Healthcare Management System - Testing Guide

This document provides detailed information about testing strategies implemented in the Healthcare Management System, with a focus on unit testing using Mockito.

Table of Contents

1.Overview

The Healthcare Management System employs a comprehensive testing approach to ensure code quality, functionality, and reliability. The testing strategy includes:

- Unit testing with Mockito for isolated component testing
- Integration testing for validating component interactions
- Test-driven development practices for key functionality

2. Testing Strategy

Test Types

- Unit Tests : Test individual components in isolation
- Integration Tests : Test interactions between components
- Service Tests : Test service layer functionality
- Controller Tests : Test API endpoints and request handling
- Repository Tests : Test data access layer

3.Testing Tools

- JUnit 5 : Testing framework
- Mockito : Mocking framework for unit tests
- Spring Test : Spring Boot testing utilities
- H2 Database : In-memory database for testing

4.Unit Testing with Mockito

Introduction to Mockito

Mockito is a mocking framework used to create and configure mock objects for isolated unit testing. It enables testing of classes by mocking their dependencies, allowing focus on testing a single unit of code without dependencies affecting the test outcome.

5.Key Mockito Features Used

- Mock creation
- Behavior stubbing
- Argument matching
- Verification
- Spy objects
- Argument captors

6. Sample Test Structure

java

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @Mock
    private PasswordEncoder passwordEncoder;

    @InjectMocks
    private UserService userService;
```

```
    @Test
    public void testFindByEmail() {
        // Arrange
        String email = "test@example.com";
        User expectedUser = new User();
        expectedUser.setEmail(email);


when(userRepository.findByEmail(email)).thenReturn(Optional.of(expectedUser)
);

        // Act
        Optional<User> result = userService.findByEmail(email);

        // Assert
        assertTrue(result.isPresent());
        assertEquals(email, result.get().getEmail());
        verify(userRepository, times(1)).findByEmail(email);
    }

    @Test
    public void testUpdateUser() {
        // Arrange
        User user = new User();
        user.setId(1L);
        user.setFirstName("John");
        user.setLastName("Doe");

        when(userRepository.save(any(User.class))).thenReturn(user);

        // Act
        User savedUser = userService.updateUser(user);

        // Assert
        assertNotNull(savedUser);
        assertEquals("John", savedUser.getFirstName());
```

```
        verify(userRepository).save(user);
    }
}
```

7.Service Layer Testing

The service layer contains the core business logic and is extensively tested using Mockito to mock repository dependencies:

```
 java
@ExtendWith(MockitoExtension.class)
public class AppointmentServiceTest {

    @Mock
    private AppointmentRepository appointmentRepository;

    @Mock
    private UserService userService;

    @Mock
    private DoctorService doctorService;

    @InjectMocks
    private AppointmentService appointmentService;

    @Test
    public void testCreateAppointment() {
        // Arrange
        User patient = new User();
        patient.setId(1L);

        Doctor doctor = new Doctor();
        doctor.setId(1L);

        Appointment appointment = new Appointment();
        appointment.setPatient(patient);
```

```
        appointment.setDoctor(doctor);
        appointment.setAppointmentDateTime(LocalDateTime.now().plusDays(1));
        appointment.setStatus(Appointment.Status.SCHEDULED);


when(appointmentRepository.save(any(Appointment.class))).thenReturn(appoin
tment);

        // Act
        Appointment created =
appointmentService.saveAppointment(appointment);

        // Assert
        assertNotNull(created);
        assertEquals(Appointment.Status.SCHEDULED, created.getStatus());
        verify(appointmentRepository).save(appointment);
    }
}
```

8.Controller Layer Testing

Controllers are tested using `MockMvc` from Spring Test, with dependencies mocked using Mockito:

java

```
@ExtendWith(MockitoExtension.class)
@WebMvcTest(ProfileController.class)
public class ProfileControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @MockBean
```

```
    private PatientService patientService;

    @Test
    @WithMockUser(username = "test@example.com")
    public void testViewProfile() throws Exception {
        // Arrange
        User user = new User();
        user.setEmail("test@example.com");
        user.setFirstName("Test");
        user.setLastName("User");

        Patient patient = new Patient();
        patient.setUser(user);

        when(userService.findByEmail(anyString())).thenReturn(Optional.of(user));

when(patientService.findByUser(any(User.class))).thenReturn(Optional.of(patien
t));

        // Act & Assert
        mockMvc.perform(get("/profile"))
            .andExpect(status().isOk())
            .andExpect(view().name("profile/view"))
            .andExpect(model().attributeExists("user"))
            .andExpect(model().attributeExists("patient"));

        verify(userService).findByEmail("test@example.com");
        verify(patientService).findByUser(user);
    }
}
```

9. Repository Layer Testing

Repositories are tested using the Spring Data JPA test utilities:

java

```
@DataJpaTest
public class UserRepositoryTest {

   @Autowired
   private UserRepository userRepository;

   @Test
   public void testFindByEmail() {
      // Arrange
      User user = new User();
      user.setEmail("test@example.com");
      user.setPassword("password");
      user.setFirstName("Test");
      user.setLastName("User");
      userRepository.save(user);

      // Act
      Optional<User> found = userRepository.findByEmail("test@example.com");

      // Assert
      assertTrue(found.isPresent());
      assertEquals("test@example.com", found.get().getEmail());
   }
}
```

10. Integration Testing

Integration tests use the Spring Boot test framework to ensure components work together:

java

```
@SpringBootTest
public class AppointmentIntegrationTest {

   @Autowired
   private AppointmentService appointmentService;
```

```
    @Autowired
    private UserService userService;

    @Autowired
    private DoctorService doctorService;

    @Test
    public void testAppointmentCreationFlow() {
        // Test the full appointment creation flow
        // involving multiple services
    }
}
```

11.Test Coverage

The project aims for high test coverage with particular focus on:

- Core business logic in service classes
- Controller endpoints
- Data access methods
- Security configurations