

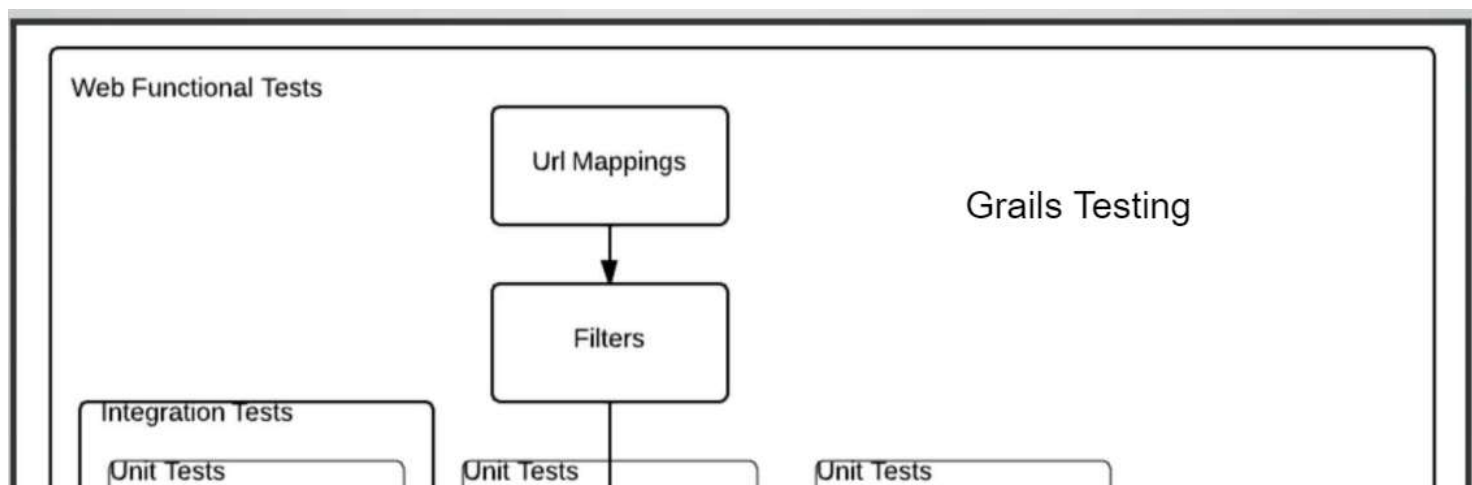
## Disclaimer & first words:

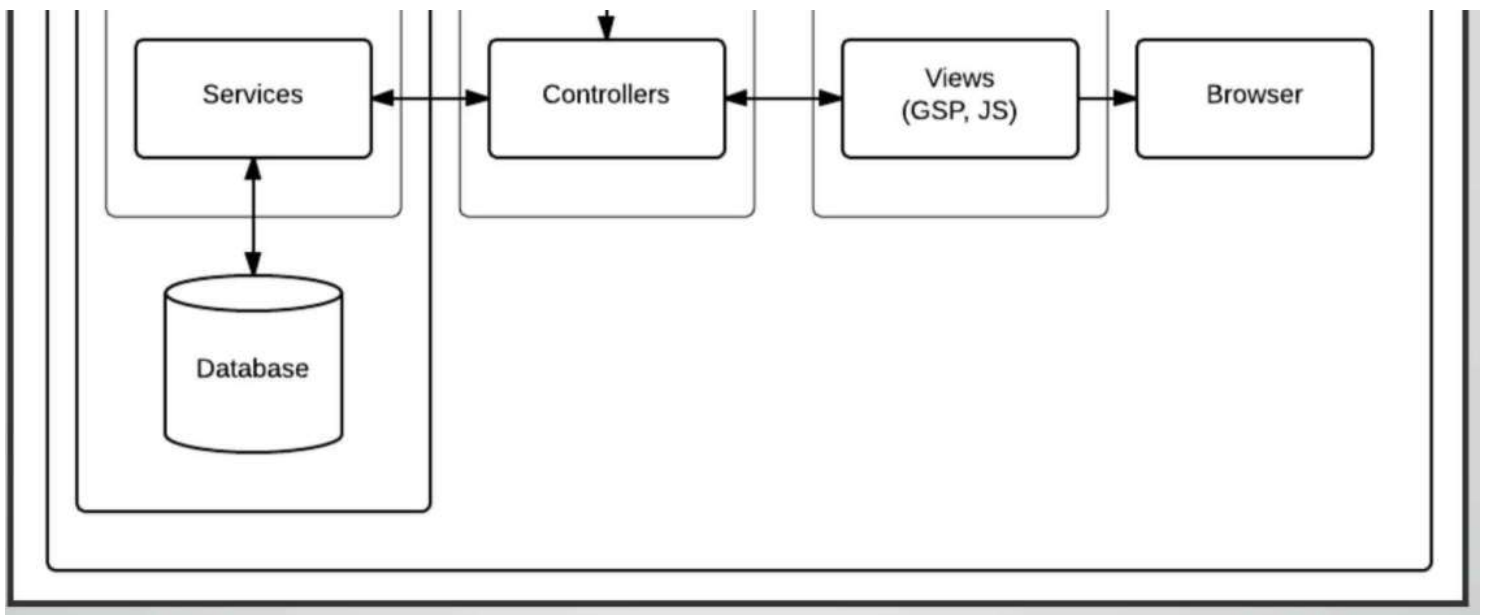
This was made along with my learning experience in Grails and the testing of it. There will be imperfections and there might be some slightly inaccurate information.

It is, therefore, more of a practical companion to the main Grails Testing Documentation, and should be treated as such.

## General Notes:

- This guide is meant to be used alongside the source code of the Classes & the tests of them for each version's project, screenshots of certain examples will however be provided here as well.
- Each Grails version that I add will have its own section
- Each version section will have its own sub-section, one for Spock and one for JUnit (Both are testing frameworks).
- Basic information will be in Grails 2.4.4, which means if something is exactly the same for another version of Grails, I will not be writing the same stuff again.  
Of course, any differences/twists will be noted, however.
- Functional testing will actually be done with Geb (and maybe original Selenium).
- Each sub-section will cover (at most) Unit, Integration and Functional tests.
- Due to practicality, I will only be demonstrating Controller Unit Tests & Service Integration Tests, as these are the most important in each Test Type.
- There will be (in time) examples both with GORM and without it, although the experience is roughly similar.
- There will be examples and notes, as well as the original code the testing is done on.
- Ideally, the tests will aim for high percentage of code coverage.
- Hibernate is only used in the GORM examples, but the version used is stated in general at the start of each section.
- If you see "Note X" comments inside the code, they are referring to the notes at the end of this guide.





## Spock Notes:

- Spock Test files end with the "Spec" suffix, like this: "MusifyControllerSpec"
- Has a "setup" method that should be placed above other methods, used for initializations/preparations for the whole testing process.
- Makes use of blocks like When Then etc.
- The general order of execution is:
  - Given: - Optional, you do initializations/setup here
  - When: - The actual logic is executed here, like method calls etc
  - Then: - This is where you make sure everything that happened in the When: block was correct
  - Cleanup: - Optional, you do cleanup here (If Rollback can't do it for you. This block is more relevant on Functional Tests)
- Block notes:
  - If an assertion in the Then: block Fails, Cleanup: will still be executed. It will not be executed, however, if there's an error/exception in the code, so keep an eye out for possible SQL exceptions or relevant issues if that happens.
  - Multiple When: - Then: pairs can exist in a single test method.
  - Of course, there are other block as well, read up on them at Spock's Documentation, at the section Scope of Interactions.
- Will extend the classes "Specification" for Unit Testing and "IntegrationSpec" for Integration Testing.

## JUnit Notes:

- To be added...

## Geb Notes:

- Framework used by default by Grails for Functional Testing.
- While indeed used for Functional Testing, it isn't the one to actually do the tests. Spock / JUnit is still responsible for that but Geb really helps simulate real User Interaction to perform a Functional Test

responsible for that, but Selenium helps simulate real user interaction to perform a functional test properly and without intense psychological damage to the programmer.

- Has different libraries for testing with Spock & JUnit.
- Is based off Selenium and is depended of it.
- Has a pretty great and thorough Documentation which I recommend reading.

## Unit Test Notes:

- Small, self-contained tests.
- One testing class per class being tested (example: "MusifyControllerSpec" tests the "MusifyController").
- Doesn't use external sources like Databases and doesn't communicate with other classes.
- Mocking is used to simulate calls to other classes, like a Service.
- Mocking is a way of simulating external classes or method calls to act in a predictable manner, as we don't really care what happens in that other class/method, only what data returns from it. We will only care about that once we get to that class/method's testing.
- Because of Mocking, almost every test will require some predictable Sample Data to run properly.
- As usual for Grails, class injection is as simple as a definition or a class annotation.

## Integration Test Notes:

- Same as Unit Tests, tests a single class one method at a time (example: "MusifyServiceIntegrationSpec" tests the "MusifyService").
- It communicates, however, normally with external sources like Databases, or other classes/methods.
- Doesn't use Mocking, instead makes actual method calls.
- Once again you need to inject dependencies per usual.
- Is supposed to be run after Unit Tests are complete & Passed, to ensure each individual method works properly first, as an Integration Test's success is based on every extra class/method called during testing (example: If MusifyServiceIntegrationSpec calls MusicHelper's method, and that method doesn't work properly, then the test might fail).
- Still uses Sample Data, but not to the extent of Unit Tests.
- Changes to the Database should be Rollbacked, meaning they will only stay for the duration of the test.
  - Spock, as far as I can tell, does this automatically as long as you extend the IntegrationSpec super class, which you should anyway for an Integration Test.
  - JUnit I think uses @Rollback annotation, but testing is required...

## Functional Test Notes:

- Tests the application as a whole, one scenario at a time. Example of a scenario: Test "Creating a Music Album":
  - Visit the "View All Albums" page
  - Click "Add a new Album" button
  - Be redirected to "Create new Album" page

- Enter data into the form
- Click "Confirm"
- Be redirected back to "View All Albums" page
- Make sure the new Album is being listed
- They are executed on a browser.
- Nothing is Mocked , but sample data is still used.
- Is supposed to be run after Integration Tests are complete & Passed, for maximum safety & chance of success.
- You can't inject dependencies here as Functional Tests run client-side (they literally happen on the browser the same way a human would go and click the buttons themselves), more on that matter in my "Grails Functional Testing issue" document.
- A browser & its driver are needed to do Functional Testing, however Gradle handles that for us, thankfully.
  - There is, however, an option to go for a "headless" browser (like PhantomJS) which runs much faster but doesn't have any visualization at all, compared to a typical browser in which you can see live the pages being loaded, forms being filled with data automatically etc.
  - Both typical & headless Functional Testing achieve the same results with the same code, but of course a typical browser makes for much easier debugging (and looks cool).

## Grails 2.4.4

JDK 1.7, Hibernate 4.3.6.1

Source Code: <https://github.com/alexioannou/Musify>

## *Unit Testing*

### Spock

First, we annotate the test then provide some sample data.

```
@TestFor(MusifyController)
class MusifyControllerSpec extends Specification {

    def sampleTitle = "sampleTitle"
    def sampleArtist = "sampleArtist"
    def sampleGenre = "sampleGenre"
    def sampleGenres1 = [[id:1, name:"Genre1"], [id:2, name:"Genre2"], [id:3, name:"Genre3"]]
    def sampleGenres2 = [[id:4, name:"Genre4"], [id:5, name:"Genre5"]]
    def sampleAlbum1 = [id:1, title:"sampleTitle1", artist:"sampleArtist1", genres:sampleGenres1]
    def sampleAlbum2 = [id:2, title:"sampleTitle2", artist:"sampleArtist2", genres:sampleGenres2]
```


Annotation defining the Controller this test is made for

Sample Data

Then, we setup the test by defining a Mock of the Service this controller works with, then replacing the original with it (it only affects this Test, of course). It can be more than one of course. Notice how every Service Method called by the Controller is mocked, but not any others. That's because we only care what gets returned when we call a method, not what happens in it.

The idea of Mocking works somewhat like this: Think of what a method called \*should\* return with some predictable given data, and make it return that.

```
def setup() {  
  controller.musifyService = [  
    fetchAllGenres: {  
      return sampleGenres1  
    },  
  
    createAlbum: { String title, String artist, def genres ->  
      //Note 1  
    },  
  
    fetchSingleAlbum: {int id ->  
      return [id:id, title:sampleTitle, artist:sampleArtist, genres: sampleGenres2]  
    },  
  
    updateAlbum: {int id, String title, String artist, def genres ->  
      //Note 1  
    },  
  
    listAlbums: {  
      return [sampleAlbum1, sampleAlbum2]  
    },  
  
    searchAlbumsAlongWithGenres: {String title, String artist, String genre ->  
      return [sampleAlbum1, sampleAlbum2]  
    },  
  
    deleteAlbum: {int id ->  
      //Note 1  
    }  
  ] as MusifyService  
}
```



Tells the controller to use this mocked Service instead

Then we begin testing each method of the Controller. Notice how we made one Testing Method for each Controller Method being tested. We can name each Testing Method as we like, but I like giving them the same name as the actual method being tested, with a "test" prefix.

Again here, we must think "what should happen/be true after this Controller Method executes?" and give it predictable sample data then assert that everything worked as expected.

In the below testing of method "Edit", we decide we're pretending to edit a music album with an ID of 12. We pass the ID to the Controller's params before calling it and catch the output of it (which is the Model being passed to the corresponding View, but we don't care about that here).

The "real" edit method is being executed, but since we mocked MusifyService, when it tried to call fetchAllGenres & fetchSingleAlbum Service Methods, it will instead call our mocked versions of them.

That way, we don't need to find a "real" album to edit, we just make the mocked Service serve a sample album

for the Controller to edit.

Lastly, we make sure everything went correctly. That is, we check the model to see if the output album is there and actually changed as needed. If it contains the new data after the edit, then everything went ok and the test Passes.

//Note 2

```
def testEdit() {
  given:
    controller.params.id = 12
  when:
    def model = controller.edit()
  then:
    assertEquals(sampleGenres1, model.genres)
    assertEquals(12, model.album.id)
    assertEquals(sampleTitle, model.album.title)
    assertEquals(sampleArtist, model.album.artist)
    assertEquals(sampleGenres2, model.album.genres)
}
```

## JUnit

To be added...

---


## Integration Testing

### Spock

No annotation needed here, but we need to inject Services used, both the one being Tested as well as external services called (as we don't do Mocks now, instead call real Service Methods).

Do note I have created a helper Service Class (MusifyTestToolkitService) to assist with testing. It provides simple methods for collecting new albums persisted in the Database to make sure they were created properly and similar functions.

```
class MusifyServiceIntegrationSpec extends IntegrationSpec {
  def musifyService
  def musifyTestToolkitService
  def sampleTitle = "sampleTitle"
  def sampleArtist = "sampleArtist"
  def sampleGenre1 = [id:12345, name:"GenOne"]
  def sampleGenre2 = [id:23456, name:"GenTwo"]
  def sampleGenre3 = [id:34567, name:"GenThree"]
  def sampleGenre4 = [id:45678, name:"GenFour"]
}
```



Service injection,  
simple as that



```
def sampleGenre5 = [id:56789, name:"GenFive"]
def sampleGenre6 = [id:67890, name:"GenSix"]
def sampleGenreIds1 = [1, 2, 3]
```

Other than that, the rest of the Test goes similarly to a Unit Test, more or less.

In the example below, testing the method that creates a Music Album, we first call the actual method with some sample data, catching the returned ID of the newly-created Album.

Then we use the helper class to bring from the Database the new Album and we assert that it's created as expected.

```
def testCreateAlbum() {
  when:
    int newAlbumId = musifyService.createAlbum(sampleTitle, sampleArtist, sampleGenreIds1)
    def createdAlbum = musifyTestToolkitService.fetchAlbumAlongWithGenreIds(newAlbumId)
  then:
    println JsonOutput.prettyPrint(JsonOutput.toJson(createdAlbum))
    assert createdAlbum.title == sampleTitle
    assert createdAlbum.artist == sampleArtist
    assert createdAlbum.genres.sort() == sampleGenreIds1
}
```

A nice little console print for some debugging or just making-sure never hurt anyone :)

In the below example we used the helper class to persist a sample album to the Database, then called the deleteAlbum method being tested, then used the helper class again to make sure it actually got deleted (which I translated into "bring it back and make sure it's Null, meaning nothing was brought back. Might have been an exaggeration but it works).

```
def testDeleteAlbum() {
  when:
    musifyTestToolkitService.persistAlbumAlongWithGenres(sampleAlbum1)
    musifyService.deleteAlbum(sampleAlbum1.id)
    def albumDeleted = musifyTestToolkitService.fetchAlbum(sampleAlbum1.id)
  then:
    println JsonOutput.prettyPrint(JsonOutput.toJson(albumDeleted))
    assert albumDeleted == null
}
```

## JUnit

To be added...

---

## Functional Testing

There will be no Functional Testing done in 2.4.4, as due to lack of documentation & examples of it, dropping of support for this version from Grails as well as time passed since 2.4.4 was active, I couldn't manage to make properly working Functional Tests for this version. If for some reason I stumble upon a way of doing it however, I'll add it here.

---

# Grails 4.0.6

JDK 1.8, Hibernate 5.7.0.4

Source Code: <https://github.com/alexioannou/Musify4> (pretty much copy-pasted code from "Musify" project, but it works in Grails 4)

## Unit Testing

### Spock

Same as 2.4.4...

### JUnit

Same as 2.4.4...

---

## Integration Testing

### Spock

Same as 2.4.4...

### JUnit

Same as 2.4.4...

---

## Functional Testing

### Spock

First we annotate with @Integration to take advantage of it's auto-wiring (automatic injection handling), then create some sample data to use.

IMPORTANT: This is the issue mentioned in my Grails Functional Testing issue document, seriously, go read it first to avoid headaches.

```
@Integration
class Musify4FunctionalSpec extends GebSpec{

    def musify4TestToolkitService

    def sampleTitle = "sampleTitle"
    def sampleArtist = "sampleArtist"
    def sampleGenre1 = [id:12345, name:"GenOne"]
    def sampleGenre2 = [id:23456, name:"GenTwo"]
    def sampleGenre3 = [id:34567, name:"GenThree"]
    def sampleGenre4 = [id:45678, name:"GenFour"]
    def sampleGenre5 = [id:56789, name:"GenFive"]
    def sampleGenre6 = [id:67890, name:"GenSix"]
    def sampleGenreIds1 = [1, 2, 3]
    def sampleAlbum1 = [id:1111111, title:"sampleTitle1", artist:"sampleArtist1", genres:[sampleGenre1, sampleGenre2]]
    def sampleAlbum2 = [id:2222222, title:"sampleTitle2", artist:"sampleArtist2", genres:[sampleGenre3, sampleGenre4, sampleGenre5]]
```

As we annotated with @Integration, we can easily inject our dependency without much fuss

Copy-pasted sample data from previous tests  
Why not? :)

Then, as usual for spock, we have access to setup() & cleanup() methods before running any tests, but I won't



make use of them, instead make setup/initializations individually for each test method.

Now we are ready to start doing tests.

We're testing the scenario where a user would view a list of all the albums:

We persist a couple albums in the database.

We go to the correct page.

We assert that these 2 albums are shown.

Lastly, we cleanup by removing them from the database. Here there's no rollback, as this is done client-side, so you need to manually set your cleanup block.

```
def testViewAlbums() {  
  given:  
    musify4TestToolkitService.persistAlbumAlongWithGenres(sampleAlbum1)  
    musify4TestToolkitService.persistAlbumAlongWithGenres(sampleAlbum2)  
  when:  
    ListAlbumsPage listAlbumsPage = to(ListAlbumsPage)  
    Thread.sleep(5000)  
  then:  
    assert listAlbumsPage.containsAlbum(sampleAlbum1)  
    assert listAlbumsPage.containsAlbum(sampleAlbum2)  
  cleanup:  
    musify4TestToolkitService.cleanupAlbumWithGenres(sampleAlbum1)  
    musify4TestToolkitService.cleanupAlbumWithGenres(sampleAlbum2)  
}
```

You may have noticed the "ListAlbumsPage". That's a class that extends Geb's Page class, and provides much easier way of navigating through the actual page & interacting with it, although it requires some minor setup first:

```
class EditAlbumPage extends Page{  
  
  static url = "http://localhost:8080/musify4/edit"  
  static at = { title.contains("Edit Album") }  
  
  static content = {  
    titleField(wait:true) {"#titleField"}  
    artistField(wait:true) {"#artistField"}  
    genres(wait:true) {"#genres"}  
    confirmButton(wait:true) {"#confirmButton"}  
    cancelButton(wait:true) {"#cancelButton"}  
  }  
}
```

```

def selectGenre(def genre) {
    genres = [genre.id]
}
}

```

The above code "translates" the page below in such a way that can be manipulated for groovy-style.

The general idea is this:

- Each page in a Geb test has it's own class counterpart.
  - While optional, it's still a good idea to create a Page class for every page in your project (or the part that's being tested) as it makes for (much) cleaner testing code as well as adding modularity.
- The main components of a page are:
  - Its url, which is self-explanatory. Note this is the base url of the page, so in the above example if I was to edit the specific album with id 87, the actual url would be composed of the base url shown in the image plus a "/87" concatenated at the end.
  - Its "at" block. This is used in certain page methods where you need assert that you ARE in the correct page at this point of time.
  - Its content block. This is the main thing here. You basically give a name to every html element in the page (that you care about) using Geb selectors (which are based of and more or less identical to jQuery selectors). This way, you can then do, for example: `editAlbumPage.titleField.value("whatever")`, which would go select that field and type "whatever" in it, in a way that's very elegant/groovy-style.
    - Every element "named" here is now called a Navigator.
    - Geb's selectors work the same as jQuery selectors, in the aspect that, if you define a navigator with the selector `$("input")`, it will be a list of navigators instead, so you can do stuff like `.each{} etc`
- As this is a class, it can have methods as usual.
- Read up more on this on Geb's Documentation under the section of Pages

Another thing Geb provides is what's called Modules.

A Module is a definition of an element (or more often, a group of elements) that's reusable between different pages.

With modules you can now define a navigator (say, a table) in multiple pages that consists of a number of modules (say, rows) without having to define that table navigator for each page.

- Or, you can make the module inside the Page file (but outside the Page class) so only this Page can use it, in the case that this page has, for example, multiple tables.
- I guess it can also help make the code cleaner this way by seperating part of it from the rest?

All instances of the same module work the same.

Example of a Module below:

```

class AlbumListRow extends Module {
    static content = {
        // ...
    }
}

```

```

artistCell(wait:true) {$( selector: "#artistCell")}
titleCell(wait:true) {$( selector: "#titleCell")}
genresCell(wait:true) {$( selector: "#genresCell")}
editCell(wait:true) {$( selector: "#editCell")}
deleteCell(wait:true) {$( selector: "#deleteCell")}
}
}

```

Now I can just go to a page and define a table navigator do this:

Note here that I don't really care about the table itself, so instead I directly reference its `tBody` tag, which holds the actual rows of the table, as those are what I need to interact with

```

tableBody(wait: true) { $( selector: "#myTable tbody").children().moduleList(AlbumListRow) }

```

With this, I just defined the navigator `tableBody` AND also said that it's children are actually `AlbumListRow` modules.

For this to work, of course, it would be required that INSIDE the `tableBody` there are actually some elements with the IDs mentioned in the `AlbumListRow` example, so that they could be properly translated/mapped/correspond to (choose your favorite work) an `AlbumListRow`.

Again, Geb's documentation section `Modules` covers this more properly and in greater detail.

## JUnit

To be added...

### Notes:

- Note 1:  
A **Controller Action** that doesn't get returned any output data from a **Service Method** shouldn't need to worry about what the **Service Method** it calls does, therefore we don't need to simulate any logic inside the mocked **Service Method** as well as not needing to return anything.

- Note 2:  
`assert` can be used in place of `==` and of `assertEquals`.

For example, the next 3 statements can be used interchangeably:

```

name == "John"
assert name == "John"
assertEquals("John", name) //Recommended method

```

However, `assertEquals` is the recommended way to test if two variables have the same value, as it makes for clearer testing code and reduces risk of error, like using a single `=` instead of `==`.

- 

- Note 3:

When comparing **Lists** of any type (ex. **ArrayList**) using **assertEquals**, order matters.  
For example, the first statement is false but the second is true:

```
assertEquals([1, 2], [2, 1])    //returns false
assertEquals(["John", "George"], ["John", "George"])    //returns true
```

- Note 4:

The parameters (**params**) a **Controller Action** has when it is called can be accessed in the respective **Test Method** through either of the 3 ways below:

```
params
controller.params    //Recommended method
this.params
```

However, **controller.params** is the recommended way to reference **params**, as it makes for clearer testing code.

- Note 5:

When using **assertEquals** always put the **Expected Value** as the first parameter and the **Value You Are Checking** as the second.

Example:

```
String name = "John Smith"
Customer customer = new Customer(name)
assertEquals(name, customer.getName())
```

Although it doesn't produce any different example, it makes for clearer testing code and help during debugging.

- Note 6:

If you want to achieve **Full Code Coverage** with the test (meaning: test all scenarios / every piece of code) and there are **if/else/switch** in your **Method/Action**, you need to make one **Unit Test** per possible "branch/path".

For example: