

Improved existing notes.

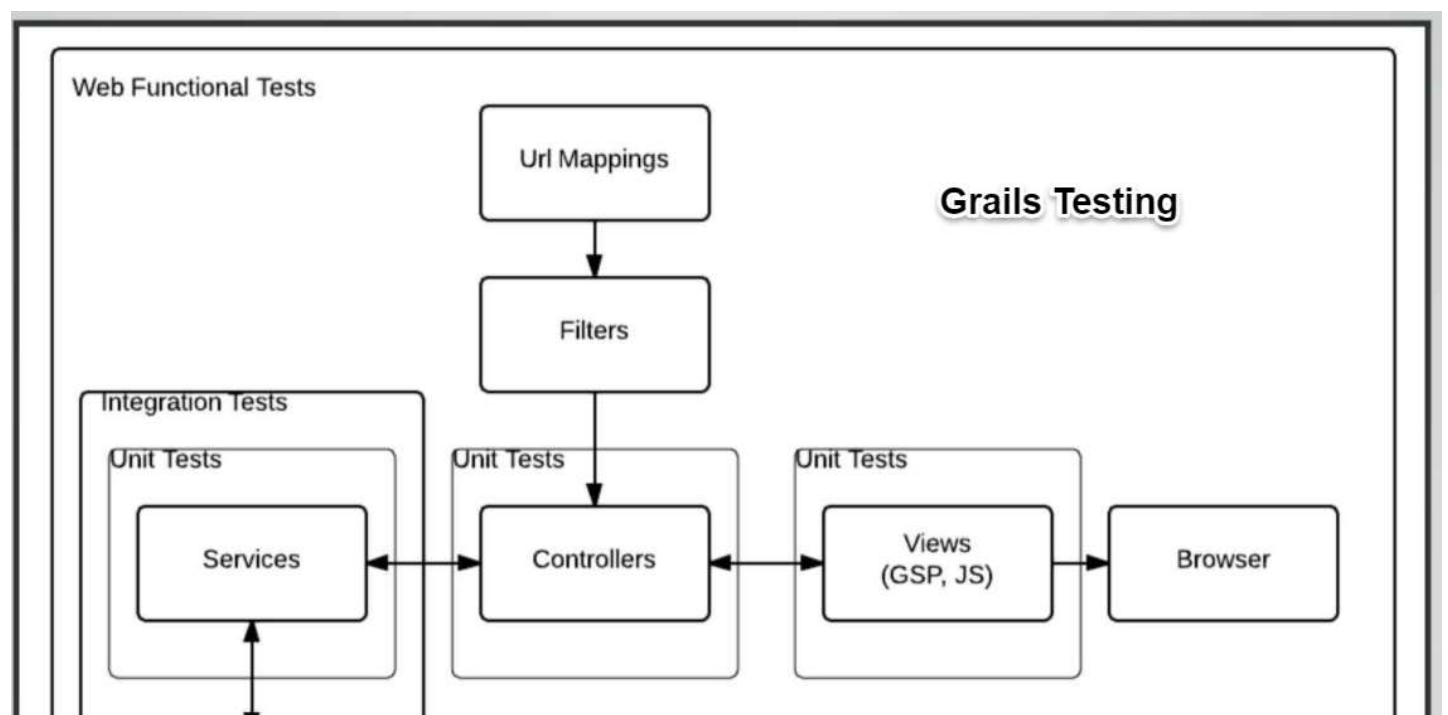
## Disclaimer & first words:

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).
- 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 testing process.
- Makes use of the Given, When, Then keywords.
- "Given" is optional, used to do definitions, make initializations of data or other preparations for this method's test.
- "When" is where the method you are testing is called, along with any necessary functionality that needs to take place after that method call, if any.
- "Then" is where you make sure the testing Passed. You place assertions or other Boolean-outputting code here, and as long as none of them returned false, the test has passed.
- Code that doesn't return true/false (like a println or other method calls) can still exist in "Then".
- The test will still be considered Passed even if there are no True statements in "Then". The important thing is to not have any False statements.
- Will extend the classes "Specification" for Unit Testing and "IntegrationSpec" for Integration Testing.

## JUnit Notes:

- To be added...

## 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.
- 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 will be Rollbacked, meaning they will only stay for the duration of the test. This is, I believe, due to the extension of the IntegrationSpec super class, as no @Rollback annotation is needed here).

## Functional Test Notes:

- Tests the application as a whole, one scenario at a time (example: test "Creating a Music Album" scenario).
- Nothing is Mocked and no Sample Data is used, except for User Input.
- Is supposed to be run after Integration Tests are complete & Passed, for maximum safety & chance of success.
- More to be added...

---

## 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)  Annotation defining the Controller this test is made for
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]
```

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

Service injection,  
simple as that

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.

## Spock Unit Testing Controllers

### Quick notes:

- One Unit Test file per Controller
- Define some sample data that will be used during testing
- Define the mocked Services and their Methods that the Controller uses/calls
- Test each Controller Action in a single Test Method

## 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:

```
assertEquals("John", name)
```

```
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:

```
boolean flag = true
if(flag)
{
    //doSomething...
}
else
{
    //doSomethingElse...
}
```

This would require two distinct [Unit Tests](#), one where [flag = true](#) and one where [flag = false](#).

Ideally you should always aim for [Full Code Coverage](#) but there are some cases where a certain part