Core-features: The backend functions are mentioned in this document. For user, project and module management, we have completed the frontend implementation as well. For the test case management, we prepared the backend. We have also added unit tests for methods in module and project management. We have prepared a dashboard for managing all the functionalities and created pages for facilitating login, logout features.

1. **User management** (login, logout, register, reset password) functionality is handled by a service called **UserService** in the backend. Inside which we have the methods such as **saveUser** for saving user information when someone tries to register, change or update user info (including password change). In this service, we also have methods for finding all the users that have registered into the system and this data can be fetched with **findAllUsers** method. This service class also contains methods such as **findByEmail** to find User information by their email. The code snippet provided below lists the codes from the service class that contains the methods we talked about:

```
public class UserServiceImpl implements UserService {
   private UserRepository userRepository;
   private RoleRepository roleRepository;
   private PasswordEncoder passwordEncoder;

   public UserServiceImpl(UserRepository userRepository, RoleRepository
roleRepository, PasswordEncoder passwordEncoder) { //no need for one
constructor to have @Autowired annotation.
       this.userRepository = userRepository;
       this.roleRepository = roleRepository;
       this.passwordEncoder = passwordEncoder;
   }


   /// User for saving and updating user information (during registration and
reset)

   @Override
   public void saveUser(UserDto userDto) {
       User user = new User();
       user.setName(userDto.getName());
       user.setEmail(userDto.getEmail());  //conversion of form data to jpa
entity, here mapper won't work as userDto don't have common attributes with
User.
       user.setPassword(passwordEncoder.encode(userDto.getPassword())); //
before setting the password we are encrypting using Bcrypt by Spring security.

       //userRepository.save(user);

       Role role = roleRepository.findByName("ROLE_USER");

       if (role == null) {
```

```
            role = checkRoleExists();
        }
        user.setRoles(Arrays.asList(role));   //As we have list of roles field
in user checking any role in db exists or not if not creating by a private
function and saving it in db
        userRepository.save(user); // now saving user to db.

    }

    //Used to find User information by it's email (email is unique)

    @Override
    public Optional<User> findByEmail(String email) {
        return userRepository.findByEmailIgnoreCase(email);
    }


    /// Used to get all registered user information available in the system.

    @Override
    public List<UserDto> findAllUsers() {
        List<User> users = userRepository.findAll();
        List<UserDto> user_dto = new ArrayList<>();
        for (User user : users) {
            UserDto userDto = new UserDto();
            userDto.setId(user.getId());
            userDto.setName(user.getName());
            userDto.setEmail(user.getEmail());
            user_dto.add(userDto);
        }
        return user_dto;
    }


    private Role checkRoleExists() {
        Role role = new Role();
        role.setName("ROLE_ADMIN");
        return roleRepository.save(role);
    }
}
```

Also, for filtering the requests (maintaining the user session and checking for authorized url), we have a filter to automatically do that.
The user model that we used to store and retrieve the data is:

```
@Entity
@Table(name = "users")
public class User {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @ManyToMany(fetch = FetchType.EAGER, cascade = {CascadeType.DETACH,
CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH})
    @JoinTable(
            name = "user_roles",
            joinColumns = {@JoinColumn(name = "user_id", referencedColumnName =
"id")},
            inverseJoinColumns = {@JoinColumn(name = "role_id",
referencedColumnName = "id")}
    )
    private List<Role> roles = new ArrayList<>();

//    @OneToMany(mappedBy = "sentTo", cascade = CascadeType.ALL, orphanRemoval
= true)
//    private List<Notification> notifications = new ArrayList<>();
}
```

It contains the necessary fields as well as the mappings.

2. **Project management** (Project create, delete, update and fetch all functionality) logic is written in a service class named **ProjectServiceImpl**. It contains all the project related business logic such as saving a project after creation (with **createProject** method), getting project information by it's project id from the database (with **getProjectById** method), updating project information (by **updateProject** method), deleting project information from the database (using **deleteProject** method) and finally pulling all project information stores in the database (**getAllProjects** method).

```
public class ProjectServiceImpl implements ProjectService {

    private final ProjectRepository projectRepository;

// Fetches all projects stored in the database
```

```
    @Override
    public List<Project> getAllProjects() {
        return projectRepository.findAll();
    }

// Fetches information about a individual project by searching it by its id.

    @Override
    public Optional<Project> getProjectById(Long id) {
        return projectRepository.findById(id);
    }

// For creating a new project and storing it in DB

    @Override
    public Project createProject(Project project) {
        return projectRepository.save(project);
    }

// For updating a existing a project in DB

    @Override
    public Project updateProject(Long id, Project projectDetails) {
        Project project = projectRepository
                .findById(id)
                .orElseThrow(() -> new RuntimeException("Project not found"));

        project.setProjectName(projectDetails.getProjectName());
        project.setDescription(projectDetails.getDescription());
        project.setStartDate(projectDetails.getStartDate());
        project.setProjectManager(projectDetails.getProjectManager());
        project.setStatus(projectDetails.getStatus());
        project.setClientName(projectDetails.getClientName());

        return projectRepository.save(project);
    }

// For deleting a project from the DB

    @Override
    public void deleteProject(Long id) {
        projectRepository.deleteById(id);
    }
}
```

The code for the Project entity model is provided as well to show it's mapping with other entities:

```
public class Project {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String projectName;
    private String description;

    @Temporal(TemporalType.DATE)
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private Date startDate;

    @ManyToOne
    @JoinColumn(name = "project_manager_id", nullable = false)
    private User projectManager;

    @Enumerated(EnumType.STRING)
    private ProjectStatus status;

    private String clientName;

    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Module> modules = new ArrayList<>();
}
```

3. **Module Management** (module creation, deletion, update, fetch all functionalities) logic is written in a service layer class named **ModuleServiceImpl**. It contains methods that enables functionalities such as module creation (with **add** method), module deletion (with **delete** method), fetching module by id (with **get** method), update module information (with **update** method), getting all modules (**getAll** method).

```
public class ModuleServiceImpl implements ModuleService {
    private final ModuleRepository moduleRepository;
    private final ProjectRepository projectRepository;

//Used to create and save a module to the database

    @Override
    public Module add(Module module) {
        return moduleRepository.save(module);
    }

//Used to fetch specific module from DB

    @Override
```

```
    public Module get(Long id) {
        return moduleRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Module with id " + id +
" not found"));
    }

// Used for updating module information

    @Override
    public Module update(Module module) {
        Module savedModule = moduleRepository.findById(module.getModuleId())
                .orElseThrow(() -> new RuntimeException("Module with id " +
module.getModuleId() + " not found"));
        savedModule.setModuleName(module.getModuleName());
        savedModule.setDescription(module.getDescription());
        savedModule.setTestCases(module.getTestCases());
        return moduleRepository.save(savedModule);
    }

// Used for deleting a certain module

    @Override
    public void delete(Long id) {
        moduleRepository.deleteById(id);
    }

    @Override
    public List<Module> getAllByProjectId(Long projectId) {
        Project project = projectRepository.findById(projectId)
                .orElseThrow(() -> new RuntimeException("Project with id " +
projectId + " not found"));
        return moduleRepository.getModulesByProject(project);
    }


// Used to get all modules
    @Override
    public List<Module> getAll() {
        return moduleRepository.findAll();
    }
}
```

The module entity looks like the following:

```
public class Module {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long moduleId;
```

```
    @Column(nullable = false)
    private String moduleName;

    private String description;

    @OneToMany(mappedBy = "module", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<TestCase> testCases = new ArrayList<>();

    @ManyToOne
    private Project project;
}
```

4. **TestCase Management** methods (create, delete, find, fetch all and update) are written in **TestCaseServiceImpl** class. The **save** method is written for test case creation, **get** method is for test case fetching, **update** is for updating a test case information, **delete** is for deleting test case and to fetch all test case under a module, we have method named **getByModuleId**.

```
public class TestCaseServiceImpl implements TestCaseService {
    private final TestCaseRepository testCaseRepository;
    private final ModuleRepository moduleRepository;

// For getting a test case information using it's ID from DB
    @Override
    public TestCase get(Long id) {
        return testCaseRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("TestCase with id " + id
+ " not found"));
    }

//Storing test case to DB

    @Override
    public TestCase save(TestCase testCase) {
        return testCaseRepository.save(testCase);
    }

//Deleting test case from DB

    @Override
    public void delete(Long id) {
        testCaseRepository.deleteById(id);
    }

/// Updating test case information
    @Override
```

```
    public TestCase update(TestCase testCase) {
        TestCase savedTestCase =
testCaseRepository.findById(testCase.getTestCaseId())
                .orElseThrow(() -> new RuntimeException("TestCase with id " +
testCase.getTestCaseId() + " not found"));
        savedTestCase.setAssignedTo(testCase.getAssignedTo());
        savedTestCase.setDescription(testCase.getDescription());
        savedTestCase.setDueDate(testCase.getDueDate());
        savedTestCase.setPriority(testCase.getPriority());
        savedTestCase.setStatus(testCase.getStatus());
        return testCaseRepository.save(savedTestCase);
    }

// Getting all test cases under a module

    @Override
    public List<TestCase> getByModuleId(Long moduleId) {
        Module module = moduleRepository.findById(moduleId)
                .orElseThrow(() -> new RuntimeException("Module not found"));

        return testCaseRepository.getTestCasesByModule(module);
    }
}
```

5. **Unit tests for ProjectService**.

They cover create, delete, update, fetch all and get functionality.
Comments are made to explain the test cases in the code block to help understand the functionality of the test cases.

Below are test cases for ProjectService class:

```
@ExtendWith(MockitoExtension.class)
class ProjectServiceImplTest {

    @Mock
    private ProjectRepository projectRepository;

    @InjectMocks
    private ProjectServiceImpl projectService;

    private Project project;
    private User projectManager;

    @BeforeEach
    void setUp() {
        projectManager = new User();
        projectManager.setId(100L);
```

```
        project = new Project();
        project.setId(1L);
        project.setProjectName("Test Project");
        project.setDescription("Project Description");
        project.setStartDate(new Date());
        project.setProjectManager(projectManager);
        project.setClientName("Test Client");
    }

// Test to check if fetch all works or not.
    @Test
    void testGetAllProjects() {

when(projectRepository.findAll()).thenReturn(Arrays.asList(project));

        List<Project> projects = projectService.getAllProjects();

        assertFalse(projects.isEmpty());
        assertEquals(1, projects.size());
        assertEquals("Test Project", projects.get(0).getProjectName());
        verify(projectRepository, times(1)).findAll();
    }

// Test to check if project can be fetched by it's id

    @Test
    void testGetProjectById_Found() {

when(projectRepository.findById(1L)).thenReturn(Optional.of(project));

        Optional<Project> foundProject =
projectService.getProjectById(1L);

        assertTrue(foundProject.isPresent());
        assertEquals("Test Project", foundProject.get().getProjectName());
        verify(projectRepository, times(1)).findById(1L);
    }

// Test to check if project can be fetched by it's id


    @Test
    void testGetProjectById_NotFound() {
        when(projectRepository.findById(1L)).thenReturn(Optional.empty());

        Optional<Project> foundProject =
projectService.getProjectById(1L);
```

```
        assertTrue(foundProject.isEmpty());
        verify(projectRepository, times(1)).findById(1L);
    }

// test to check if project creation works or not

    @Test
    void testCreateProject() {
        when(projectRepository.save(project)).thenReturn(project);

        Project savedProject = projectService.createProject(project);

        assertNotNull(savedProject);
        assertEquals("Test Project", savedProject.getProjectName());
        verify(projectRepository, times(1)).save(project);
    }

// test to check if project update is successful or not

    @Test
    void testUpdateProject_Success() {
        Project updatedProjectDetails = new Project();
        updatedProjectDetails.setProjectName("Updated Project");
        updatedProjectDetails.setDescription("Updated Description");
        updatedProjectDetails.setStartDate(new Date());
        updatedProjectDetails.setProjectManager(projectManager);
        updatedProjectDetails.setClientName("Updated Client");


when(projectRepository.findById(1L)).thenReturn(Optional.of(project));

when(projectRepository.save(any(Project.class))).thenReturn(updatedProjec
tDetails);

        Project result = projectService.updateProject(1L,
updatedProjectDetails);

        assertNotNull(result);
        assertEquals("Updated Project", result.getProjectName());
        assertEquals("Updated Description", result.getDescription());
        verify(projectRepository, times(1)).findById(1L);
        verify(projectRepository, times(1)).save(any(Project.class));
    }

// test to check if project update is successful or not

    @Test
    void testUpdateProject_NotFound() {
        Project updatedProjectDetails = new Project();
```

```
        updatedProjectDetails.setProjectName("Updated Project");

        when(projectRepository.findById(1L)).thenReturn(Optional.empty());

        Exception exception = assertThrows(RuntimeException.class, () ->
projectService.updateProject(1L, updatedProjectDetails));

        assertEquals("Project not found", exception.getMessage());
        verify(projectRepository, times(1)).findById(1L);
    }

// test to check if project deletion works or not.

    @Test
    void testDeleteProject() {
        doNothing().when(projectRepository).deleteById(1L);

        projectService.deleteProject(1L);

        verify(projectRepository, times(1)).deleteById(1L);
    }
}
```

6. **Unit tests for ModuleService**.
   They cover create, delete, update, fetch all and get functionality.
   Comments are made to explain the test cases in the code block to help understand the functionality of the test cases.

   Below are the test cases for ModuleService class:

```
@ExtendWith(MockitoExtension.class)
class ModuleServiceImplTest {

    @Mock
    private ModuleRepository moduleRepository;

    @Mock
    private ProjectRepository projectRepository;

    @InjectMocks
    private ModuleServiceImpl moduleService;

    private Module module;
    private Project project;

    @BeforeEach
    void setUp() {
```

```
        User projectManager = new User();
        projectManager.setId(100L);

        project = new Project();
        project.setId(1L);
        project.setProjectName("Test Project");
        project.setDescription("Project Description");
        project.setStartDate(new Date());
        project.setProjectManager(projectManager);
        project.setClientName("Test Client");

        module = new Module();
        module.setModuleId(1L);
        module.setModuleName("Test Module");
        module.setDescription("Module Description");
        module.setProject(project);
    }

// test to check if add module functionality works or not

    @Test
    void testAddModule() {
        when(moduleRepository.save(module)).thenReturn(module);

        Module savedModule = moduleService.add(module);

        assertNotNull(savedModule);
        assertEquals("Test Module", savedModule.getModuleName());
        assertEquals(project, savedModule.getProject());
        verify(moduleRepository, times(1)).save(module);
    }

// test to check if module can be fetched by id or not

    @Test
    void testGetModuleById() {

when(moduleRepository.findById(1L)).thenReturn(Optional.of(module));

        Module foundModule = moduleService.get(1L);

        assertNotNull(foundModule);
        assertEquals(1L, foundModule.getModuleId());
        verify(moduleRepository, times(1)).findById(1L);
    }

// test to check if module can be fetched by id or not
```

```
    @Test
    void testGetModuleById_NotFound() {
        when(moduleRepository.findById(1L)).thenReturn(Optional.empty());

        Exception exception = assertThrows(RuntimeException.class, () ->
moduleService.get(1L));

        assertEquals("Module with id 1 not found",
exception.getMessage());
        verify(moduleRepository, times(1)).findById(1L);
    }

// Test to check if module can be updated or not

    @Test
    void testUpdateModule() {

when(moduleRepository.findById(1L)).thenReturn(Optional.of(module));
        when(moduleRepository.save(any(Module.class))).thenReturn(module);

        Module updatedModule = new Module();
        updatedModule.setModuleId(1L);
        updatedModule.setModuleName("Updated Module");
        updatedModule.setDescription("Updated Description");

        Module result = moduleService.update(updatedModule);

        assertNotNull(result);
        assertEquals("Updated Module", result.getModuleName());
        assertEquals("Updated Description", result.getDescription());
        verify(moduleRepository, times(1)).findById(1L);
        verify(moduleRepository, times(1)).save(any(Module.class));
    }

// Test to check if module can be updated or not (negative case)
    @Test
    void testUpdateModule_NotFound() {
        when(moduleRepository.findById(1L)).thenReturn(Optional.empty());

        Exception exception = assertThrows(RuntimeException.class, () ->
moduleService.update(module));

        assertEquals("Module with id 1 not found",
exception.getMessage());
        verify(moduleRepository, times(1)).findById(1L);
    }

// test to check if module can be deleted or not
```

```
    @Test
    void testDeleteModule() {
        doNothing().when(moduleRepository).deleteById(1L);

        moduleService.delete(1L);

        verify(moduleRepository, times(1)).deleteById(1L);
    }

// test to check if all modules stored in DB can be fetched or not

    @Test
    void testGetAllModulesByProjectId() {

when(projectRepository.findById(1L)).thenReturn(Optional.of(project));

when(moduleRepository.getModulesByProject(project)).thenReturn(Arrays.asL
ist(module));

        List<Module> modules = moduleService.getAllByProjectId(1L);

        assertFalse(modules.isEmpty());
        assertEquals(1, modules.size());
        assertEquals("Test Module", modules.get(0).getModuleName());
        verify(projectRepository, times(1)).findById(1L);
        verify(moduleRepository, times(1)).getModulesByProject(project);
    }

// test to check if all modules stored in DB can be fetched or not
(negative case)

    @Test
    void testGetAllModulesByProjectId_ProjectNotFound() {
        when(projectRepository.findById(1L)).thenReturn(Optional.empty());

        Exception exception = assertThrows(RuntimeException.class, () ->
moduleService.getAllByProjectId(1L));

        assertEquals("Project with id 1 not found",
exception.getMessage());
        verify(projectRepository, times(1)).findById(1L);
    }

//// test to check if all modules (regardless or the project) stored in
DB can be fetched or not

    @Test
    void testGetAllModules() {
```

```
when(moduleRepository.findAll()).thenReturn(Arrays.asList(module));

        List<Module> modules = moduleService.getAll();

        assertFalse(modules.isEmpty());
        assertEquals(1, modules.size());
        verify(moduleRepository, times(1)).findAll();
    }
}
```

7. Dashboard ui is written as a thymleaf template and the code can be found in **src/main/resource/static/templates** directory of the project.
8. Login and logout ui pages can be found in **src/main/resource/static/templates** directory of the project.
9. Pages for module and project creation can be found in **src/main/resource/static/templates/modules** and **src/main/resource/static/templates/projects** folders respectively.

Finally, we have also designed a page to create test cases from the ui, but it's not fully tested. It's code can be found in **src/main/resource/static/templates/tests** directory.