# **Семинар №10 Spring Testing Junit и Mockito для написания тестов**

1. **Инструментарий:**

[Урок](https://docs.google.com/document/d/18x8nHJlK7YSUO7mtUrvTqDwZpfGu2EKX/edit?usp=drive_link&ouid=117975875178229561167&rtpof=true&sd=true)

[Презентация](https://docs.google.com/presentation/d/1MCrLn-ZFXf8NeFq3NMrFMLOItgVzXXVx/edit?usp=drive_link&ouid=117975875178229561167&rtpof=true&sd=true)

1. **Цели семинара №10:**

* Познакомить слушателей с основами тестирования в Spring с использованием JUnit.
* Рассмотреть принципы создания mock-объектов с помощью Mockito и их интеграцию с Spring.

По итогам семинара №10 слушатель должен **знать**:

* Что такое Spring Test Context Framework и для чего он используется.
* Основные аннотации JUnit и их назначение (например, @Test, @BeforeEach, @AfterEach).

По итогам семинара №10 слушатель должен **уметь**:

* Написать базовые юнит-тесты для Spring-компонентов с использованием JUnit.
* Создавать mock-объекты с помощью Mockito и внедрять их в Spring-компоненты для тестирования.

1. **План Содержание:**

|  |  |  |
| --- | --- | --- |
| **Этап урока** | **Тайминг, минуты** | **Формат** |
| Введение, обзор темы | 20 | Модерирует преподаватель |
| Задание 1 | 40 | Студенты выполняют, преподаватель помогает в решении проблем |
| Задание 2 | 40 | Студенты выполняют, преподаватель помогает в решении проблем |
| Вопросы и обсуждение | 20 | Модерирует преподаватель |
| **Длительность:** | **120** |  |

### **Блок 1.**

Тайминг:

Объяснение правил – 10 минут

Работа в команде – 30 минут

Задание:

Разработайте юнит-тесты для сервиса, который управляет пользователями в системе. У вас есть `UserService` с методами `addUser(User user)`, `deleteUser(Long id)`, `findUserByEmail(String email)`. Данный сервис использует репозиторий `UserRepository` для взаимодействия с базой данных.

1. Создайте mock для `UserRepository`.

2. Напишите тест, который проверяет, что при добавлении пользователя метод `save` репозитория вызывается один раз.

3. Напишите тест, который проверяет, что при удалении пользователя метод `deleteById` репозитория вызывается с правильным `id`.

4. Напишите тест, который проверяет, что метод `findUserByEmail` возвращает правильного пользователя.

Пример решения:

@RunWith(SpringRunner.class)

@SpringBootTest

public class UserServiceTest {

@InjectMocks

private UserService userService;

@Mock

private UserRepository userRepository;

@BeforeEach

public void setup() {

MockitoAnnotations.openMocks(this);

}

@Test

public void testAddUser() {

User user = new User();

userService.addUser(user);

Mockito.verify(userRepository, Mockito.times(1)).save(user);

}

@Test

public void testDeleteUser() {

Long id = 1L;

userService.deleteUser(id);

Mockito.verify(userRepository, Mockito.times(1)).deleteById(id);

}

@Test

public void testFindUserByEmail() {

User mockUser = new User();

mockUser.setEmail("test@example.com");

Mockito.when(userRepository.findByEmail("test@example.com")).thenReturn(mockUser);

User foundUser = userService.findUserByEmail("test@example.com");

assertEquals("test@example.com", foundUser.getEmail());

}

}

Часто встречающиеся ошибки:

1. Не инициализировать mock-объекты перед запуском теста, что приводит к `NullPointerException`.

2. Не указать ожидаемое поведение для mock-объекта с помощью `Mockito.when(...)`. Это может привести к тому, что mock-объект всегда будет возвращать `null`.

3. Использовать реальный репозиторий вместо mock, что может привести к изменениям в реальной базе данных или неожиданным результатам теста.

4. Не проверять все возможные сценарии или пограничные случаи. Например, что происходит, если передать `null` в метод сервиса.

### **Блок 2.**

Тайминг:

Объяснение правил – 10 минут

Работа в команде – 30 минут

Задание:

Вашей задачей является написание юнит-тестов для сервиса заказов `OrderService`. Этот сервис имеет методы `placeOrder(Order order)`, `cancelOrder(Long orderId)` и `getOrderTotal(Long orderId)`. Для управления данными сервис использует репозиторий `OrderRepository` и взаимодействует с `ProductService` для получения стоимости товара.

1. Создайте mock-объекты для `OrderRepository` и `ProductService`.

2. Напишите тест, который проверяет, что при размещении заказа метод `save` репозитория вызывается один раз.

3. Напишите тест, который проверяет, что при отмене заказа метод `deleteById` репозитория вызывается с правильным `orderId`.

4. Напишите тест, который проверяет корректное вычисление общей стоимости заказа на основе стоимости товаров из `ProductService`.

Пример решения:

@RunWith(SpringRunner.class)

@SpringBootTest

public class OrderServiceTest {

@InjectMocks

private OrderService orderService;

@Mock

private OrderRepository orderRepository;

@Mock

private ProductService productService;

@BeforeEach

public void setup() {

MockitoAnnotations.openMocks(this);

}

@Test

public void testPlaceOrder() {

Order order = new Order();

orderService.placeOrder(order);

Mockito.verify(orderRepository, Mockito.times(1)).save(order);

}

@Test

public void testCancelOrder() {

Long orderId = 1L;

orderService.cancelOrder(orderId);

Mockito.verify(orderRepository, Mockito.times(1)).deleteById(orderId);

}

@Test

public void testGetOrderTotal() {

Order order = new Order();

order.setId(1L);

order.addProduct(new Product(1L), 2); // 2 items of product with ID=1

Mockito.when(productService.getProductPrice(1L)).thenReturn(100.0); // Each product costs 100

Mockito.when(orderRepository.findById(1L)).thenReturn(Optional.of(order));

double total = orderService.getOrderTotal(1L);

assertEquals(200.0, total, 0.001);

}

}

Часто встречающиеся ошибки:

1. Забыть инициализировать mock-объекты, что приводит к `NullPointerException`.

2. Не настроить ожидаемое поведение для mock-объектов с помощью `Mockito.when(...)`. В результате mock-объект будет всегда возвращать `null`.

3. Использование реальных компонентов вместо mock-объектов, что может привести к непредсказуемым результатам теста.

4. Не учитывать пограничные условия или исключительные ситуации, такие как отсутствующий заказ или продукт.

1. **Домашнее задание**

Разработайте тесты для службы аутентификации `AuthService`. Этот сервис имеет методы `login(String username, String password)`, `register(User user)` и `logout(Long userId)`. Служба использует `UserRepository` для управления данными пользователя и `SessionRepository` для управления сессионными данными.

1. Создайте mock-объекты для `UserRepository` и `SessionRepository`.

2. Напишите тест, который проверяет, что при успешной регистрации метод `save` репозитория пользователя вызывается.

3. Напишите тест, который проверяет, что при входе в систему для существующего пользователя создается новая сессия.

4. Напишите тест, проверяющий корректное завершение сессии при выходе из системы.

Пример решения:

@RunWith(SpringRunner.class)

@SpringBootTest

public class AuthServiceTest {

@InjectMocks

private AuthService authService;

@Mock

private UserRepository userRepository;

@Mock

private SessionRepository sessionRepository;

@BeforeEach

public void setup() {

MockitoAnnotations.openMocks(this);

}

@Test

public void testRegister() {

User user = new User();

authService.register(user);

Mockito.verify(userRepository, Mockito.times(1)).save(user);

}

@Test

public void testLogin() {

User user = new User();

user.setUsername("test");

user.setPassword("password");

Mockito.when(userRepository.findByUsername("test")).thenReturn(user);

authService.login("test", "password");

Mockito.verify(sessionRepository, Mockito.times(1)).createNewSession(Mockito.any());

}

@Test

public void testLogout() {

Long userId = 1L;

authService.logout(userId);

Mockito.verify(sessionRepository, Mockito.times(1)).terminateSession(userId);

}

}

Рекомендации для преподавателей по оценке задания:

1. Понимание задачи: Убедитесь, что студент понимает различие между юнит-тестированием и интеграционным тестированием, и в этом задании фокусируется на юнит-тестировании.

2. Использование mock-объектов: Проверьте, правильно ли студент создал и использовал mock-объекты для репозиториев.

3. Качество тестов: Убедитесь, что тесты написаны таким образом, чтобы действительно проверять ожидаемое поведение службы. Тесты должны быть независимыми и возможными для повторного использования.

4. Обработка исключений: Рассмотрите возможные исключительные ситуации (например, вход с неправильными учетными данными) и убедитесь, что студент также написал тесты для этих сценариев.

5. Чистота кода: Код тестов должен быть читаемым и хорошо структурированным.