Test Case:

Here the landing site after the login is the dashboard page. The page is navigated to the prescribed page url. The testing is done in the POM methodology where the function defined in another class is imported into another class and calls the needed functions**.**

For every button and events to automate the element paths are inspected from the browser itself. Here the dropdown paths are inspected and assigned in a function called. For the search section the related xpath is set in the function called “searchForBook” and the book's details are given in it.

_Enter function is to be generated as no search button is provided(“protractor.Key.ENTER”). The click function for adding the book is given as “selectBookFromList”. After selection the system waits for the confirmation message which pop ups after the book is added. The expected text from the pop up is stored in a variable and verified with a computed message. If matched, it passes the test case.

```
it('should search for book and add the same 01',async()=>{

  page.searchForBook("hunter");
  browser.actions().sendKeys(protractor.Key.ENTER).perform();
  browser.sleep(3000);
  browser.driver.executeScript('window.scrollTo(94,188);').then(function() {
   page.selectBookFromList();
})
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(1000);
  expect(val).toEqual(`Laws, etc added to your favourites`)

})
```

Fig 3.61: Save Book Test Case.

After writing the code the test cases are run in an integrated terminal with the command “protractor protractor.conf.js”, which in turn starts the Jasmine and initiates the test case. If success the test description is shown with a green tick.

Fig.3.62: Save Book Success Test Case.

3.7.2 Saving the Book(Failure) Test Case:

For every button and events to automate the element paths are inspected from the browser itself. Here the dropdown paths are inspected and assigned in a function called. For the search section the related xpath is set in the function called “searchForBook” and the book's details are given in it.

Enter function is to be generated as no search button is provided. The click function for adding the book is given as “selectBookFromList”. After selection the system waits for the confirmation message which pop ups after the book is added. The expected text from the pop up is stored in a variable and verified with a computed message. If matched, it passes the test case. But here it is checked that if an existing book is added again it must show an error with a pop up message.

The book addition must fail and the error message is expected. The expected message is saved in a variable that is verified with the computed error message. If the message is “Book already exists in your favourites”, then the test case is passed.

```
it('should show error when same book is added',async()=>{

   page.clickOnAll();
   page.clickOnTitle();
project/mybooksapp/MyBooksUI/e2e/src/app.po.ts
                                    rollTo(0,0);').then(function() {
    page.searchForBook("hunter");
   })
    browser.actions().sendKeys(protractor.Key.ENTER).perform();
    browser.sleep(3000);
    browser.driver.executeScript('window.scrollTo(94,188);').then(function() {
     page.selectBookFromList();
 })
   browser.sleep(2000);
   let val = element(by.xpath("/html/body/div[1]")).getText();
   browser.sleep(1000);
   expect(val).toEqual(`Book already available in your Favourites`)

})
```

Fig.3.63: Save Book Negative Test Case.

Fig.3.64: Save Book Negative Test Case Result.

3.7.3 Verify the Books are Added:

For this event the browser must scroll back to the top of the page and click on the favourites section which is initiated by the function “clickOnFavourite” where the element path is stored and click() is provided.The expected url must be the favourites url and then return to the dashboard section.

```
it('should go to favourites to verify',async()=>{
  browser.sleep(3000);
  browser.driver.executeScript('window.scrollTo(0,0);').then(function() {
   page.clickOnFavourite();
  })
   browser.sleep(3000);
   page.clickOnDashboard();
})
```

Fig.3.65: Favourites Verification.

Fig.3.66: Favourite Verification Result.

3.7.4 Saving New Book(Success) Test Case:

Previously the book was added to test both the saving and saving the existing book test cases. Here a new book is added to verify that every other new book can be added to the favourites.

Enter function is to be generated as no search button is provided(“protractor.Key.ENTER”). The click function for adding the book is given as “selectBookFromList”. After selection the system waits for the confirmation message which pop ups after the book is added. The expected text from the pop up is stored in a variable and verified with a computed message. If matched, it passes the test case.

```
it('should search for book and add the same 02',async()=>{

   page.searchForBook("gama");
   browser.actions().sendKeys(protractor.Key.ENTER).perform();
   browser.sleep(3000);
   browser.driver.executeScript('window.scrollTo(94,188);').then(function() {
    page.selectBookFromList();
})
  browser.sleep(2000);
  let val = element(by.xpath("/html/body/div[1]")).getText();
  browser.sleep(1000);
  expect(val).toEqual(`Gama added to your favourites`)

})
```

Fig.3.67: Add New Book

```
My-Books App
  ✓ should search for book and add the same 01
  ✓ should show error when same book is added
  ✓ should go to favourites to verify
  ✓ should search for book and add the same 02
```

Fig.3.68: Add New Book Result.

3.8 Container Page:

In the project the container page performs the task of listing the favourite books being added from the dashboard page and the deletion of the same books are performed. Also the logout process is carried within this page.

3.8.1 Deleting the First and the Second Book:

Here the books that are saved from the dashboard are listed and the same can be deleted by the delet button provided below the preview.

For this the page navigates to the favourites after the login has been done. After that for accessing the delete button the element path is taken and assigned to a function.

As in the POM methodology the functions are implemented in the respective pages. For the scrolling function the inbuilt method **"scrollTo()"** is called as some time the browser page accessibility will be limited. For verification the span message **"Deleted from your favourites" is** taken.

After the deleting for verifying visually it navigates back to the dashboard and then back to the favourites section to make sure that the changes have been completely applied over the browser. After the verification process the logout functionality is carried on.

```
it('should delete the first book', async()=>{
    browser.sleep(2000);
    browser.driver.executeScript('window.scrollTo(94,188);').then(function() {
    page.clickOnDelete01();
    })
    browser.sleep(2000);

    let val = element(by.xpath('/html/body/div[1]')).getText();
    browser.sleep(2000);
    expect(val).toEqual('Laws, etc deleted from your favourites');
})
```

Fig:3.69 Delete the first book.

```
  it('should delete the second book', async()=>{
    browser.sleep(4000);
    browser.driver.executeScript('window.scrollTo(94,188);').then(function() {
    page.clickOnDelete02();
    })
    browser.sleep(2000);

    let val = element(by.xpath('/html/body/div[1]')).getText();
    browser.sleep(2000);
    expect(val).toEqual('Gama deleted from your favourites');
})
```

Fig:3.70 Delete the second book..

3.8.2 Logout from favourites:

Here the element path for the **"Logout"** link is taken from the navigation header. After the click event the page should navigate to the initial **"Login"** page. The expected url is the login page url.

If the expected url matches the computed url, the test case will get passed.

```
it("should logout", async()=>{
  browser.sleep(2000);
  browser.driver.executeScript('window.scrollTo(0,0);').then(function() {
    page.clickOnLogout();
  })
  browser.sleep(1000);
  expect(browser.getCurrentUrl()).toEqual("http://localhost:4200/login");
})
```

Fig:3.71 Logout from favourites

3.8.3 Deletion Failure:

Here the deletion failure purpose, the main aspect is to find whether the element for deletion is present or not. For that the "isPresent(_)" is assigned for the element of the specific book meant to be deleted. In return the element must not be present for the book after deletion thus returning the function output as false. Thus it passes the test case.

```
it('deletion failure', async()=>{
  browser.sleep(3000);
  browser.driver.executeScript('window.scrollTo(0,0);').then(function() {

  })
  browser.sleep(2000);

   browser.sleep(2000);
  let book =
element(by.xpath("/html/body/app-root/book-favourite/div/book-container/div/book-thumbnail[1]/mat-card/mat-card-header/div/mat-card-title")).isPresent
  expect(book).toEqual(false);

})
```

```
My-Books App
  ✓ should delete the first book
  ✓ should go to favourites to verify
  ✓ should delete the second book
  ✓ should logout
```

Fig:3.72 Overall container page result.

OVERALL FRONT-END E2E TEST RESULTS

```
ubuntu@ip-172-31-33-156:~/Desktop/project/mybooksapp/MyBooksUI/e2e$ protractor protractor.conf.js
[19:07:45] I/launcher - Running 1 instances of WebDriver
[19:07:45] I/direct - Using ChromeDriver directly...
Jasmine started

  My-Books App
    ✓ Error while registering existing user
    ✓ Register User without first name
    ✓ Register User without last name
    ✓ Register User without UserId
    ✓ Register User without password
    ✓ Register User without UserID and password
    ✓ Test for clear button
    ✓ should register new user

[19:10:52] W/element - more than one element found for locator By(css selector, span) - the first result will be used
  My-Books App
    ✓ should display My Favourite Books as application title
    ✓ Invalid Login test 01
    ✓ Invalid Login test 02
    ✓ Invalid Login Test 03
    ✓ Valid Login Test01

  My-Books App
    ✓ should search for book and add the same 01
    ✓ should show error when same book is added
    ✓ should go to favourites to verify
    ✓ should search for book and add the same 02
    ✗ should search for book and add the same 03
      - Expected 'Book already available in your Favourites' to equal 'Gama added to your favourites'.
```

```
My-Books App
  ✓ should delete the first book
  ✓ should go to favourites to verify
  ✓ should delete the second book
  ✓ deletion failure
  ✓ should logout
```

Fig:3.73 Overall e2e test result.

4. DEFECTS AND BUGS

For the software testing process reporting a bug or a defect is one of the most important parts of the process completion. The detailed defect report will be useful for the responsible developer to look after where the flaws of coding had occurred.

For this detailed behaviour of the bug or defect must be reported. For this purpose tabulation sheets can be used for accurate depiction of the flaw.

Here the detailed report of the defects found during the testing of **"My Books App"** are recorded.

4.1 Defect 1:

Missing search button:

This defect was found at the **"dashboard page"** of the **My Books App** browser page. After the book details for the search purpose is given in the search box page there must be a search button or instead the result must pop-up automatically.

Here after the book details are filled in the search box , manual key entering has to be done for the results to come in the dashboard page. During automation the inbuilt methods are written to perform the enter function.

The picture depiction is given below:

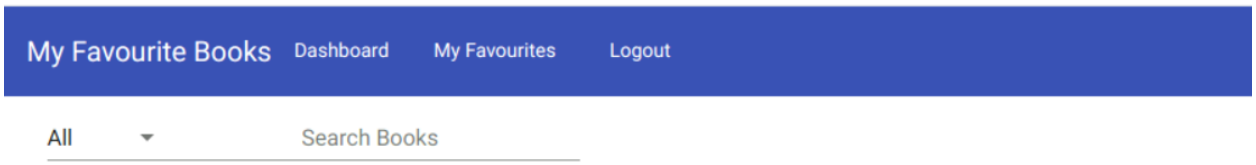


Fig4.1 Search box with no search button.

Here there is no search button to perform any search action.During automation procedure inbuilt code methods are implemented to access the enter function from the keyboard. This will actuate the process of the **"ENTER"** key as it is performed from an actual keyboard.

This defect is recorded with details in the defect report sheet.

4.2 Defect 2:

Not adding book with same name but different author:

This error was found during the addition of an extra book from the dashboard page. Here the books with almost the same name were selected. But in this case the book with the same name but with different authors, when tried to add to the favorites and error pop-up saying **"Book already available in your favourites"**.

For proper depiction of the defect scenario, the record from the browser and the front-end testing results are provided here:

Steps:

1. After logging in to the account , go to the search box page.
2. Give the sample name of the book as "Gama".
3. After giving details press the **"ENTER"** key.
4. Select the first book from results.
5. Pop-up for **"Gama added to your favourites"** will appear.
6. Select the second book from the same result list.
7. Pop-up for **"Book already available in your favourites"** will appear.

These are the steps followed for reaching the affected area.

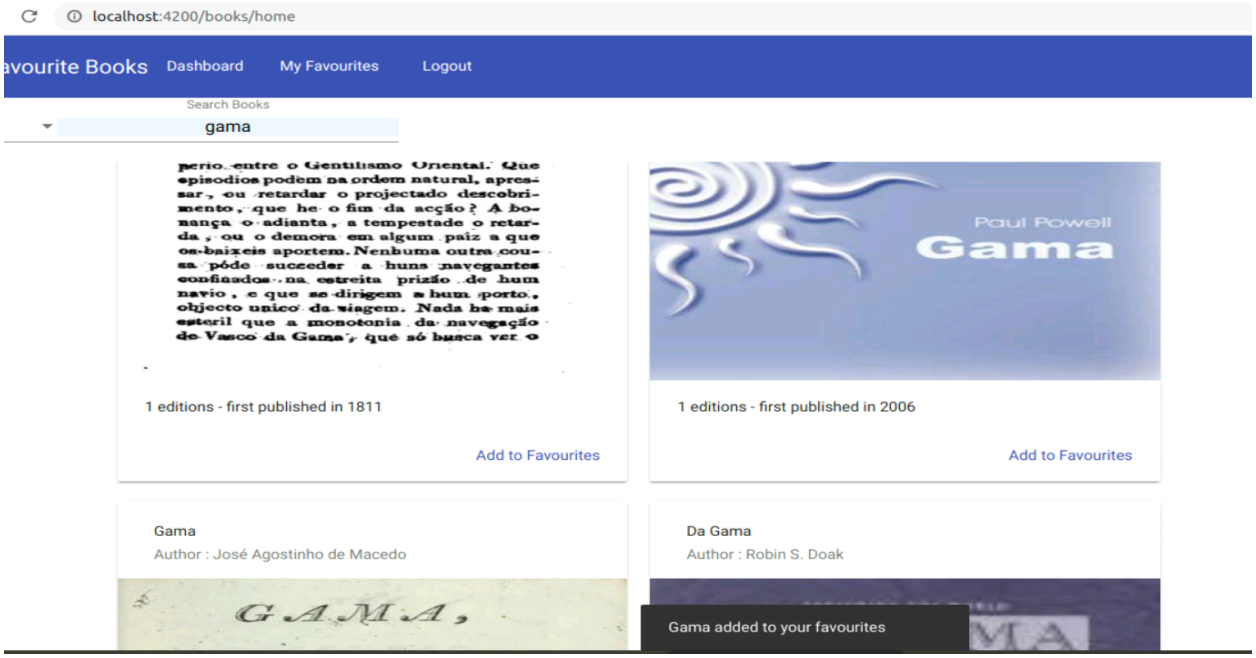


Fig 4.2. Pop-up for book being added.

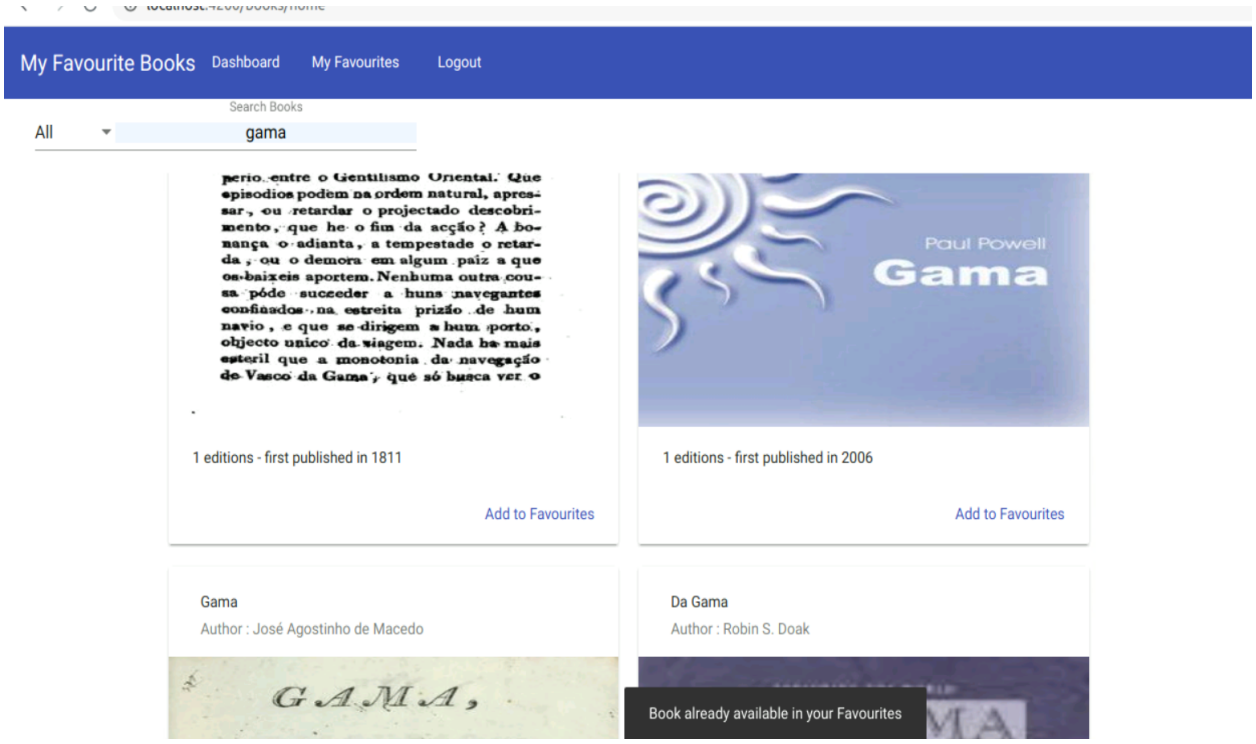


Fig 4.3. Pop-up for a book with the same name and different author.

Test results from the front-end testing:

```
it('should search for book and add the same 03',async()=>{

  browser.driver.executeScript('window.scrollTo(0,0);').then(function() {
  page.searchForBook("gama");
  browser.actions().sendKeys(protractor.Key.ENTER).perform();
  })

  browser.sleep(3000);
  browser.driver.executeScript('window.scrollTo(94,188);').then(function() {
   page.selectBookFromList02();
})
 browser.sleep(2000);
 let val = element(by.xpath("/html/body/div[1]")).getText();
 browser.sleep(1000);
 expect(val).toEqual(`Gama added to your favourites`)

})
```

Fig 4.4 Test case for the defect.

Fig 4.5 Test case fails for the defect.

Detailed Defect Report:

Defect ID	Defect description	Reproducible(Yes\No)	Steps to Reproduce	Priority	Severity	Reported By	Reported Date	Resolution	Stage / PD of Origin	Cause of Defect	Resolved By	Resolved Date	Remarks	Closed By	Closed Date	Defect Status	Resolution Status
MyBooks_D101	In dasboard page for My Books App, there is no search button for the search field. User may require to use ENTER key for the search purpose.	Yes	Login with valid credentials.->Go to search field->Give book details-> Press ENTER	Urgent	Critical	Batch - 02	25-02-2021	Major defect.Must be resolved for the search function to occur.		N/A						Open	Unresolved.
MyBooks_D102	While trying to add a book with same name but different author, a pop-up with message "Book already available in your favourites".	Yes	Login with valid credentials.->Go to search field->Give book details-> Add one book->Add another book with same name.	Urgent	Critical	Batch - 02	05-03-2021	Major defect. Must add book with same name and different authors.		When trying to add book with same name with different author.						Open	Unresolved.

Fig 4.6 Defect report.

5.BACKEND-DEVELOPMENT

For the backend development part, the task was to add a new column to the existing entity class **"User"**. The new column will be for the profile photo that can hold the reference of the image file to the specific user.

Next comes the updating and viewing of the user details. Thus the controller section, service and service implementations were taken and were tested using **"POSTMAN**".

The system requirements are:

Backend development IDE: **Spring Tool Suite(STS).**

Testing : **"POSTMAN".**

POSTMAN:

- Postman is a standalone software testing API (Application Programming Interface) platform to build, test, design, modify, and document APIs. It is a simple Graphic User Interface for sending and viewing HTTP requests and responses.
- While using Postman, for testing purposes, one doesn't need to write any HTTP client network code. Instead, we build test suites called collections and let Postman interact with the API.
- In this tool, nearly any functionality that any developer may need is embedded. This tool has the ability to make various types of HTTP requests like GET, POST, PUT, PATCH, and convert the API to code for languages like JavaScript and Python.

5.1 Creating Column for Profile and Taking File as Multipart File:

Here the first task was to create an extra column or field for the **"User"** entity class and provide the image type as a multipart file. Thus the controller, service and configuration file have been written for the same.

The needed constructor and proper getters and setters are also given within the same entity class.

5.1.1 User Entity Class:

```
@ApiModelProperty(name="User Id",notes="Id of the user",dataType="string")
@Id
@Column(length=50)
private String userId;

@ApiModelProperty(name="Password",notes="Password for the user",dataType="string")
@Column
private String password;

@ApiModelProperty(name="Profile",notes="Profie image for the user",dataType="string")
@Column
private String profile;

@ApiModelProperty(name="First Name",notes="First Name of the user",dataType="string")
@Column
private String firstName;

@ApiModelProperty(name="Last Name",notes="Last Name of the user",dataType="string")
@Column
private String lastName;


public User() {
    super();
}

public User(String userId, String password,String profile, String firstName, String lastName) {
    super();
    this.userId = userId;
    this.password = password;
    this.profile = profile;
    this.firstName = firstName;
    this.lastName = lastName;
}
```

```
public void setUserId(String userId) {
    this.userId = userId;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}


public String getProfile() {
    return profile;
}

public void setProfile(String profile) {
    this.profile = profile;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}
```

Fig:5.2 User entity class

5.1.2 Authentication Controller:

Here the authentication controller will have a change regarding the normal saving of the user. For the controller unit "Multipart" file is expected with an annotation **"@RequestParam("image")"** is called as an multipart file.For the image to get stored the file path has to be made available for which the **"StringUtils.cleanPath"** is used.The user profile is assigned with object of the location to be stored. The same will be responsible for reflecting the data towards the database.

For the user photo to get saved in the directory first it requires ion which reference if the user shall the image be saved. Thus for the same the **"userId"** is taken and assigned to the upload directory. It will help in the case of updations and verifications.

A separate utility file for the service of saving the photo to the directory is given.

```
 @ApiOperation(value="registerUser", notes="Register user for My Favourite Books App")
@PostMapping(path = AuthenticationConstants.REGISTER_PATH,consumes = {"multipart/mixed"})
public ResponseEntity<?>  registerUser(@ApiParam(name="User", value="User", required=true) @RequestBody User user,@RequestParam("image") MultipartFile multipartFile) th
    ResponseEntity<String> reponse;
    try {
        if(null == user.getFirstName() || user.getFirstName().isEmpty() ) {
            return new ResponseEntity<String>(AuthenticationConstants.FIRST_NAME_MSG,HttpStatus.BAD_REQUEST);
        }
        if(null == user.getLastName() || user.getLastName().isEmpty()) {
            return new ResponseEntity<String>(AuthenticationConstants.LAST_NAME_MSG,HttpStatus.BAD_REQUEST);
        }
        if(null == user.getUserId() || user.getUserId().isEmpty() ||
                null == user.getPassword() || user.getPassword().isEmpty()) {
            return new ResponseEntity<String>(AuthenticationConstants.USERID_MSG,HttpStatus.BAD_REQUEST);
        }
        if(null == user.getProfile() || user.getProfile().isEmpty()) {
            return new ResponseEntity<String>(AuthenticationConstants.PROFILE_MSG,HttpStatus.BAD_REQUEST);
        }
         String fileName = StringUtils.cleanPath(multipartFile.getOriginalFilename());
            user.setProfile(fileName);

        reponse = new ResponseEntity<>(authService.saveUser(user),HttpStatus.CREATED);


        String uploadDir = "user-photos/" + user.getUserId();

        FileUploadUtil.saveFile(uploadDir, fileName, multipartFile);
    }catch(UserAlreadyExistException e) {
        return new ResponseEntity<String>(e.getErrorMessage(),HttpStatus.CONFLICT);
    }
    return reponse;
```

Fig:5.2 Authentication controller.

5.1.3 File Upload Utility File:

The utility file being created proceeds with the saveFile(method). Here the byte wise data from the uploaded file is being taken with the help of a predefined method **".getInputStream()"**. The input byte stream checks for any existing byte files and overwrites it as given in the next step.

The path for saving is checked with an "if" statement and a new directory is called if no same exists. After that sequence of the image file has to be taken as bytes, for that the stream of bytes from the multipart file

are taken. If any file exists with the same type the **"StandardCopyOption"** is set to **"REPLACE_EXISTING"**.

```
package com.stackroute.userservice.service;

import java.io.*;

public class FileUploadUtil {

    public static void saveFile(String uploadDir, String fileName,
            MultipartFile multipartFile) throws IOException {
        Path uploadPath = Paths.get(uploadDir);

        if (!Files.exists(uploadPath)) {
            Files.createDirectories(uploadPath);
        }

        try (InputStream inputStream = multipartFile.getInputStream()) {
            Path filePath = uploadPath.resolve(fileName);
            Files.copy(inputStream, filePath, StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException ioe) {
            throw new IOException("Could not save image file: " + fileName, ioe);
        }
    }
}
```

Fig:5.3 File upload utility file.

5.1.4 Web MVC Configuration File:

For the results to be on the front end there must be resource handler configuration to the expose the directory path in which the image file is being uploaded.For this an MVC configurer file which implements "WebMVCConfigurer" is created where the image directory is exposed with the directory extensions.Handler handles the extensions and the resource path are added to the same and exposed.

```
1  package com.stackroute.userservice;
2
3  import java.nio.file.Path;
8
9  @Configuration
10 public class MvcConfig implements WebMvcConfigurer {
11
12
13     @Override
14     public void addResourceHandlers(ResourceHandlerRegistry registry) {
15         exposeDirectory("user-photos", registry);
16     }
17
18     private void exposeDirectory(String dirName, ResourceHandlerRegistry registry) {
19         Path uploadDir = Paths.get(dirName);
20         String uploadPath = uploadDir.toFile().getAbsolutePath();
21
22         if (dirName.startsWith("../")) dirName = dirName.replace("../", "");
23
24         registry.addResourceHandler("/" + dirName + "/**").addResourceLocations("file:/"+ uploadPath + "/");
25     }
26
```

Fig:5.4 MvcConfig file.

5.2 Updating The User:

Here it is required to create the update functionality for the user service which can bring changes in the existing user and save the updates to the database. For this a separate controller unit and service implementation is created.For identification of the specific user a specific logon token is passed before the existing details are fetched. This token is created by the JWT token filter that creates a specific token every time when a valid user logs in to the account.

The token describes the user details from which the userId is also fetched. This specific data of the user is given to the **"updateUser"** method. The same is created in the**" userService-services"** and the remaining task is carried on the service implementation class.In the service implementation class the existing user details are taken using **" get()"** and setting using **"set(user.getAttribute())"**. This gets the existing user details and overrides with the new value. Thus the **".save()"** in the repository is called to save the changes.

```
@PutMapping("/update")
public ResponseEntity<?> updateUserDetails(@RequestBody User user,HttpServletRequest request,
        HttpServletResponse response) {
    String authHeader = request.getHeader("authorization");
    String token = authHeader.substring(7);
    String userId = Jwts.parser().setSigningKey("bookApp").parseClaimsJws(token).getBody().getSubject();
    try {

        User updatedUser=new User();
     updatedUser = authService.updateUser(user,userId);
    return new ResponseEntity<>(updatedUser,HttpStatus.CREATED);
    }
    catch (Exception e) {
        return new ResponseEntity<String>("{\"message\":\"" + e.getMessage() + "\"}", HttpStatus.UNAUTHORI
    }
}
@GetMapping("/view")
```

Fig:5.5 Controller for update.

```
@Override
public User updateUser(User user,String userId) throws UserNotAvailableException {
    try {
    Optional<User> user1= authRepo.findById(userId);
        if(user1.isPresent()) {
            user1.get().setFirstName(user.getFirstName());
            user1.get().setLastName(user.getLastName());
            user1.get().setPassword(user.getPassword());
            user1.get().setProfile(user.getProfile());
            User user2=user1.get();
            authRepo.save(user2);

            return user2;
        }
        else
        {
            return null;
        }

   }catch(Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Fig:5.6 Service implementation for update.

The results of the update function are checked using **POSTMAN.** Here the api for the update function is called and the data to be updated is given in the JSON format. Thus the updates can be saved. For this to identify the user,user login is done using **POSTMAN** and the token is pasted to the bearer token in the authorization field.

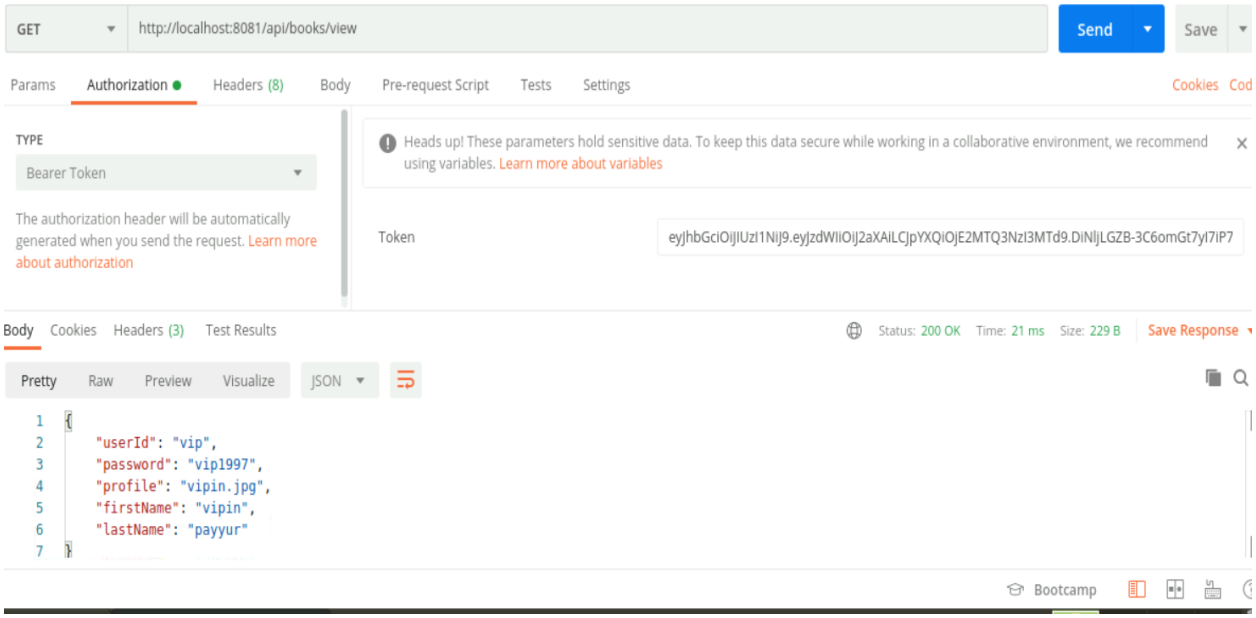


Fig 5.7 Token passing to the authorization field.

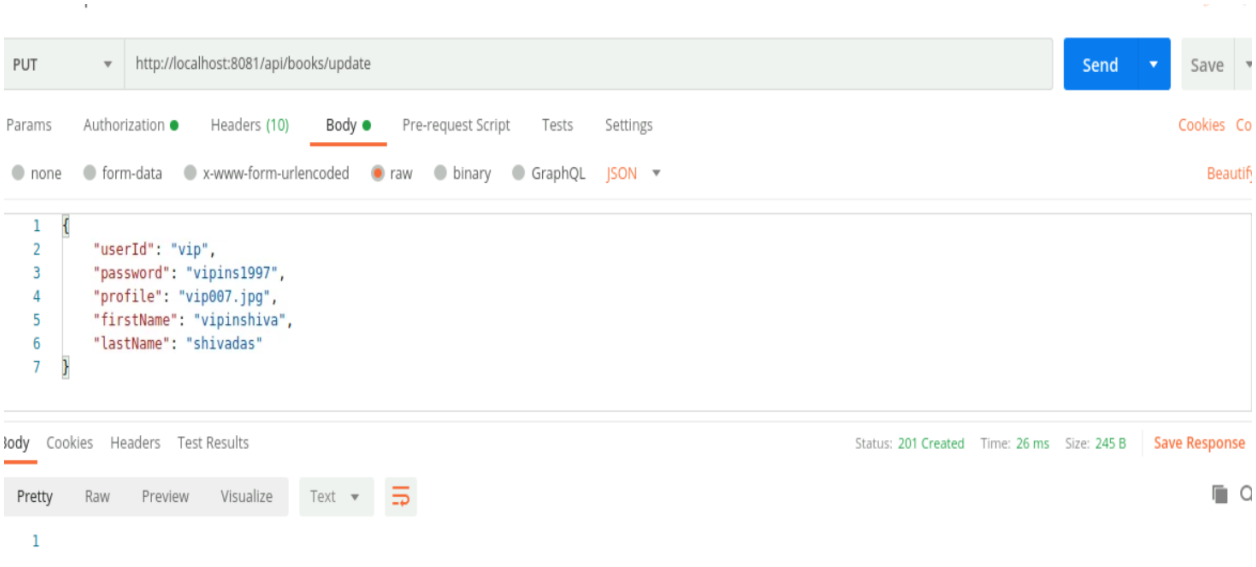


Fig 5.8 JSON values to be updated.

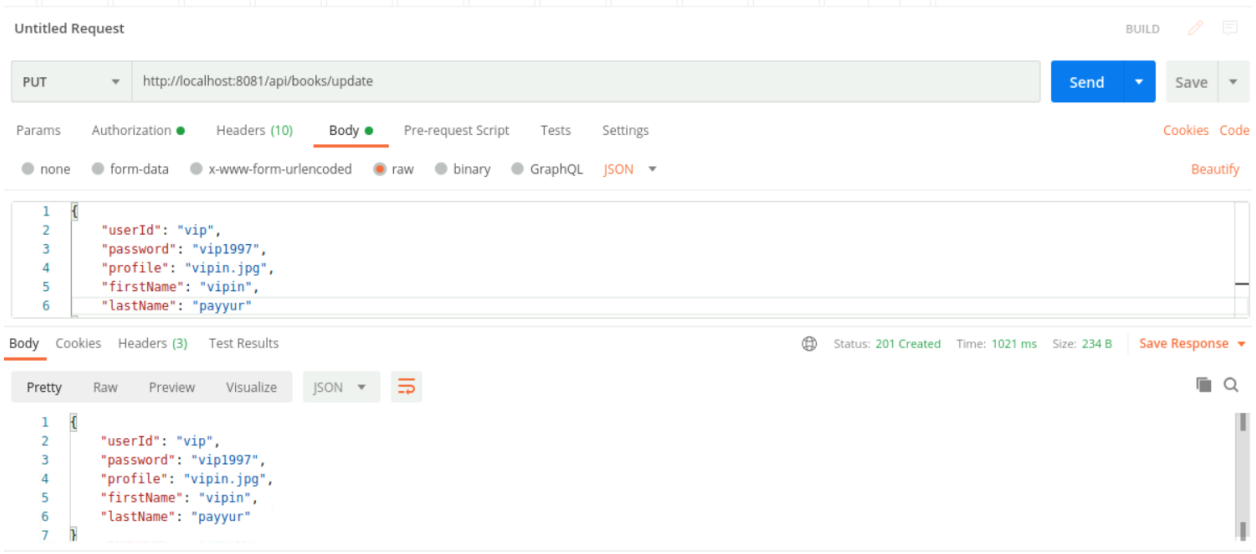


Fig 5.9 Update result

5.2 View User Details:

Here the specific user details have to be viewed For this **POSTMAN** is used. The token for the specified user is given in the **"userService-controller"** . The specific user details are fetched from the pojo class. The user parameters are selected and then the "viewUser()" method is called. The remaining task is carried in the service implementation section.

In service implementation class, the user is called by taking the userId from the user pojo and the **"findById(userId)"** is initiated from the repository.

The results are shown in the **POSTMAN**.

```
@GetMapping("/view")
public ResponseEntity<?> viewUserDetails(HttpServletRequest request,HttpServletResponse response) {
    String authHeader = request.getHeader("authorization");
    String token = authHeader.substring(7);
    String userId = Jwts.parser().setSigningKey("bookApp").parseClaimsJws(token).getBody().getSubject();
    try {
        User viewUser=new User();
     viewUser = authService.viewUser(userId);
    return new ResponseEntity<>(viewUser,HttpStatus.OK);
    }
    catch (Exception e) {
        return new ResponseEntity<String>("{\"message\":\"" + e.getMessage() + "\"}", HttpStatus.UNAUTHORIZED);
    }
}
```

Fig.5.10 Controller for view.

```
@Override
public User viewUser(String userId) throws UserNotAvailableException {
    try {
    Optional<User> user1= authRepo.findById(userId);
        if(user1.isPresent()) {
            User user2=user1.get();
            return user2;
        }
        else
        {
            return null;
        }

   }catch(Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Fig 5.11 Service implementation for view.

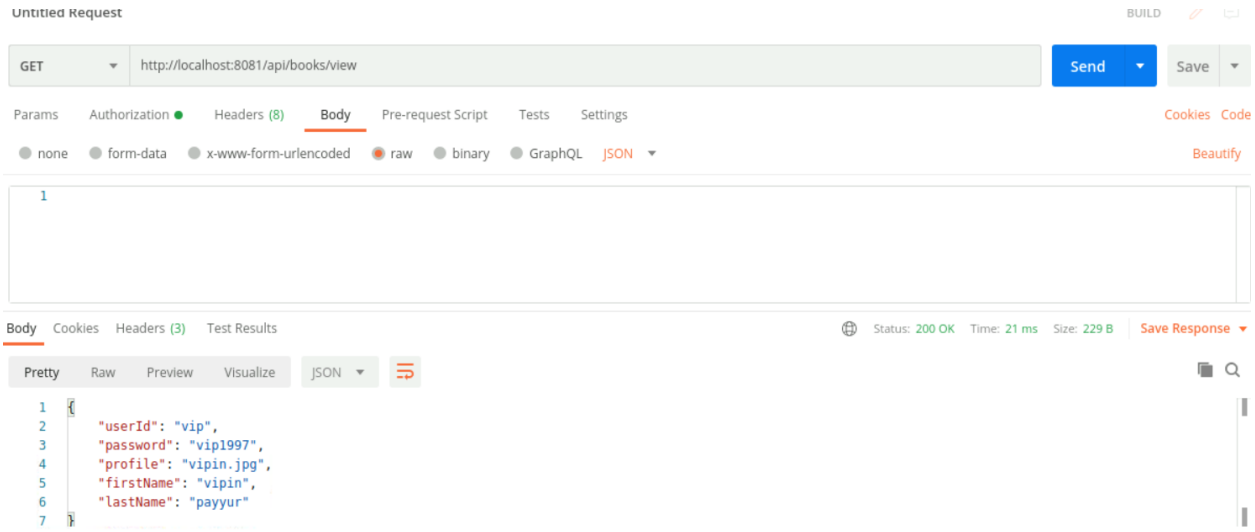


Fig 5.12 View result .

6.CONCLUSION

My Books App is developed to make users available with their favourite books at any instant. This app provides services like account creation, finding books and adding the books to the favourites etc. Thus as it contains user authentication there must be testing performed in order to make sure that everything works according to the needs.

Here the testing is performed in both the backend and the frontend as well.

For the backend there are:

- User Service Module.
- Favourite Service Module.

In the User Service module, there are userService controller and the userService services part.For the controller part **"Rest-Assured+TestNg"** testing frameworks are used while for services part "Mockito+Junit" framework. This trend is followed in both the modules in the project.

In the **"userService-controller"** section there using Rest-Assured, dummy values are given in which the task of saving the user and logging in of the user is provided.Various conditions in which the users can actually access the web page and way they can be controlled from accessing is tested here.

Trying to register with limited details and asserting the expected result with the computed result is done. If the event moves as expected the specific test case will be passed. Also the same is carried over for the user login process also, where valid and nvalid login credentials are given to test the expected result and the computed result, thus passing the test case if the verifications are successful.

For the **"userService-controller"** part no failing test cases have appeared till time from starting the test procedure.

In the "userService-services" section using the "Mockito" framework, mock values were injected to the service implementation class. The test cases were performed where to check the registration of users with valid details and insufficient details. The expected result and the computed results were verified by assertions and passing the test cases.

For the **"userService-services"** there were no failing cases appearing till time from starting the test procedure.

In the **"favouriteService-controller"** section using Rest-Assured framework dummy values were given

to test calling the path variable for specific functionality. As per the test cases given the expected status code and the computed status code were verified,if successful the test cases were a complete pass.

For **"favouriteService-controller"** there were no failing cases appearing from the start of performing the test cases.

In **"favouriteService-services"** using Mockito framework mock values were injected to the service implementation class. Same process was performed for every method followed for every functionality in the backend. The expected value and the computed value are verified by assertions and if matched it passes the test case.

For**"favouriteService-services"** there have been no failing cases appearing from the start of the testing procedure.

For the frontend there are:

- Register Page
- Login Page
- Dashboard Page
- Container Page

For the frontend testing **"Protractor+Jasmine"** framework were used. Here the POM methodology was implemented. This will help in the case of debugging if any error prevails while performing the test case. Here the login page and the register page were tested with limited details, valid and invalid details.

Each event causes a pop-up to appear accordingly. The expected message from the span is verified with the computed message. If verification passes the test case will pass.

For the login and register page there were no failing cases reported.

For the dashboard page,the search page element path was taken and book details were given using **"sendKeys()"** and the **ENTER** function had to be performed in automation as there is no search button available for the search event to occur.Favourite books were added and the books moved to the favourite container.

In dashboard page, there were some failing cases appearing:

1. There was no search button for the search function as the only option was to perform by using **ENTER** from the keyboard or the inbuilt code function to perform the search task. A button for this process should have been provided.
2. If the user tries to add a new book and another with the same name but different author, an pop-up with the message**"Book already available in your favourites"** appears.

All these defects have been documented and reported.

In the development part:

An extra column named **"profile"** was given to the user pojo class and the save method must include the profile as a multipart file in the database. The logic behind the process was implemented.

The same saved user has to be updated and the results were shown in **POSTMAN**. The update user was implemented in the controller as well as the service implementation class also. The specific tokens for a user were passed for authorization purpose and were given as a header function.

The view functions were to be performed using the same method as implemented for the update user. The controller and service implementation were given and the results were seen through the POSTMAN using the API path variable.

All the events and testing were recorded and well documented.