#### [Angular Integration and Mockito](javascript:void(0);)

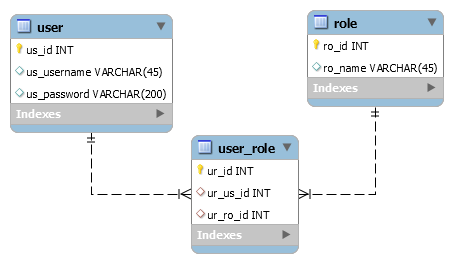
**Objectives**

* Demonstrate implementation of spring security integrated with repository to get user details
  + UserDetailsService, UserDetail, loadUserByUsername(), UsernameNotFoundException
    - Reference - https://docs.spring.io/spring-security/site/docs/5.2.1.BUILD-SNAPSHOT/reference/htmlsingle/#tech-userdetailsservice
* Demonstrate implementation of isolated testing of service using Mockito
  + Importance of isolated testing, MockitoJUnitRunner, Mockito.mock(), when(), thenReturn()
    - Spring Boot with Mockito - https://www.springboottutorial.com/spring-boot-unit-testing-and-mocking-with-mockito-and-junit
* Demonstrate code coverage report generation on unit test cases executed
  + Eclipse, ECL Emma, understanding the color coding
    - EMMA Reference - https://www.eclemma.org/
* Demonstrate end to end integration with Angular, RESTful Web Services and Spring Data JPA
  + Implement an event triggered in angular that executes a restful web service, which in turn invoke Spring Data JPA Repository that gets or persists data in database

Reference Code : FSE-ORM-029Complexity : Level1

**Integrate Spring Security with database**   
  
In "Spring RESTful Web Services" module, we implemented Spring Security with in memory database. In this hands on we will integrate Spring Security to use Spring Data JPA repository that manages the user details.  
  
**Prepare the Schema and Repository classes**

* Create relevant tables for storing user and role in ormlearn schema. (Refer the ER diagram in the end)
* Create classes User and Role with necessary many-to-many mapping
* Define roleList in User with eager fetch
* Create UserRepository with inclusion of a Query Method 'findByUsername' to get an user based on username.
* In user table create two users one for user and one for admin
* Make the password as $2a$10$R/lZJuT9skteNmAku9Y7aeutxbOKstD5xE5bHOf74M2PHZipyt3yK, which is the bcrypt encoding of the text 'pwd'
* Test if findByUsername() method works fine and retrieves the user and roles

​​​​​​​

**AppUser.java**  
This new bean class has to be implement UserDetails class of spring security framework, so that spring framework can read user details from this bean. Refer steps below to implement.

* Create bean class AppUser to hold the user details. This needs to be implement UserDetails class of spring security framework, so that spring framework can read user details from this bean. Refer steps below to implement:
* Right click on com.fis.spring-learn.security > New > Class
* Enter "Name" as "AppUser"
* In "Interfaces" click "Add.." and select "org.springframework.security.core.userdetails.UserDetails"
* Click Finish
* The above steps will create AppUser class and populate the code with all the methods that needs to be implemented.
* Include below instance variables in this class:

    private User user; // entity reference

    private Collection<? extends GrantedAuthority> authorities; // to store role details

* Create a constructor with user as parameter. Obtain the roles from the user and convert them to authorities. Refer code below for authorities conversion that needs to be included in the constructor.

        this.authorities = user.getRoleList().stream()

                .map(role -> new SimpleGrantedAuthority(role.getName())).collect(Collectors.toList());

* The above code converts the list of roles into a list of SimpleGrantedAuthority.
* Return authorities in getAuthorities() method
* Return user.getPassword() in getPassword() method
* Return true in all the below methods:
  + isAccountNonExpired()
  + isAccountNonLocked()
  + isCredentialsNonExpired()
  + isEnabled()

**AppUserDetailsService.java**

* Create a new class AppUserDetailsService in com.fis.spring-learn.security package.
* This class has to implement org.springframework.security.core.userdetails.UserDetailsService, so that spring security framework uses this class to retrieve user details.
* Autowire UserRepository
* Implement loadUserByUsername() method

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException

* Import UsernameNotFoundException from org.springframework.security.core.userdetails;
* Invoke findByUsername() method from UserRepository and get the user.
* If user is null throw UsernameNotFoundException
* If user is not null then create an instance of AppUser passing user obtained from repository.
* Return the AppUser reference.
* Test the execution of loadUserByUsername() to see if user details are retrieved along with roles.

**SecurityConfig.java**

* Autowire AppUserDetailsService
* In configure() method, comment out the in memory definition and bind spring framework to use the appUserDetailsService for authentication.

auth.userDetailsService(appUserDetailsService).passwordEncoder(passwordEncoder());

* Test the authentication web service using curl or postman.
* Reference Code : FSE-ORM-030Complexity : Level1
* **A Unit Test should be self sufficient**   
    
  As the name "Unit Testing" suggests, it is supposed to test a specific method in isolation without worrying about the dependencies.  
    
  Let us try to understand this with an example. Consider a Service method that calls a method in Repository. The service method is dependent on repository and the repository is dependent on database. When we have to test a service method, we should test it without the dependency on repository.  
    
  A good unit test should be isolated. Avoid dependencies such as environment settings, register values, or databases.  
    
  Let us consider the loadUserByUsername() method we implemented recently.  
    
  In this code below, find out the lines that has dependency on repository and database.
* public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
* LOGGER.info("Start");
* LOGGER.debug("UserRepository:{}", userRepository);
* User user = userRepository.findByUsername(username);
* LOGGER.debug("User:{}", user);
* if (user == null) {
* throw new UsernameNotFoundException(username);
* }
* LOGGER.info("End");
* return new AppPrincipalUser(user);
* }
* SME to walkthrough each line of the above code and identify the lines that has dependency.  
    
  In the subsequent hands on we will identify how to avoid the dependency using mockito.

Reference Code : FSE-ORM-031Complexity : Level1

**Implement integrated testing of loadUserByUsername() method** 

* Open spring-learn application in Eclipse
* Create a new package com.fis.spring-learn.service in src/test/java
* Create a new class UserDetailsServiceTest
* Define the following annotations at class level. This ensures that the class has spring application context.

@RunWith(SpringRunner.class)

@SpringBootTest

* Autowire UserDetailsService and implement the below test method. This method validates if the password returned is as per what is available in the database.

    @Test

    public void testLoadByUserName() {

        LOGGER.info("Start");

        UserDetails user = userDetailsService.loadUserByUsername("usr");

        String expected = "$2a$10$R/lZJuT9skteNmAku9Y7aeutxbOKstD5xE5bHOf74M2PHZipyt3yK";

        assertEquals(expected, user.getPassword());

        LOGGER.info("End");

    }

* Execute this as JUnit Test Case by right clicking on this test class and selecting Run > JUnit Test
* Check the logs to see that the service calls repository and it retrieves the data from database
* Ask yourself the following questions:
  + Is this method executed in isolation?
  + Is this test unit testing?

This kind of testing is integration testing and not unit testing.  
  
In the next hands on we will see how to isolate the testing using Mockito.

Reference Code : FSE-ORM-032Complexity : Level1

**Implement isolated testing with Mockito**   
  
Refer the code of the method to be tested:

    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        LOGGER.info("Start");

        LOGGER.debug("UserRepository:{}", userRepository);

        User user = userRepository.findByUsername(username);

        LOGGER.debug("User:{}", user);

        if (user == null) {

            throw new UsernameNotFoundException(username);

        }

        LOGGER.info("End");

        return new AppPrincipalUser(user);

    }

Our unit test cases should ensure that all the lines and all possible scenarios are tested.  
  
The above method can be tested in two scenarios:

* The method findByUsername() returns a valid
* The method findByUsername() returns null due to non-availability of user in the database

**Implement the steps below to test the first scenario:**

* Create a new test class UserDetailsServiceMockTest in src/test/java, com.fis.spring-learn.service package
* Annotate @RunWith with MockitoJUnitRunner.class (import org.mockito.junit.MockitoJUnitRunner)
* Annotate @SpringBootTest
* Include test method mockTestLoadUserByUsername()
* Implement following steps in the test method.
* Using Mockit.mock() method create a mock of the repository

UserRepository repository = Mockito.mock(UserRepository.class);

* Using below line of code define what happens when findByUsername() method in repository is called. This is how we are isolating the dependency.

when(repository.findByUsername("usr")).thenReturn(createUser());

* Due to the above defintion the database call will not happen, we need to simulate the return of User as part of this call. We achieve this by using a new createUser() method that takes care of creating a user instance with necessary password and role that is expected. Implement this method within the test class.
* Create a new constructor in AppUserDetailsService that accepts UserRepository as input and sets the userRepository instance variable.
* Create AppUserDetailsService instance using the mocked repository

AppUserDetailsService service = new AppUserDetailsService(repository); // repository refers to the mock repository created

* Invoke the method we want to test. When this method is executed, the repository's find method will not be invoke, but mockito will just return whatever is returned by createUser() method.

UserDetails user = service.loadUserByUsername("usr");

* Include the lines below to assert if the password matches.

String expected = "$2a$10$R/lZJuT9skteNmAku9Y7aeutxbOKstD5xE5bHOf74M2PHZipyt3yK";

assertEquals(expected, user.getPassword());

* Run the class as JUnit test and check if the test passes.

**Implement the mockito test for the second scenario when user is returned as null**

* Mock the findByUsername() to return null
* Pass the test case if UsernameNotFoundException is thrown. This can be implemented by asserting true in the catch block of UsernameNotFoundException handling. Include return after assert.
* Assert it as false after catch block

Reference Code : FSE-ORM-033Complexity : Level1

**Code Coverage of JUnit Testing**   
  
Following steps below to check the test coverage of JUnit testing:

* Right click on test class
* Select "Coverage As" > "JUnit Test"
* This executes the following actions:
  + Executes the unit test cases
  + Marks each line as red and green to denote if a line was executed or not. Open various files of your choice and check the relevance of red and green lines.
  + Generates a coverage report that can be seen in a new view "Coverage". Maximize the "Coverage" view, expand the packages to view the class wise code coverage percentage.

Reference Code : FSE-ORM-034Complexity : Level1

**Integrate employee web services with Spring Data JPA**   
  
Hope you all remember the following features implemented as part of angular hands on:

* Display list of employees
* Clicking edit link on an employee will open employee form with populated values
* Clicking save on edit form to persist the employee details
* Clicking delete link on an employee has to delete the employee

Currently the above implementations are done with in memory static data.  
  
Using repository and appropriate beans implement this with repository and service.   
  
Test in the angular application to see if the features work end to end.

Reference Code : FSE-ORM-035Complexity : Level1

**Implement end to end country search feature**   
  
Create a new component and route in angular application for searching country.  
  
This component should have a text box and a div in the bottom of the text box. This div will list the countries that matches with the keyword entered in text box. The countries within the div needs to be displayed line by line.  
  
Create necessary service in angular application that invokes the respective REST API created for returning the country list based on keyword.  
This REST API needs to be called on the key press event of the search text box.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1) | |  | | --- | | **Which class in Spring Security Framework is used to define role?** | | |  |  |  | | --- | --- | --- | |  | **1)** | **GrantedAuthority** | |  | 2) | Authority | |  | 3) | Role | |  | 4) | RoleAuthority | | |
| 2) | |  | | --- | | **Identify the class that helps to get the current spring security configuration** | | |  |  |  | | --- | --- | --- | |  | 1) | SecurityConfigHolder | |  | **2)** | **SecurityContextHolder** | |  | 3) | SecurityContext | |  | 4) | SecurityConfig | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3) | |  | | --- | | **What needs to be passed as input parameter of thenReturn() method?** | | |  |  |  | | --- | --- | --- | |  | **1)** | **The actual object that has to be returned by the mocked method** | |  | 2) | The object that needs to be mocked | |  | 3) | The class that needs to be mocked | |  | 4) | The method that needs to be mocked | | |
| 4) | |  | | --- | | **A unit test should be self sufficient** | | |  |  |  | | --- | --- | --- | |  | 1) | FALSE | |  | **2)** | **TRUE** | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5) | |  | | --- | | **A method in CountryService needs to be unit tested. In this scenario, identify the**  **right code that will mock the class.** | | |  |  |  | | --- | --- | --- | |  | **1)** | **CountryService service = Mockito.mock(CountryService.class);** | |  | 2) | CountryService service = Mockito.createMock(CountryService.class); | |  | 3) | Mockito.mock(CountryService.class); | |  | 4) | CountryService service = Mockito.generateMock(CountryService.class); | | |
| 6) | |  | | --- | | **Which method in UserDetailsService needs to be implemented, so that Spring**  **Security Framework can get the user details of a specific user?** | | |  |  |  | | --- | --- | --- | |  | **1)** | **loadUserByUsername()** | |  | 2) | loadUserByUserName() | |  | 3) | loadByUserName() | |  | 4) | loadByUsername() | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7) | |  | | --- | | **Select all aspects applicable for the statement "Unit Test should be self sufficient"** | | |  |  |  | | --- | --- | --- | |  | **1)** | **Avoid dependencies on file system** | |  | **2)** | **Avoid dependencies on database** | |  | 3) | Avoid dependencies within methods and classes | |  | **4)** | **Avoid dependencies on environment settings** | | |