**Yêu cầu**

Trước khi tìm hiểu Mockito, bạn cần biết JUnit.

Trong bài viết có sử dụng [Lombok](https://loda.me/general-huong-dan-su-dung-lombok-giup-code-java-nhanh-hon-69-loda1552789752787)

**Giới thiệu**

Về cơ bản, Unit test là phương pháp tiếp cận độc lập, tức là mỗi một chức chăng năng sẽ được đi kèm với một hoặc nhiều bài test để chắc chắn ràng cái chức năng đó hoạt động ổn. Vậy nên Unit Test thường là đơn nhỏ nhất và test case cũng dễ dàng khởi tạo

@Test

public void testSum(){

Assert.assertEquals(3, MathUtil.sum(1,2));

}

Bài toán mở rộng ra hơn, Khi chúng ta phải test sự hoạt động phối hợp giữa nhiều chức năng hay thành phần với nhau hoặc muốn test một chức năng lớn thì sẽ trở nên phức tạp và khó xây dựng hơn rất nhiều.

Các kịch bản sau thường diễn ra:

* Chức năng A gọi tới chức năng B. tuy nhiên, B chưa viết xong
* Chức năng A gọi tới chức năng B. tuy nhiên, B khởi tạo rất phức tạp (truy xuất Database, yêu cầu nhiều params, v.v.)
* Muốn test chức năng A, tuy nhiên A yêu cầu nỗ lực lớn của cả hệ thống (Yêu cầu có HTTP-server, authorize, v.v.)

// Giả sử muốn test hàm này

public int getAllData(){

// Giả sử như thằng driver không được khởi tạo hoặc bị lỗi

// thì function này coi như hỏng

MySQLdriver.connect() // Kết nối xuống Database

var data = MySQLdriver.get()// Lấy dữ liệu

// Chúng ta muốn test logic xử lý dữ liệu ở dưới đây,

// mà k cần kết nối databse, phải làm sao?

// Xử lý dữ liệu

...

// trả ra dữ liệu

return data

}

Rất nhiều kịch bản phức tạp xảy ra, mà trên thực tế, chúng ta chỉ mong muốn confirm rằng A ổn, chứ thằng B, C, D gì đó thì hãy cứ tạm coi là nó "đã ổn" đi.

Để đơn giản hoá các kịch bản test như trên, khái niệm Mock ra đời.

Tư tưởng của Mock đơn giản là khi muốn test (A gọi B) thì thay vì tạo ra một đối tượng B thật sự, bạn tạo ra một thằng B' giả mạo, có đầy đủ chức năng như B thật (nhưng không phải thật)

Bạn sẽ giả lập cho B' biết là khi thằng A gọi tới nó, nó cần làm gì, trả lại cái gì (hardcode). Miễn làm sao cho nó trả ra đúng cái thằng A cần để chúng ta có thể test A thuận lợi nhất.

// Đại loại như này

// Khi driver.get() được gọi, hãy trả ra một List(1,2,3)

Mockito.when(driver.get())

.thenReturn(Arrays.asList(1, 2, 3));

Ở trong Java, Mockito chính là thư viện nổi tiếng nhất để các bạn làm việc này.

**Cài đặt**

Để sử dụng Mockito bạn cần:

<!--https://mvnrepository.com/artifact/org.mockito/mockito-core -->

<dependency><groupId>org.mockito</groupId><artifactId>mockito-core</artifactId><version>3.2.4</version><scope>test</scope></dependency>

Chúng ta đang viết test, nên đừng quên JUnit nữa nhé:

<!--https://mvnrepository.com/artifact/junit/junit -->

<dependency><groupId>junit</groupId><artifactId>junit</artifactId><version>4.12</version><scope>test</scope></dependency>

**Cách Mock**

Khái niệm cơ bản đầu tiên, đó là làm sao để tạo ra một đối tượng bị Mock.

**Cách 1: Mockito.mock()**

Sử dụng Mockito.mock() để tạo ra một object của Class bạn yêu cầu, nó thậm chí còn không quan tâm hàm khởi tạo của Class ý như nào và ra sao, vì nó đâu có thật :v

@Test

public void testUserMockFunction() {

List mockList = Mockito.mock(List.class);

Mockito.when(mockList.size()).thenReturn(2);

Assert.assertEquals(2, mockList.size());

}

**Cách 2: Sử dụng @Mock**

Bạn muốn mock cái gì, thì đánh dấu @Mock lên ý, đơn giản vãi :v

@Mock

List<String> mockedList;

Tuy nhiên, gắn @Mock là chưa đủ, bạn cần kích hoạt Mockito để nó mock các object ý cho bạn.

Sau khi kích hoạt, thì tất cả các object được gắn @Mock sẽ trở thành 1 object giả mạo, và đã được khởi tạo (tức là != null)

Các cách kích hoạt như sau:

1. Gắn @RunWith(MockitoJUnitRunner.class) lên class test của bạn

// Cách 1

@RunWith(MockitoJUnitRunner.class)

public class MockitoAnnotationTest {

@Mock

List<String> mockedList;

}

1. tạo ra một đối tượng MockitoRule bên trong class test của bạn

public class MockitoAnnotationTest {

// Cách 2

@Rule

public MockitoRule initRule = MockitoJUnit.rule();

@Mock

List<String> mockedList;

}

1. Sử dụng Mockito.init()

public class MockitoAnnotationTest {

@Mock

List<String> mockedList;

@Before

public void setUp() throws Exception {

// Cách 3

// Nếu bạn không dùng cách 1 hoặc 2 thì phải sử dụng

// dòng code dưới đây:

MockitoAnnotations.initMocks(this);

}

}

Vậy là xong, bạn đã tạo ra được một Object bị mock.

**Cách sử dụng**

Mockito cung cấp rất nhiều methods khác nhau để giúp bạn giả lập đầy đủ các thứ bạn cần.

Hay sử dụng nhất là hàm when().

// Trả là size 100 khi gọi hàm size()

Mockito.when(mockedList.size()).thenReturn(100);

// Throw exception khi gọi hàm size()

Mockito.when(mockedList.size()).thenThrow(NullPointerException.class);

// Bạn có thể đổi ngược cách viết, còn logic vẫn vậy

// Ném ra exception khi gọi hàm .get() với tham số truyền vào bất kì

Mockito.doThrow(NullPointerException.class).when(mockedList.get(Mockito.any()));

**Spy**

Spy là việc sửa một đối tượng Thật -> Giả

@RunWith(MockitoJUnitRunner.class)

public class SpyTest {

@Spy

List<String> list = new ArrayList<>();

@Test

public void testSpy() {

list.add("one");

list.add("two");

// show the list items

System.out.println(list);

Mockito.verify(list).add("one");

Mockito.verify(list).add("two");

// @Spy thực sự gọi hàm .add của List nên nó có size là 2 mà không cần giả lập

Assert.assertEquals(2, list.size());

// Vẫn có thể làm giả thông tin gọi hàm với @Spy

Mockito.when(list.size()).thenReturn(100);

Assert.assertEquals(100, list.size());

}

}

**Captor**

Captor có nhiệm vụ ghi lại các tham số của đối tượng

@RunWith(MockitoJUnitRunner.class)

public class CaptorTest {

@Mock

List<Object> list;

@Captor

ArgumentCaptor<Object> captor;

@Test

public void testCaptor1() {

list.add(1);

// Capture lần gọi add vừa rồi có giá trị là gì

Mockito.verify(list).add(captor.capture());

System.out.println(captor.getAllValues());

Assert.assertEquals(1, captor.getValue());

}

}

**Inject Mock**

Quay lại với ví dụ đầu bài viết:

Tôi có một lớp Service chứa lớp DatabaseDriver như sau:

public interface DatabaseDriver {

List<Object> get() throws SQLException;

}

@Data

@AllArgsConstructor

public static class SuperService {

DatabaseDriver driver;

public List<Object> getObjects() throws SQLException {

System.out.println("LOG: Getting objects");

List<Object> list = driver.get();

System.out.println("LOG: Sorting");

Collections.sort(list, Comparator.comparingInt(value -> Integer.valueOf(value.toString())));

System.out.println("LOG: Done");

return list;

}

}

Rõ ràng thì driver chính là mắt xích trong trong việc SuperService có hoạt động được hay không. Vì thế, muốn test được SuperService, chúng ta phải mock đối tượng driver.

kịch bản bây giờ sẽ là tôi mock một đối tượng của DatabaseDriver rồi truyền nó vào đối tượng SuperService.

Để truyền một đối tượng mock vào một đối tượng khác, chúng ta dùng @InjectMocks.

@RunWith(MockitoJUnitRunner.class)

public class InjectMockTest {

@Mock

DatabaseDriver driver;

/\*\*

\* Inject driver vào superService.

\* Mọi người có thể liên tưởng nó giống với Spring Injection

\*/

@InjectMocks

SuperService superService;

@Test(expected = SQLException.class)

public void testInjectMock() throws SQLException {

// Giả lập cho driver luôn trả về list (3,2,1) khi được gọi tới

Mockito.doReturn(Arrays.asList(3, 2, 1)).when(driver).get();

Assert.assertEquals(driver, superService.getDriver());

// Test xem superService trả ra ngoài có đúng không?

Assert.assertEquals(Arrays.asList(1, 2, 3), superService.getObjects());

// Giả lập cho driver bắn exception

Mockito.when(driver.get()).thenThrow(SQLException.class);

superService.getObjects();

}

}

**Kết**

Tới đây, bạn đã có thể dùng Mockito xoành xoạch rồi.

Tiếp theo, có thể đọc cách sử dụng Mockito với Spring Boot [tại đây](https://loda.me/spring-boot-0-series-lam-chu-spring-boot-zero-to-hero-loda1558963914472)