## J14 Тестирование

## Unit тесты – где проблема (Junit, TestNG)

## Интеграционные тесты – проблемы взаимодествия. (@SpringBootTest)

## Smoke тесты – критические ошибки

## Unit тестирование модульное тестирование направлено на тестирование отдельных модулей, которые изолированы от других модулей и компонентов.

## тестируются только общедоступные методы или интерфейсы, а внутренний код тестируется за счет изменения входных данных;

## для получения данных, требуемых тестируемой логике, должны создаваться фиктивные зависимости.

**JUnit** — библиотека для модульного тестирования программ Java.

**JUnit** – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов.

Mockito — фреймворк для работы с заглушками.

**Запустить тесты**. С помощью **java -cp org.junit.runner.JUnitCore [имена тест-классов]** можно запускать классы целиком. Для запуска отдельных тестов нужно писать свой раннер, но сборщики, например gradle предоставляют свои. Иногда может понадобиться указать в classpath все используемые jar-файлы.

**./gradlew test**

@InjectMocks - создает экземпляр класса и внедряет макеты, созданные с помощью аннотаций @Mock(или @Spy), в этот экземпляр.

**@Mock** – в любом методе или конструкторе где есть параметр этого типа будет подставлен этот мок.

Сервис мокается чтобы на него писать стаб

**@Spy** – работает с реальным объектом (например отслеживать вызовы, сколько раз отработал)

Мокать – то что собираюсь тестить (сервис), сами обращения к сторонним сервисам стабаем.

* [mock()](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#mock-java.lang.Class-)/ [@Mock](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mock.html): создать макет
  + опционально указать, как он должен вести себя через [Answer](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/stubbing/Answer.html)/[MockSettings](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/MockSettings.html)
  + [when()](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#when-T-)/ [given()](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/BDDMockito.html" \l "given-T-)чтобы указать, как должен вести себя макет
  + Если предоставленные ответы не соответствуют вашим потребностям, напишите их самостоятельно, расширив [Answer](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/stubbing/Answer.html)интерфейс .
* [spy()](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#spy-T-)/ [@Spy](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Spy.html): частичная имитация, реальные методы вызываются, но их все еще можно проверить и заглушить
* [@InjectMocks](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/InjectMocks.html): автоматически вводить поля mocks/spy, аннотированные @Spyили@Mock
* [verify()](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#verify-T-): чтобы проверить вызовы методов с заданными аргументами
  + можно использовать гибкое сопоставление аргументов, например любое выражение через [any()](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/ArgumentMatchers.html#any--)
  + или зафиксируйте, какие аргументы были вызваны, используя [@Captor](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Captor.html) вместо этого
* Попробуйте синтаксис разработки, основанный на поведении, с помощью [BDDMockito](http://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/BDDMockito.html)
* Используйте Mockito на Android благодаря команде, работающей над [dexmaker.](https://github.com/crittercism/dexmaker)

Разница между аннотациями @Mock и @Spy:  
1. @Mock создает все фиктивные объекты. Перед наложением определенных методов все атрибуты и методы фиктивного объекта устанавливаются в значение null (0 или null); и В соответствии с аннотацией @Spy, @Spy может создавать частичные фиктивные объекты, и все методы-члены частичных фиктивных объектов будут выполняться в соответствии с логикой исходного метода, пока они не будут накапливаться для возврата определенного значения. @Mock и @Spy - две сопоставимые концепции.  
2. Метод mock () Mokcito имеет ту же функцию, что и @Mock, но его использование и сценарии другие. Точно так же @Spy также соответствует методу spy ().  
3. Аннотированные объекты @Mock и @Spy могут быть введены с помощью @InjectMock в обрабатываемый объект.

|  |  |
| --- | --- |
| Представление (запись) | Принцип работы (описание) |
| @Test public void testMethod() | Тестовый методы |
| @Test(timeout=100) public void testMethod() | Провал теста при превышении заданного «таймаута» |
| @Test (expected = MyException.class) public void testMethod() | Выбор исключения, относящегося к MyException, в противном случае — провал |
| @Ignore public void testMethod() | Игнорирование заданного тестового метода |
| @BeforeClass public static void testMethod() | Вызывается один раз для класса. Делается это перед тем, как выполнить проверку. Размещается инициализация, выполняемая всего лишь единожды. Пример – считывание информации, используемых в будущем в тестовых методах |
| @AfterClass public static void testMethod() | Вызывается после реализации проверки для заданного класса. Тут пишут то, что выполняется один раз. Пример – прекратить соединение с БД или удалить лишнюю информацию |
| @Before public static void beforeMethod() | Вызов перед каждой проверкой в пределах тестового класса. Чаще всего применяется при инициализации чего-либо |
| @After public static void afterMethod() | Схож с предыдущим вариантом. Обычно характеризует деинициализацию |

## Проверки Asserts и их типы

|  |  |
| --- | --- |
| Что за проверка | Характеристика |
| fail() fail(String message) | Завершение с ошибкой, провал проверки |
| assertTrue(boolean condition) assertTrue(java.lang.String message, boolean condition) | Проверка «кондишн» на равенство значения по true |
| assertFalse(boolean condition) assertFalse(String message, boolean condition) | Аналогично предыдущему варианту, но с false |
| assertEquals(<тип> expected, <тип> actual) assertEquals(String message, <тип> expected, <тип> actual) | Удостоверение равенства, где к типу относят Object, а также переменные разных видов |
| assertArrayEquals(byte[] expecteds, byte[] actuals) assertArrayEquals(String message, <тип>[] expecteds, <тип>[] actuals) | Равенство массивов |
| assertNotNull(Object object) assertNotNull(String message, Object object) | Удостоверение в том, что объект не null |
| assertNull(Object object) assertNull(String message, Object object) | Удостоверение в том, что заданный объект выступает в качестве пустого (null) |
| assertSame(Object expected, Object actual) assertSame(String message, Object expected, Object actual) | Сверка равенства двух элементов |

Mockito – фреймворк моков и стабов

Наибольшее распространение получили следующие возможности *Mockito* :

* создание заглушек для классов и интерфейсов;
* проверка вызыва метода и значений передаваемых методу параметров;
* использование концепции «частичной заглушки», при которой заглушка создается на класс с определением поведения, требуемое для некоторых методов класса;
* подключение к реальному классу «шпиона» *spy* для контроля вызова методов.

### 1. Определение поведения - when(mock).thenReturn(value)

Этот метод позволяет определить возвращаемое значение при вызове метода *mock* с заданными параметрами. Если будет указано более одного возвращаемого значения, то они будут возвращены методом последовательно, пока не вернётся последнее; после этого при последующих вызовах будет возвращаться только последнее значение. Таким образом, чтобы метод всегда возвращал одно и то же значение, седует просто определить одно условие.

### 1. Определение поведения - when(mock).thenReturn(value)

Этот метод позволяет определить возвращаемое значение при вызове метода *mock* с заданными параметрами. Если будет указано более одного возвращаемого значения, то они будут возвращены методом последовательно, пока не вернётся последнее; после этого при последующих вызовах будет возвращаться только последнее значение. Таким образом, чтобы метод всегда возвращал одно и то же значение, седует просто определить одно условие.

@RunWith(MockitoJUnitRunner.class)

public class Test\_Mockito

{

@Mock

ICalculator mcalc;

// используем аанотацию @InjectMocks для создания mock объекта

@InjectMocks

Calculator calc = new Calculator(mcalc);

@Test

public void testCalcAdd()

{

// определяем поведение калькулятора для операции сложения

when(calc.add(10.0, 20.0)).thenReturn(30.0);

// проверяем действие сложения

assertEquals(calc.add(10, 20), 30.0, 0);

// проверяем выполнение действия

verify(mcalc).add(10.0, 20.0);

// определение поведения с использованием doReturn

doReturn(15.0).when(mcalc).add(10.0, 5.0);

// проверяем действие сложения

assertEquals(calc.add(10.0, 5.0), 15.0, 0);

verify(mcalc).add(10.0, 5.0);

}

}

Метод *verify* позволяет проверить, была ли выполнена проверка с определенными параметрами. Если проверка не выполнялась или выполнялась с другими параметрами, то *verify* вызовет исключение.

### 2. Подсчет количества вызовов - atLeast, atLeastOnce, atMost, times, never

Для проверки количества вызовов определенных методов *Mockito* предоставляет следующие методы :

* **atLeast (int min)** - не меньше min вызовов;
* **atLeastOnce ()** - хотя бы один вызов;
* **atMost (int max)** - не более max вызовов;
* **times (int cnt)** - cnt вызовов;
* **never ()** - вызовов не было;

### 3. Обработка исключений - **when(mock).thenThrow()**

*Mockito* позволяет вызвать исключение при определенных условиях. Для этого необходимо использовать следующий синтаксис кода :

// создаем исключение

RuntimeException exception = new RuntimeException ("Division by zero");

// определение поведения для вызова исключения

doThrow(exception).when(mock).divide(5.0, 0));

В представленном коде создали [исключение](https://java-online.ru/java-throws.xhtml) RuntimeException. После этого определили условия вызова исключения - вызов метода деления на 0. В следующем тесте выполняется проверка метода divide. При делении на 0 вызывается исключение.

@Test

public void testDevide()

{

when(mcalc.divide(15.0, 3)).thenReturn(5.0);

assertEquals(calc.divide(15.0, 3), 5.0, 0);

// проверка вызова метода

verify(mcalc).divide(15.0, 3);

// создаем исключение

RuntimeException exception = new RuntimeException ("Division by zero");

// определяем поведение

doThrow(exception).when(mcalc).divide(15.0, 0);

assertEquals(calc.divide(15.0, 0), 0.0, 0);

verify(mcalc).divide(15.0, 0);

}

### 4. Использование интерфейса org.mockito.stubbing.Answer<T>

Иногда описание поведения *mock* объекта требует определенной проверки с усложнением логики. В этом случае можно использовать интерфейс **Answer<T>**, который позволяет реализовать заглушки методов со сложным поведением. В следующем тесте *testThenAnswer* при вызове метода сложения с определенными параметрами *mcalc.add(11.0, 12.0)* будет вызван метод *answer*, который подготовит ответ. Параметр *InvocationOnMock* позволяет получить информацию о вызываемом методе и параметрах.

// метод обработки ответа

private Answer<Double> answer = new Answer<Double>() {

@Override

public Double answer(InvocationOnMock invocation) throws Throwable

{

// получение объекта mock

Object mock = invocation.getMock();

System.out.println ("mock object : " + mock.toString());

// аргументы метода, переданные mock

Object[] args = invocation.getArguments();

double d1 = (double) args[0];

double d2 = (double) args[1];

double d3 = d1 + d2;

System.out.println ("" + d1 + " + " + d2);

return d3;

}

};

@Test

public void testThenAnswer()

{

// определение поведения mock для метода с параметрами

when(mcalc.add(11.0, 12.0)).thenAnswer(answer);

assertEquals(calc.add(11.0,12.0), 23.0, 0);

}

### 5. Использование шпиона spy на реальных объектах

При использовании шпиона реальное поведение объекта сохраняется.

*Mockito* позволяет подключать к реальным объектам «шпиона» **spy**, **контролировать** **возвращаемые** методами **значения** и отслеживать **количество вызовов** методов. В следующем тесте создадим шпиона *scalc*, который подключим к реальному калькулятору и будем вызывать метод *double15()*. Необходимо отметить, что метод реального объекта *double15* должен вернуть значение 15. Однако *Mockito* позволяет переопределить значение и согласно вновь назначенному условию новое значение должно быть 23.

@Test

public void testSpy()

{

Calculator scalc = spy(new Calculator());

when(scalc.double15()).thenReturn(23.0);

// вызов метода реального класса

scalc.double15();

// проверка вызова метода

verify(scalc).double15();

// проверка возвращаемого методом значения

assertEquals(23.0, scalc.double15(), 0);

// проверка вызова метода не менее 2-х раз

verify(scalc, atLeast(2)).double15();

}

В результате выполнения теста видим, что метод возвращает значение 23. Таким образом, фреймворк **Mockito** в сочетании с JUnit можно использовать для тестов реального класса. При этом, можно проверить, вызывался ли метод, сколько раз и с какими параметрами. Кроме этого, можно создавать заглушки только для некоторых методов. Это позволяет проверить поведение одних методов, используя заглушки для других.

### 6. Проверка вызова метода с задержкой, timeout

Фреймворк *Mockito* позволяет выполнить проверку вызова определенного метода в течение заданного в **timeout** времени. Задержка времени определяется в милисекундах.

@Test

public void testTimout()

{

// определение результирующего значения mock для метода

when(mcalc.add(11.0, 12.0)).thenReturn(23.0);

// проверка значения

assertEquals(calc.add(11.0,12.0), 23.0, 0);

// проверка вызова метода в течение 10 мс

verify(mcalc, timeout(100)).add(11.0, 12.0);

}

### 7. Использование в тестах java классов

В следующем тесте при создании *mock* объектов используются java классы Iterator и Comparable. После этого определяются условия проверок и выполняются тесты.

@Test

public void testJavaClasses()

{

// создание объекта mock

Iterator<String> mis = mock(Iterator.class);

// формирование ответов

when(mis.next()).thenReturn("Привет").thenReturn("Mockito");

// формируем строку из ответов

String result = mis.next() + ", " + mis.next();

// проверяем

assertEquals("Привет, Mockito", result);

Comparable<String> mcs = mock(Comparable.class);

when(mcs.compareTo("Mockito")).thenReturn(1);

assertEquals(1, mcs.compareTo("Mockito"));

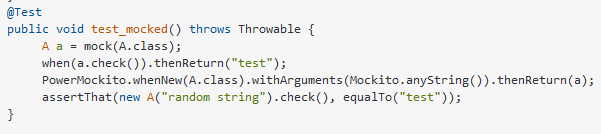
Comparable<Integer> mci = mock(Comparable.class);

when(mci.compareTo(anyInt())).thenReturn(1);

assertEquals(1, mci.compareTo(5));

}

Можно **Замокать конструктор,** с помощью библиотеки **PowerMockito**



## Интеграционное тестирование

* Реальный объект
* Нормальное взаимодействие между сервисом
* Генерятся тестовые данные
* Сервис – сервис – результат

Стратегия интеграционного тестирования основывается на проверке прикладного кода в окружении, близком к фактическому окружению, но не являющимся им. Главная цель данной стратегии – убедиться в правильности взаимодействия кода с внешними ресурсами и взаимодействия различных технологий в приложении между собой.

То есть у нас запускается весь контекст приложения.

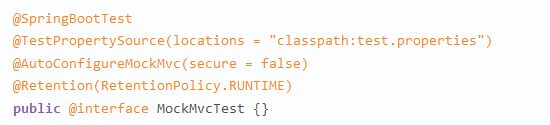
В интеграционных тестах часто используется тестовая база данных, что дает возможность проверить сущности jpa.

**Spring Mock-MVC**

предназначен для тестирования контроллеров. Позволяет убедиться, что все части работают вместе, от контроллера до базы данных.

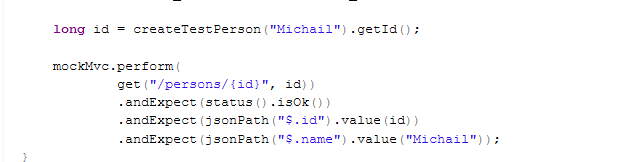
Он позволяет тестировать контроллеры без запуска http-сервера. То есть при выполнении тестов сетевое соединение не создается.

Spring позволяет нам *объединять несколько аннотаций Spring* в одну настраиваемую аннотацию.

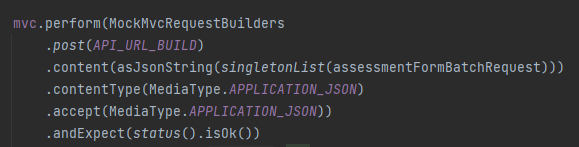


Теперь мы можем использовать эту аннотацию на всех своих тестах.

GET



PUT или POST



Rules:

## ****Timeout Rule****

## ****ExpectedException****