Unit Testing –

Mocking → Used to replace external dependencies.

Stubbing → fake classes that allow you to mimic the real database. They return the data values.

#1

Let's setup a new project. Add Spring web & DevTools in dependency.

Create a new Business package, a new class inside that, name it SumBusinessImpl.

```
public int calculateSum(int[] data
    int sum = 0;
    for(int value:data) {
        sum += value;
    }
    return sum;
}
```

Let's write our first Unit Test for this business logic.

→ In our Test folder, create a new class SomeBusinessTest.

```
@Test
public void calculateSum_basic() {
    SomeBusinessImpl business = new SomeBusinessImpl();
    int actualResult = business.calculateSum(new int[] { 1,2,3});
    int expectedResult = 6;
    assertEquals(expectedResult, actualResult);
}
```

Now run the test.

→ Okay, in our business logic, we are directly passing the data, but in real world we will use some data service in order to get data.

↓

Dependency.

→ Let's make another package data, and make a new class SomeDataService.

```
public interface SomeDataService {

    int[] retrieveAllData();

}
```

Also, changing in SomeDataService.

```
public int calculateSumUsingDataService() {
    int sum = 0;
    int[] data = someDataService.retrieveAllData();
    for(int value:data) {
        sum += value;
    }
```

return sum;

}

and add on top →

```
private SomeDataService someDataService;

public void setSomeDataService(SomeDataService so
    this.someDataService = someDataService;
}
```

← setter

Now let's write a unit test for this.

Copy the test class and rename it to SomeBusinessStubTest.java →

```
@Test
public void calculateSumUsingDataService_basic() {
    SomeBusinessImpl business = new SomeBusinessImpl();
    business.setSomeDataService(someDataService);
    int actualResult = business.calculateSumUsingDataService();
    int expectedResult = 6;
    assertEquals(expectedResult, actualResult);
}
```

For this dependency, we need to have a STUB implementation.

```
class SomeDataServiceStub implements SomeDataService {

    @Override
    public int[] retrieveAllData() {
        return new int[] {1,2,3};
    }

}
```


and change →

```
business.setSomeDataService(new SomeDataServiceStub());
```

Problem with this method is that as the test cases grow, its difficult to maintain all these stubs.

The solution is mocking.

So, copy the stub test class and name it SomeBusinessMockTest

```
@Test
public void calculateSumUsingDataService_basic() {
    SomeBusinessImpl business = new SomeBusinessImpl();

    SomeDataService dataServiceMock = mock(SomeDataService.class);
    when(dataServiceMock.retrieveAllData()).thenReturn(new int[] { 1, 2, 3 });

    business.setSomeDataService(dataServiceMock);

    int actualResult = business.calculateSumUsingDataService();
    int expectedResult = 6;
    assertEquals(expectedResult, actualResult);
}
```

Now, these two lines are common for all the test cases, hence can be made global. And before every test, we have set our data service.

Let's move them out of our tests.

```
public class SomeBusinessMockTest {

    SomeBusinessImpl business = new SomeBusinessImpl();
    SomeDataService dataServiceMock = mock(SomeDataService.class);

    @Before
    public void before() {
        business.setSomeDataService(dataServiceMock);
    }
```

Let's optimise more. So instead of using reflection what we can do is use mockito annotations.

```
@RunWith(MockitoJUnitRunner.class)
public class SomeBusinessMockTest {

    @InjectMocks
    SomeBusinessImpl business;

    @Mock
    SomeDataService dataServiceMock;

    @Test
    public void calculateSumUsingDataService_basic() {
        when(dataServiceMock.retrieveAllData()).thenReturn(new in
        assertEquals(6, business.calculateSumUsingDataService());
    }
```


Also, with this way we do not need the setter.

Now test it.!!!

Let's move ahead, create a new class

ListMockTest

```
@Test
public void size_basic() {
    List mock = mock(List.class);
    when(mock.size()).thenReturn(5);
    assertEquals(5,mock.size());
}

@Test
public void returnDifferentValues() {
    List mock = mock(List.class);
    when(mock.size()).thenReturn(5).thenReturn(10);
    assertEquals(5,mock.size());
    assertEquals(10,mock.size());
}
```


↑ can return list of results.

Let's move List.mock at top as it is common.

```
List mock = mock(List.class);
```

We can also return parameter in our test,

```
@Test
public void returnWithParameters() {
    when(mock.get(0)).thenReturn("in28Minutes");
    assertEquals("in28Minutes",mock.get(0));
    assertEquals(null,mock.get(1));
}
```

```
@Test
public void returnWithGenericParameters() {
    when(mock.get(anyInt())).thenReturn("in28Minutes");

    assertEquals("in28Minutes",mock.get(0));
    assertEquals("in28Minutes",mock.get(1));
}
```

→ Verification :

```
@Test
public void verificationBasics() {
    //SUT
    String value1 = mock.get(0);
    String value2 = mock.get(1);

    //Verify
    verify(mock).get(0);
    verify(mock).get(anyInt());
    verify(mock, times(1)).get(anyInt());

}
```

This will fail as it should because tested once.

Change it to -1

```
//Verify
verify(mock).get(0);
verify(mock, times(2)).get(anyInt());
verify(mock, atLeast(1)).get(anyInt())
verify(mock, atLeastOnce()).get(anyInt
verify(mock, atMost(2)).get(anyInt());
verify(mock, never()).get(2);
```

Verifying how a specific mock is called in the test. This is important if nothing is returned instead values are stored and a specific method is called with a specific value.

→ Now, Let's capture the argument and verify the same.

You can verify what is the value passed to the mock.

```java
@Test
public void argumentCapturing() {

    //SUT
    mock.add("SomeString");

    //Verification
    ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);
    verify(mock).add(captor.capture());

    assertEquals("SomeString", captor.getValue());

}
```

Let's verify captor for multiple calls.

```java
@Test
public void multipleArgumentCapturing() {

    //SUT
    mock.add("SomeString1");
    mock.add("SomeString2");

    //Verification
    ArgumentCaptor<String> captor = ArgumentCaptor.forClass(String.class);

    verify(mock, times(2)).add(captor.capture());

    List<String> allValues = captor.getAllValues();

    assertEquals("SomeString1", allValues.get(0));
    assertEquals("SomeString2", allValues.get(1));

}
```

A mock does not retain the behaviour of the methods until it is stubbed i.e. until it is overridden.

mocking

```
@Test
public void spying() {
    ArrayList arrayListMock = mock(ArrayList.class);
    System.out.println(arrayListMock.get(0));//null
    System.out.println(arrayListMock.size());//0
    arrayListMock.add("Test");
    arrayListMock.add("Test2");
    System.out.println(arrayListMock.size());//0
    when(arrayListMock.size()).thenReturn(5);
    System.out.println(arrayListMock.size());//5
}
```

Even on adding it won't change changes are after stubbed

But a spy retains the behavior of the class.

```
public void spying() {
    ArrayList arrayListSpy = spy(ArrayList.class);
    arrayListSpy.add("Test0");
    System.out.println(arrayListSpy.get(0));//Test0
    System.out.println(arrayListSpy.size());//1
    arrayListSpy.add("Test");
    arrayListSpy.add("Test2");
    System.out.println(arrayListSpy.size());//3

    when(arrayListSpy.size()).thenReturn(5);
    System.out.println(arrayListSpy.size());//5

    arrayListSpy.add("Test4");
    System.out.println(arrayListSpy.size());//5
}
```

spying will override the value

But when stubbed, it will take control from spy and will always return the stubbed value.

#) Testing in web Environment

Let's write a basic test without any dependency overhead.

First, lets make a controller.

↑ inside this package.

HelloWorldController.java | HelloWorldControllerTest.java

```
1  package com.in28minutes.unittesting.unittesting.controller;
2
3  import org.springframework.web.bind.annotation.GetMapping;
5
6  @RestController
7  public class HelloWorldController {
8
9      @GetMapping("/hello-world")
10     public String helloWorld() {
11         return "Hello World";
12     }
13
14 }
15
```

→ Lets write a test for this controller.

WorldController.java | *HelloWorldControllerTest.java

```
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

@RunWith(SpringRunner.class)
@WebMvcTest(HelloWorldController.class)
public class HelloWorldControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void helloWorld_basic() throws Exception {
        //call GET "/hello-world"  application/json
        RequestBuilder request = MockMvcRequestBuilders
                .get("/hello-world")
                .accept(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(request).andReturn();

        //verify "Hello World"
        assertEquals("Hello World", result.getResponse().getContentAsString());

    }

}
```


We are sending a GET request and doing a verification.

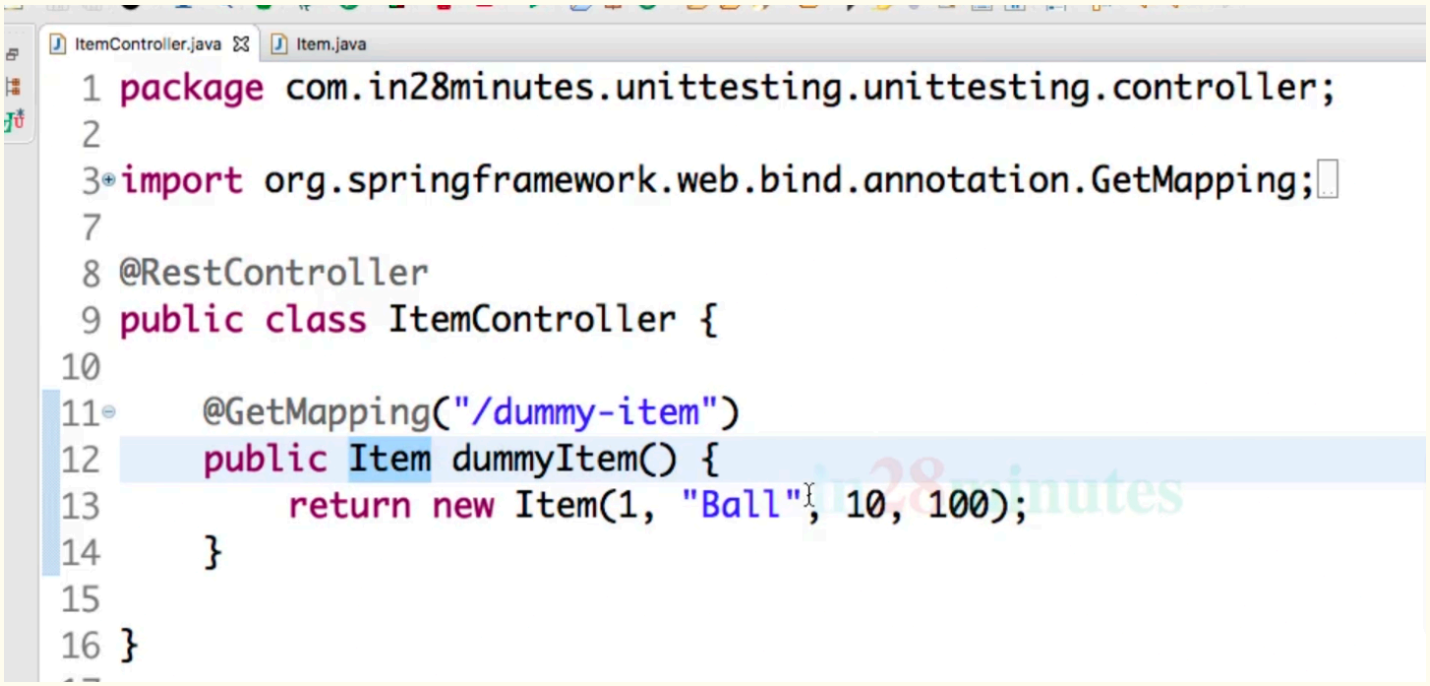
Let's change the way of verifying a bit.

```
        MvcResult result = mockMvc.perform(request)
                .andExpect(status().isOk())
                .andExpect(content().string("Hello World"))
                .andReturn();

        //verify "Hello World"
        //assertEquals("Hello World", result.getResponse().getContentAsString());
    }
```

→ Now let's make another controller which is a bit more complex.

So in the same controller package, make another class :-



One more class for item →

make getters, setters, toString methods.

*ItemController.java *Item.java

```java
package com.in28minutes.unittesting.unittesting.model;

public class Item {

    private int id;
    private String name;
    private int price;
    private int quantity;

    public Item(int id, String name, int price, int quantity) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

}
```

Now, let's write a test for that.

ItemController.java Item.java ItemControllerTest.java

```java
public class ItemControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void dummyItem_basic() throws Exception {

        RequestBuilder request = MockMvcRequestBuilders
                .get("/dummy-item")
                .accept(MediaType.APPLICATION_JSON);

        MvcResult result = mockMvc.perform(request)
                .andExpect(status().isOk())
                .andExpect(content().json("{\"id\": 1,\"name\":\"Ball\",\"price\":1
                .andReturn();
    }

}
```

this uses JSON Assert that does a not strict checking.

Now let's add another endpoint to our item controller.

```
@Autowired
private ItemBusinessService businessService;
```

```
@GetMapping("/item-from-business-service")
public Item itemFromBusinessService() {
    return businessService.retreiveHardcodedItem();
}
```

Let's make ItemBusinessService a class in business package

ItemController.java | ItemBusinessService.java

```
1  package com.in28minutes.unittesting.unittesting.business;
2
3  import org.springframework.stereotype.Component;
4
5  import com.in28minutes.unittesting.unittesting.model.Item;
6
7  @Component
8  public class ItemBusinessService {
9
10     public Item retreiveHardcodedItem() {
11         return new Item(1, "Ball", 10, 100);
12     }
13
14 }
```

Now let's write unit test for our REST with business service.

In the ItemControllerTest.java

```
@Test
public void itemFromBusinessService_basic() throws Exception {

    when(businessService.retreiveHardcodedItem()).thenReturn(
            new Item(2,"Item2",10,10));

    RequestBuilder request = MockMvcRequestBuilders
            .get("/item-from-business-service")
            .accept(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(request)
            .andExpect(status().isOk())
            .andExpect(content().json("{id:2,name:Item2,price:10}"))
            .andReturn();
    //JSONAssert.assertEquals(expected, result.getResponse().getConter

}
```

Now let's add H2 database & JPA to interact with DB while writing tests.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Also, convert the Item class to a JPA entity.

Use @Entity on top, make @Id & make a transient value.

↳ never stored to the DB.

```
@Entity
public class Item {

    @Id
    private int id;
    private String name;
    private int price;
    private int quantity;

    @Transient
    private int value;
}
```

Generate getters & setters.

- Application.properties →

```
spring.jpa.show-sql=true
spring.h2.console.enabled=true
```

Now in ItemBusinessService,

```
public List<Item> retrieveAllItems() {
    return repository.findAll();
}
```

In the controller,

```
@GetMapping("/all-items-from-database")
public List<Item> retrieveAllItems() {
    return businessService.retrieveAllItems();
}
```

Adding some business logic in BusinessService,

```
public List<Item> retrieveAllItems() {
    List<Item> items = repository.findAll();

    for(Item item:items) {
        item.setValue(item.getPrice() * item.getQuantity());
    }

    return items;
}
```

Now, let's write a test for the same.

So, we just need to add the stubbing part, and it should work fine-

```
@Test
public void retrieveAllItems_basic() throws Exception {
    when(businessService.retrieveAllItems()).thenReturn(
            Arrays.asList(new Item(2,"Item2",10,10),
                    new Item(3,"Item3",20,20))
            );

    RequestBuilder request = MockMvcRequestBuilders
            .get("/all-items-from-database")
            .accept(MediaType.APPLICATION_JSON);

    MvcResult result = mockMvc.perform(request)
            .andExpect(status().isOk())
            .andExpect(content().json("[{id:2,name:Item2,price:10},{id:3,name:I
            .andReturn();
    //JSONAssert.assertEquals(expected, result.getResponse().getContentAsString
```

Now let's write a unit test for our Business logic i.e. ItemBusinessService.

→ Copy SomeBusinessMockTest, and we will change that.

Name it ItemBusinessServiceTest

```
Rerun)
nWith(MockitoJUnitRunner.class)
lic class ItemBusinessServiceTest {

  @InjectMocks
  private ItemBusinessService business;

  @Mock
  private ItemRepository repository;

  @Test
  public void calculateSumUsingDataService_basic() {
      when(repository.findAll()).thenReturn(Arrays.asList(new Item(2,"Item2",10,10),
              new Item(3,"Item3",20,20)));
      List<Item> items = business.retrieveAllItems();

      assertEquals(100, items.get(0).getValue());
      assertEquals(400, items.get(1).getValue());

  }
```

Delete the above two tests.

→ Now lets write unit tests for Data layer.

not required for a JPA repository as not code in it, but for a Hibernate layer it will be beneficial.

Make a new package name data in unittesting.

BusinessServiceTest.java | ItemRepositoryTest.java | data.sql

```
import org.springframework.beans.factory.annotation.Autowire
import org.springframework.boot.test.autoconfigure.orm.jpa.D
import org.springframework.test.context.junit4.SpringRunner;

import com.in28minutes.unittesting.unittesting.model.Item;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ItemRepositoryTest {

    @Autowired
    private ItemRepository repository;

    @Test
    public void testFindAll() {
        List<Item> items = repository.findAll();
        assertEquals(3,items.size());
    }

}
```


In data.sql we have 3 rows.

→ Integration Tests (IT)

Let's make a class for IT.

→ Imp :- For data layer, instead of using @Autowired, we can use @MockBean, that will keep our test independent of data in DBMS.

→ Tests for other request methods

```
RequestBuilder requestBuilder = MockMvcRequestBuilders.post(
        "/items")
        .accept(MediaType.APPLICATION_JSON)
        .content("{\"id\": 1,\"name\":\"Ball\",
                \"price\":10,\"quantity\":100}")
        .contentType(MediaType.APPLICATION_JSON);

MvcResult result = mockMvc.perform(requestBuilder)
                    .andExpect(status().isCreated())
                    .andExpect(header()
                        .string("location",
                        containsString("/item/")))
                    .andReturn();
```

change for PUT, POST, DELETE, CATCH etc.

→ HAMCREST MATCHERS

```
public class HamcrestMatchersTest {

    @Test
    public void learning() {
        List<Integer> numbers = Arrays.asList(12,15,45);

        assertThat(numbers, hasSize(3));
        assertThat(numbers, hasItems(12,45));
        assertThat(numbers, everyItem(greaterThan(10)));
        assertThat(numbers, everyItem(lessThan(100)));

        assertThat("", isEmptyString());
        assertThat("ABCDE", containsString("BCD"));
        assertThat("ABCDE", startsWith("ABC"));
        assertThat("ABCDE", endsWith("CDE"));
```

→ AssertJ

```java
@Test
public void learning() {
    List<Integer> numbers = Arrays.asList(12,15,45);

    //assertThat(numbers, hasSize(3));
    assertThat(numbers).hasSize(3)
                       .contains(12,15)
                       .allMatch(x -> x > 10)
                       .allMatch(x -> x < 100)
                       .noneMatch(x -> x < 0);

    assertThat("").isEmpty();
    assertThat("ABCDE").contains("BCD")
                       .startsWith("ABC")
                       .endsWith("CDE");
}
```

} → AssertJ allows for chaining.