Spocklight

NOTEBOOK

Hubert A. Klein Ikkink

HA.
KI

Spocklight Notebook

Experience the Spock framework through code snippets

Hubert Klein Ikkink

This book is for sale at http://leanpub.com/spockframeworknotebook

This version was published on 2014-10-28

Tweet This Book!

Please help Hubert Klein Ikkink by spreading the word about this book on Twitter!

The suggested hashtag for this book is #spocklightnotebook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#spocklightnotebook

Contents

About Me

I am born in 1973 and live in Tilburg, the Netherlands, with my beautiful wife and three gorgeous children. I am also known as mrhaki, which is simply the initials of his name prepended by mr. The following Groovy snippets shows how the alias comes together:

```
['Hubert', 'Alexander', 'Klein', 'Ikkink'].inject('mr') { alias, name ->
    alias += name[0].toLowerCase()
}
```

(How cool is Groovy that we can express this in a simple code sample ;-))

I studied Information Systems and Management at the Tilburg University. After finishing my studies I started to work at a company which specialized in knowledge-based software. There I started writing my first Java software (yes, an applet!) in 1996. Over the years my focus switched from applets, to servlets, to Java Enterprise Edition applications, to Spring-based software.

In 2008 I wanted to have fun again when writing software. The larger projects I was working on were more about writing configuration XML files, tuning performance and less about real development. So I started to look around and noticed Groovy as a good language to learn about. I could still use existing Java code, libraries, and my Groovy classes in Java. The learning curve isn't steep and to support my learning phase I wrote down interesting Groovy facts in my blog with the title *Groovy Goodness*. I post small articles with a lot of code samples to understand how to use Groovy. Since November 2011 I am also a DZone Most Valuable Blogger (MVB); DZone also posts my blog items on their site.

In 2010, 2011, 2012, 2013 and 2014 I was invited to speak at Gr8Conf in Copenhagen, Denmark. This is a very good conference with all the project leaders of Groovy and Groovy-related projects. In 2014 I also spoke at the Gr8Conf US. In November 2010 I presented a Gradle talk at the J-Fall conference of the Dutch Java User Group. In November 2011 I presented about the new features in Groovy 1.8 at the same conference. During the 2013 edition I presented a hands-on Gradle workshop. The conference is visited by 1000 Java developers and I got the chance to educate some of them about the greatness of Gradle and Groovy.

I work for a company called JDriven in the Netherlands. JDriven focuses on technologies that simplify and improve development of enterprise applications. Employees of JDriven have years of experience with Java and related technologies and are all eager to learn about new technologies. I work on projects using Grails and Java combined with Groovy and Gradle.

Introduction

The Spock framework is a testing and specification framework for JVM languages, like Java and Groovy. When I started to learn about Spock I wrote done little code snippets with features of Spock I found interesting. To access my notes from different locations I wrote the snippets with a short explanation in a blog: Messages from mrhaki. I labeled the post as *Spocklight*, because I thought this is good stuff that needs to be in the spotlight.

A while ago I bundled all my Groovy and Grails Goodness blog posts in a book published at Leanpub. Leanpub is very easy to use and I could use Markdown to write the content, which I really liked as a developer. So it felt natural to also bundle the Spocklight blog posts at Leanpub.

The book is intended to browse through the subjects. You should be able to just open the book at a random page and learn more about Spock. Maybe pick it up once in a while and learn a bit more about known and lesser known features of Spock.

I hope you will enjoy reading the book and it will help you with learning about the Spock framework, so you can apply all the goodness in your projects.

Code samples

Code samples in the book can contain a backslash (\) at the end of the line to denote the line actually continues. This character is not part of the code, but for better formatting of the code blocks in the book the line is split. If you want to use the code you must keep this in mind.

Introduction to Spock Testing

In this blog entry we will start with a simple Spock specification for a Groovy class we create. We can learn why to use Spock on the Spock website. In this article we show with code how our tests will work with Spock. To get started with this sample we need Gradle installed on our machine and nothing else. The current release of Gradle at the time of writing this entry is 0.9-preview-3.

First we need to create a new directory for our little project and then we create a new `build.gradle` file:

```
$ mkdir spock-intro
$ cd spock-intro
$ mkdir -p src/main/groovy/com/mrhaki/blog src/test/groovy/com/mrhaki/blog
```

Now we create the `build.gradle` file:

```
// File: build.gradle
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    groovy 'org.codehaus.groovy:groovy:1.7.3'
    testCompile 'org.spockframework:spock-core:0.4-groovy-1.7'
}
```

Next we create our tests, but in Spock they are called specifications. We only need to extend the `spock.lang.Specification` and we get all the Spock magic in our hands. We start simple by defining a specification where we want to count the number of users in a `UserService` class. We are going to create the `UserService` class later, we start first with our specification:

```
package com.mrhaki.blog

import spock.lang.Specification

class UserServiceSpecification extends Specification {

    def "Return total number of users"() {
        setup: 'Create UserService instance with 2 users'
        UserService userService = new UserService(users: ['mrhaki', 'hubert'])

        expect: 'Invoke count() method'
        2 == userService.count()
    }

}
```

Notice at line 6 how we can use very descriptive method names by using String literals. Next we create an instance of the `UserService` class and pass a list of two users at line 8. And then we check if the return value is the expected value 2 with a simple assertion statement. Spock provides a very readable way to write code. Mostly we first setup code for testing, run the code and finally test the results. This logic is supported nicely by Spock by using the labels `setup` and `expect`. Later on we see more of these labels.

Before we run the test we create our `UserService` class:

```
package com.mrhaki.blog

class UserService {

    Collection<String> users

    int count() {
        users ? users.size() : 0
    }

}
```

We can run our code and test with the following command:

```
$ gradle test
:compileJava UP-TO-DATE
:compileGroovy
:processResources UP-TO-DATE
:classes
:compileTestJava
:compileTestGroovy
:processTestResources UP-TO-DATE
:testClasses
:test

BUILD SUCCESSFUL

Total time: 12.475 secs
```

The source files are compiled and our specification is run. We get the `BUILD SUCCESSFUL` message indicating our test runs fine. If the test would fail we can open `build/reports/tests/index.html` or `build/test-results/TEST-com.mrhaki.blog.UserServiceSpecification.xml` to see the failure.

We specified the `count()` method must return the number of users, but we only check it for 2 elements, but what if we want to test to see if 0 and 1 user also return the correct count value? We can create new methods in our specification class, but Spock makes it so easy to do this elegantly:

```
package com.mrhaki.blog

import spock.lang.Specification

class UserServiceSpecification extends Specification {

    def "Return total number of users"() {
        setup: 'Create UserService instance with users'
        UserService userService = new UserService(users: userList)

        expect: 'Invoke count() method'
        expectedCount == userService.count()

        where:
        expectedCount   | userList
        0               | null
        0               | []
        1               | ['mrhaki']
        2               | ['mrhaki', 'hubert']
    }

}
```

So what happens here? We use a new label `where` which contains a data table. Each row of the data table represent a new test run with the data from the row. In the `setup` block we used an unbound variable `userList` and in the `expect` block the unbound variable `expectedCount`. The variables get their values from the data table rows in the `where` block. So the first run the `UserService` instances gets `null` assigned to the `users` property and we expect the value `0` to be returned by the `count()` method. In the second run we pass an empty list and expect also `0` from the `count()` method. We have four rows, so our test is run four times when we invoke `$ gradle test`.

We can make the fact that four tests are run explicit by using the `@Unroll` annotation. We can use a String as argument describing the specific variable values used in a run. If we use the `#` followed by the unbound variable name will it be replaced when we run the code:

```
package com.mrhaki.blog

import spock.lang.Specification
import spock.lang.Unroll

class UserServiceSpecification extends Specification {

    @Unroll("Expect to count #expectedCount users for following list #userList")
    def "Return total number of users"() {
        setup: 'Create UserService instance with users'
        UserService userService = new UserService(users: userList)

        expect: 'Invoke count() method'
        expectedCount == userService.count()
```

```
        where:
        expectedCount   | userList
        0               | null
        0               | []
        1               | ['mrhaki']
        2               | ['mrhaki', 'hubert']
    }

}
```

The generated XML with the test result contains the four runs with their specific names:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite errors="0" failures="0" hostname="ci-test" name="com.mrhaki.blog.UserService\
Specification" tests="4" time="0.707" timestamp="2010-06-29T18:17:24">
  <properties />
  <testcase classname="com.mrhaki.blog.UserServiceSpecification" name="Expect to count \
0 users for following list null" time="0.152" />
  <testcase classname="com.mrhaki.blog.UserServiceSpecification" name="Expect to count \
0 users for following list []" time="0.027" />
  <testcase classname="com.mrhaki.blog.UserServiceSpecification" name="Expect to count \
1 users for following list [mrhaki]" time="0.0050" />
  <testcase classname="com.mrhaki.blog.UserServiceSpecification" name="Expect to count \
2 users for following list [mrhaki, hubert]" time="0.0010" />
  <system-out><![CDATA[]]></system-out>
  <system-err><![CDATA[]]></system-err>
</testsuite>
```

This concludes the introduction to Spock testing. In the future we learn more about Spock and the great features it provide to make writing tests easy and fun.

Original blog post written on June 29, 2010.

Assert Magic

One of the many great features of Spock is the way assertion failures are shown. The power assert from Groovy is based on Spock's assertion feature, but Spock takes it to a next level. Let's create a simple specification for a course service, which is able to create new courses:

```
// File: CourseServiceSpec.groovy
package com.mrhaki.blog

@Grab('org.spockframework:spock-core:0.4-groovy-1.7')
import spock.lang.Specification

class CourseServiceSpec extends Specification {

    def "Create new course with teacher and description"() {
        setup:
        def courseService = new CourseService()

        when:
        def course = courseService.create('mrhaki', 'Groovy Goodness')

        then:
        'Mrhaki' == course.teacher.name
        'Groovy Goodness' == course.description
        !course.students
    }

}

class CourseService {
    Course create(String teacherName, String description) {
        new Course(teacher: new Person(name: teacherName), description: description)
    }
}

class Course {
    Person teacher
    String description
    List<Person> students
}

class Person {
    String name
}
```

At lines 16, 17, 18 we define the assertions for our specification. First of all we notice we don't add the keyword `assert` for each assertion. Because we are in the `then` block we can omit the `assert` keyword. Notice at line 18 we can test for null values by using the Groovy truth. We also notice we only have to write a simple assertion. Spock doesn't need a bunch of `assertEquals()` methods like JUnit to test the result.

Now it is time to run our specification as JUnit test and see the result:

```
$ groovy CourseServiceSpec.groovy
JUnit 4 Runner, Tests: 1, Failures: 1, Time: 232
Test Failure: Create new course with teacher and description(com.mrhaki.blog.CourseServ\
iceSpec)
Condition not satisfied:

'Mrhaki' == course.teacher.name
         |  |      |       |
         |  |      |       mrhaki
         |  |      com.mrhaki.blog.Person@34b6a6d6
         |  com.mrhaki.blog.Course@438346a3
         false
         1 difference (83% similarity)
         (M)rhaki
         (m)rhaki

    at com.mrhaki.blog.CourseServiceSpec.Create new course with teacher and description\
(CourseServiceSpec.groovy:16)
```

Wow that is a very useful message for what is going wrong! We can see our condition `'Mrhaki' == course.teacher.name` is not satisfied, but we even get to see which part of the String value is not correct. In this case the first character should be lowercase instead of uppercase, but the message clearly shows the rest of the String value is correct. As a matter of fact we even know 83% of the String values is similar.

Another nice feature of Spock is that only the line which is important is shown in the abbreviated stacktrace. So we don't have to scroll through a big stacktrace with framework classes to find out where in our class the exception occurs. We immediately see that at line 16 in our specification the condition is not satisfied.

In our sample we have three assertions to be checked in the `then`. If we get a lot of assertions in the `then` block we can refactor our specification and put the assertions in a new method. This method must have void return type and we must add the `assert` keyword again. After these changes the assertions work just like when we put them in the `then` block:

```
package com.mrhaki.blog

@Grab('org.spockframework:spock-core:0.4-groovy-1.7')
import spock.lang.Specification

class CourseServiceSpec extends Specification {

    def "Create new course with teacher and description"() {
        setup:
        def courseService = new CourseService()

        when:
        def course = courseService.create('mrhaki', 'Groovy Goodness')

        then:
        assertCourse course
    }

    private void assertCourse(course) {
        assert 'mrhaki' == course.teacher.name
        assert 'Grails Goodness' == course.description
        assert !course.students
    }

}

class CourseService {
    Course create(String teacherName, String description) {
        new Course(teacher: new Person(name: teacherName), description: description)
    }
}

class Course {
    Person teacher
    String description
    List<Person> students
}

class Person {
    String name
}
```

When can run our specification and get the following output:

```
$ groovy CourseServiceSpec.groovy
JUnit 4 Runner, Tests: 1, Failures: 1, Time: 228
Test Failure: Create new course with teacher and description(com.mrhaki.blog.CourseServ\
iceSpec)
Condition not satisfied:

'Grails Goodness' == course.description
                  |  |      |
                  |  |      Groovy Goodness
                  |  com.mrhaki.blog.Course@3435ec9
                  false
                  4 differences (73% similarity)
                  Gr(ails) Goodness
                  Gr(oovy) Goodness

    at com.mrhaki.blog.CourseServiceSpec.assertCourse(CourseServiceSpec.groovy:21)
    at com.mrhaki.blog.CourseServiceSpec.Create new course with teacher and description\
(CourseServiceSpec.groovy:16)
```

Spock provides very useful assertion messages when the condition is not satisfied. We see immediately what wasn't correct, because of the message and the fact the stacktrace only shows the line where the code is wrong.

Original blog post written on July 05, 2010.

Check for Exceptions with Spock

With Spock we can easily write feature methods in our specification to test if an exception is thrown by the methods invoked in a when block. Spock support exception conditions with the `thrown()` and `notThrown()` methods. We can even get a reference to the expected exception and check for example the message.

The following piece of code contains the specification to check for exceptions that can be thrown by a `cook()` method of the `RecipeService` class. And we check that exception are not thrown. The syntax is clear and concise, what we expect from Spock.

```
package com.mrhaki.blog

@Grab('org.spockframework:spock-core:0.5-groovy-1.7')
import spock.lang.Specification

class RecipeServiceSpec extends Specification {
    def "If cooking for more minutes than maximum allowed by recipe throw BurnedExcepti\
on"() {
        setup:
            def recipeService = new RecipeService()
            def recipe = new Recipe(time: 5, device: 'oven')

        when:
            recipeService.cook(recipe, 10, 'oven')

        then:
            thrown BurnedException
    }

    def "If cooking on wrong device throw InvalidDeviceException"() {
        setup:
            def recipeService = new RecipeService()
            def recipe = new Recipe(device: 'oven', time: 10)

        when:
            recipeService.cook(recipe, 2, 'microwave')

        then:
            InvalidDeviceException ex = thrown()
            // Alternative syntax: def ex = thrown(InvalidDeviceException)
            ex.message == 'Please use oven for this recipe.'
    }

    def """If the recipe is cooked on the right device and
           for the correct amount of minutes,
```

```
           then no exception is thrown"""() {
        setup:
            def recipeService = new RecipeService()
            def recipe = new Recipe(device: 'oven', time: 30)

        when:
            recipeService.cook(recipe, 30, 'oven')

        then:
            notThrown BurnedException
            notThrown InvalidDeviceException
    }
}

class RecipeService {
    def cook(Recipe recipe, int minutes, String device) {
        if (minutes > recipe.time) {
            throw new BurnedException()
        }
        if (device != recipe.device) {
            throw new InvalidDeviceException("Please use $recipe.device for this recipe\
."")
        }
    }
}

class Recipe {
    int time
    String device
}

class BurnedException extends RuntimeException {
    BurnedException(String message) {
        super(message)
    }
}

class InvalidDeviceException extends RuntimeException {
    InvalidDeviceException(String message) {
        super(message)
    }
}
```

Original blog post written on January 21, 2011.

Using the Old Method

Spock has some great features to write specifications or tests that are short and compact. One of them is the `old()` method. The `old()` method can only be used in a `then:` block. With this method we get the value a statement had before the `when:` block is executed.

Let's see this with a simple example. In the following specification we create a `StringBuilder` with an initial value. In the `then:` block we use the same initial value for the assertion:

```
package com.mrhaki.spock

class SampleSpec extends spock.lang.Specification {

    def "after addition of a new value the content is the initial value with the append\
ed value"() {
        given:
        final StringBuilder builder = new StringBuilder('Spock ')

        when:
        builder << appendValue

        then:
        builder.toString() == 'Spock ' + appendValue

        where:
        appendValue << ['rocks!', 'is fun.', 'amazes.']
    }

}
```

If we want to change the initial value when we create the `StringBuilder` we must also change the assertion. We can refactor the feature method and show our intention of the specification better. We add the variable `oldToString` right after we have created the `StringBuilder`. We use this in the assertion.

```
package com.mrhaki.spock

class SampleSpec extends spock.lang.Specification {

    def "after addition of a new value the content is the initial value with the append\
ed value"() {
        given:
        final StringBuilder builder = new StringBuilder('Spock ')
        final String oldToString = builder.toString()

        when:
```

```
        builder << appendValue

        then:
        builder.toString() == oldToString + appendValue

        where:
        appendValue << ['rocks!', 'is fun.', 'amazes.']
    }

}
```

But with Spock we can do one better. Instead of creating an extra variable we can use the `old()` method. In the assertion we replace the variable reference `oldToString` with `old(builder.toString())`. This actually means we want the value for `builder.toString()` BEFORE the `when:` block is executed. The assertion also is now very clear and readable and the intentions of the specification are very clear.

```
package com.mrhaki.spock

class SampleSpec extends spock.lang.Specification {

    def "after addition of a new value the content is the initial value with the append\
ed value"() {
        given:
        final StringBuilder builder = new StringBuilder('Spock ')

        when:
        builder << appendValue

        then:
        builder.toString() == old(builder.toString()) + appendValue

        where:
        appendValue << ['rocks!', 'is fun.', 'amazes.']
    }

}
```

Let's change the specification a bit so we get some failures. Instead of adding the `appendValue` data variable unchanged to the `StringBuilder` we want to add a capitalized value.

```
package com.mrhaki.spock

class SampleSpec extends spock.lang.Specification {

    def "after addition of a new value the content is the initial value with the append\
ed value"() {
        given:
        final StringBuilder builder = new StringBuilder('Spock ')

        when:
        builder << appendValue.capitalize()

        then:
        builder.toString() == old(builder.toString()) + appendValue

        where:
        appendValue << ['rocks!', 'is fun.', 'amazes.']
    }

}
```

If we run the specification we get assertion failures. In the following output we see such a failure and notice the value for the `old()` is shown correctly:

```
Condition not satisfied:

builder.toString() == old(builder.toString()) + appendValue
|       |          |  |                       | |
|       |          |  Spock                   | rocks!
|       |          |                          Spock rocks!
|       |          false
|       |          1 difference (91% similarity)
|       |          Spock (R)ocks!
|       |          Spock (r)ocks!
|       Spock Rocks!
Spock Rocks!
```

Note: If we use the `old()` method we might get an `InternalSpockError` exception when assertions fail. The error looks something like: `org.spockframework.util.InternalSpockError: Missing value for expression "..."`. Re-ordering the assertion can help solve this. For example putting the `old()` method statement last. In Spock 1.0-SNAPSHOT this error doesn't occur.

For more information we can read Rob Fletcher's blog post about the `old()` method.

Code written with Spock 0.7-groovy-2.0.

Original blog post written on August 22, 2013.

Change Return Value of Mocked or Stubbed Service Based On Argument Value

My colleague Albert van Veen wrote a blog post about Using ArgumentMatchers with Mockito. The idea is to let a mocked or stubbed service return a different value based on the argument passed into the service. This is inspired me to write the same sample with Spock.

Spock already has built-in mock and stub support, so first of all we don't need an extra library to support mocking and stubbing. We can easily create a mock or stub with the `Mock()` and `Stub()` methods. We will see usage of both in the following examples.

In the first example we simply return true or false for `ChocolateService.doesCustomerLikesChocolate()` in the separate test methods.

```
import spock.lang.*

public class CandyServiceSpecification extends Specification {

    private ChocolateService chocolateService = Mock()
    private CandyService candyService = new CandyServiceImpl()

    def setup() {
        candyService.chocolateService = chocolateService
    }

    def "Customer Albert really likes chocolate"() {
        given:
        final Customer customer = new Customer(firstName: 'Albert')

        and: 'Mock returns true'
        1 * chocolateService.doesCustomerLikesChocolate(customer) >> true

        expect: 'Albert likes chocolate'
        candyService.getCandiesLikeByCustomer(customer).contains Candy.CHOCOLATE
    }

    def "Other customer do not like chocolate"() {
        given:
        final Customer customer = new Customer(firstName: 'Any other firstname')

        and: 'Mock returns false'
        1 * chocolateService.doesCustomerLikesChocolate(customer) >> false

        expect: 'Customer does not like chocolate'
        !candyService.getCandiesLikeByCustomer(customer).contains(Candy.CHOCOLATE)
    }
```

```
}
```

In the following example we mimic the ArgumentMatcher and this time we use a stub instead of mock.

```
import spock.lang.*

public class CandyServiceSpecification extends Specification {

    private CandyService candyService = new CandyServiceImpl()

    def setup() {
        candyService.chocolateService = Stub(ChocolateService) {
            getCandiesLikeByCustomer(_) >> { Customer customer ->
                customer?.firstName == 'Albert'
            }
        }
    }

    def "Customer Albert really likes chocolate"() {
        given:
        final Customer customer = new Customer(firstName: 'Albert')

        expect: 'Albert likes chocolate'
        candyService.getCandiesLikeByCustomer(customer).contains Candy.CHOCOLATE
    }

    def "Other customer do not like chocolate"() {
        given:
        final Customer customer = new Customer(firstName: 'Any other firstname')

        expect: 'Customer does not like chocolate'
        !candyService.getCandiesLikeByCustomer(customer).contains(Candy.CHOCOLATE)
    }

}
```

Code written with Spock 0.7-groovy-2.0

Original blog post written on May 17, 2013.

Writing Assertions for Arguments Mock Methods

My colleague Arthur Arts has written a blog post Tasty Test Tip: Using ArgumentCaptor for generic collections with Mockito. This inspired me to do the same in Spock. With the ArgumentCaptor in Mockito the parameters of a method call to a mock are captured and can be verified with assertions. In Spock we can also get a hold on the arguments that are passed to method call of a mock and we can write assertions to check the parameters for certain conditions.

When we create a mock in Spock and invoke a method on the mock the arguments are matched using the `equals()` implementation of the argument type. If they are not equal Spock will tell us by showing a message that there are too few invocations of the method call. Let's show this with an example. First we create some classes we want to test:

```
package com.mrhaki.spock

public class ClassUnderTest {

    private final Greeting greeting

    ClassUnderTest(final Greeting greeting) {
        this.greeting = greeting
    }

    String greeting(final List<Person> people) {
        greeting.sayHello(people)
    }
}


package com.mrhaki.spock

interface Greeting {
    String sayHello(final List<Person> people)
}


package com.mrhaki.spock

@groovy.transform.Canonical
class Person {
    String name
}
```

Now we can write a Spock specification to test `ClassUnderTest`. We will now use the default matching of arguments of a mock provided by Spock.

```
package com.mrhaki.spock

import spock.lang.Specification

class SampleSpecification extends Specification {

    final ClassUnderTest classUnderTest = new ClassUnderTest()

    def "check sayHello is invoked with people in greeting method"() {
        given:
        final Greeting greeting = Mock()
        classUnderTest.greeting = greeting

        and:
        final List<Person> people = ['mrhakis', 'hubert'].collect { new Person(name: it\
) }

        when:
        final String greetingResult = classUnderTest.greeting(people)

        then:
        1 * greeting.sayHello([new Person(name: 'mrhaki'), new Person(name: 'hubert')])
    }

}
```

When we execute the specification we get a failure with the message that there are too few invocations:

```
...
Too few invocations for:

1 * greeting.sayHello([new Person(name: 'mrhaki'), new Person(name: 'hubert')])   (0 in\
vocations)

Unmatched invocations (ordered by similarity):

1 * greeting.sayHello([com.jdriven.spock.Person(mrhakis), com.jdriven.spock.Person(hube\
rt)])
...
```

To capture the arguments we have to use a different syntax for the method invocation on the mock. This time we define the method can be invoked with any number of arguments (`(*_)`) and then use a closure to capture the arguments. The arguments are passed to the closure as a list. We can then get the argument we want and write an assert statement.

```
package com.mrhaki.spock

import spock.lang.Specification

class SampleSpecification extends Specification {

    final ClassUnderTest classUnderTest = new ClassUnderTest()

    def "check sayHello is invoked with people in greeting method"() {
        given:
        final Greeting greeting = Mock()
        classUnderTest.greeting = greeting

        and:
        final List<Person> people = ['mrhakis', 'hubert'].collect { new Person(name: it\
) }

        when:
        final String greetingResult = classUnderTest.greeting(people)

        then:
        1 * greeting.sayHello(*_) >> { arguments ->
            final List<Person> argumentPeople = arguments[0]
            assert argumentPeople == [new Person(name: 'mrhaki'), new Person(name: 'hub\
ert')]
        }
    }

}
```

We run the specification again and it will fail again (of course), but this time we get an assertion message:

```
...
Condition not satisfied:

argumentPeople == [new Person(name: 'mrhaki'), new Person(name: 'hubert')]
|              |   |                          |
|              |   |                          com.jdriven.spock.Person(hubert)
|              |   com.jdriven.spock.Person(mrhaki)
|              false
[com.jdriven.spock.Person(mrhakis), com.jdriven.spock.Person(hubert)]

    at com.jdriven.spock.SampleSpecification.check sayHello is invoked with people in g\
reeting method_closure2(SampleSpecification.groovy:25)
...
```

Code written with Spock 0.7-groovy-2.0

Original blog post written on May 17, 2013.

Using Mock Method Arguments in Response

When we mock or stub methods we can use the method arguments passed to the method in the response for the mocked or stubbed method. We must write a closure after the rightShift operator (`>>`) and the closure arguments will resemble the arguments of the mocked or stubbed method. Alternatively we can use a single non-typed argument in the closure and this will contains the method argument list.

Let's create a specification where we use this feature. In the following sample we use a mock for the `AgeService` used in the class under test. The method `allowedMaxTime()` is invoked by the class under test and basically should return the maximum hour of the day a show can be broadcasted. In our specification we use the name of the show to return different values during the test.

```
package com.mrhaki.spock

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import spock.lang.*

class SampleSpec extends Specification {

    final ClassUnderTest classUnderTest = new ClassUnderTest()

    @Unroll
    def "show #name with start time 21h is #expected to show"() {
        setup:
        final AgeService ageService = Mock()
        classUnderTest.ageService = ageService

        when:
        final boolean allowed = classUnderTest.listing(
            new Show(name: name, startTime: 21)
        )

        then:
        1 * ageService.allowedMaxTime(_ as Show) >> { Show show ->
            show.name.toLowerCase().contains('kids') ? 10 : 23
        }
        // Alternative syntax with a non-typed closure argument:
        //1 * ageService.allowedMaxTime(_ as Show) >> { arguments ->
        //    arguments[0].name.toLowerCase().contains('kids') ? 10 : 23
        //}

        allowed == expected

        where:
        name            || expected
        'Kids rules'    || false
```

```
        'Sports united' || true
    }

}

/* Supporting classes and interface */

class ClassUnderTest {

    AgeService ageService

    boolean listing(final Show show) {
        final int hour = ageService.allowedMaxTime(show)
        show.startTime <= hour
    }
}

interface AgeService {
    int allowedMaxTime(Show show)
}

@groovy.transform.Canonical
class Show {
    String name
    int startTime
}
```

Code written with Spock 0.7-groovy-2.0.

Original blog post written on September 24, 2013.

Assign Multiple Data Variables from Provider

We can write data driven tests with Spock. We can specify for example a data table or data pipes in a `where:` block. If we use a data pipe we can specify a data provider that will return the values that are used on each iteration. If our data provider returns multiple results for each row we can assign them immediatelly to multiple variables. We must use the syntax `[var1, var2, var3] << providerImpl` to assign values to the data variables `var1`, `var2` and `var3`. We know from Groovy the multiple assignment syntax with parenthesis (`(var1, var2, var3)`), but with Spock we use square brackets.

In the following sample specification we have a simple feature method. The `where:` block shows how we can assign the values from the provider to multiple data variables. Notice we can skip values from the provider by using a _ to ignore the value.

```
package com.mrhaki.spock

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import spock.lang.*

class MultiDataVarSpec extends Specification {

    @Unroll("#value as upper case is #expected")
    def "check upper case value of String"() {
        expect:
        value.toUpperCase() == expected

        where:
        // Multi data variables value and expected,
        // will be filled with elements from a row
        // on each iteration. The first element of each
        // row is ignored.
        // E.g. on first iteration:
        // value = 'abc'
        // and expected = 'ABC'
        [_, value, expected] << [
            [1, 'abc', 'ABC'],
            [2, 'def', 'DEF'],
            [3, '123', '123']
        ]
    }


}
```

Code written with Spock 0.7-groovy-2.0 and Groovy 2.3.3.

Original blog post written on June 25, 2014.

Write Our Own Data Provider

We can use data pipes to write data driven tests in Spock. A data pipe (`<<`) is fed by a data provider. We can use `Collection` objects as data provider, but also `String` objects and any class that implements the `Iterable` interface. We can write our own data provider class if we implement the `Iterable` interface.

In the following sample code we want to test the `female` property of the `User` class. We have the class `MultilineProvider` that implements the `Iterable` interface. The provider class accepts a multiline `String` value and returns the tokenized result of each line in each iteration.

```
package com.mrhaki.spock

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import spock.lang.*

class ProviderSampleSpec extends Specification {

    @Unroll("Gender #gender for #name is #description")
    def "check if user is female or male based on gender value"() {
        given:
        def userData = '''\
            1;mrhaki;M;false
            2;Britt;F;true'''

        expect:
        new User(name: name, gender: gender).female == Boolean.valueOf(expected)

        where:
        [_, name, gender, expected] << new MultilineProvider(source: userData)

        // Extra data variable to be used in
        // @Unroll description.
        description = expected ? 'female' : 'not female'
    }

}

/**
 * Class under test.
 */
class User {
    String name, gender

    Boolean isFemale() {
        gender == 'F'
    }
```

```
}

/**
 * Class implements Iterable interface so
 * it can be used as data provider.
 */
class MultilineProvider implements Iterable {
    def source
    def lines
    def separator = ';'

    private int counter

    /**
     * Set multiline String as source
     * and transform to a List of String
     * values and assign to the lines
     * property.
     */
    void setSource(source) {
        this.source = source.stripIndent()
        lines = this.source.readLines()
    }

    @Override
    Iterator iterator() {
        [
            hasNext: {
                counter < lines.size()
            },
            next: {
                lines[counter++].tokenize(separator)
            }
        ] as Iterator
    }
}
```

Code written with Spock 0.7-groovy-2 and Groovy 2.3.3.

Original blog post written on June 25, 2014.

Extra Data Variables for Unroll Description

Spock's unroll feature is very powerful. The provider data variables can be used in the method description of our specification features with placeholders. For each iteration the placeholders are replaced with correct values. This way we get a nice output where we immediately can see the values that were used to run the code. Placeholders are denoted by a hash sign (`#`) followed by the variable name. We can even invoke no-argument methods on the variable values or access properties. For example if we have a String value we could get the upper case value with `#variableName.toUpperCase()`. If we want to use more complex expressions we must introduce a new data variable in the `where` block. The value of the variable will be determined for each test invocation and we can use the result as a value in the method description.

```
package com.mrhaki.spock

import spock.lang.*

class SampleSpec extends Specification {

    @Unroll
    def "check if '#value' is lower case"() {
        expect:
        value.every { (it as char).isLowerCase() } == result

        where:
        value || result
        'A'   || false
        'Ab'  || false
        'aB'  || false
        'a'   || true
        'ab'  || true
    }

}
```

If we look at the output of the tests we see the method names are not really representing the code we test. For example we can not see if the value was lower case or not.

Test	Duration	Result
check if 'A' is lower case	0.064s	passed
check if 'Ab' is lower case	0s	passed
check if 'a' is lower case	0s	passed
check if 'aB' is lower case	0s	passed
check if 'ab' is lower case	0s	passed

We rewrite the specification and add a new data variable `unrollDescription` in the `where` block. We then refer to this variable in our method name description.

```
package com.mrhaki.spock

import spock.lang.*

class SampleSpec extends Specification {

    @Unroll
    // Alternatively syntax as
    // unroll annotation argument:
    // @Unroll("'#value' is #unrollDescription")
    def "'#value' is #unrollDescription"() {
        expect:
        value.every { (it as char).isLowerCase() } == result

        where:
        value || result
        'A'   || false
        'Ab'  || false
        'aB'  || false
        'a'   || true
        'ab'  || true

        unrollDescription = result ? 'lower case' : 'not lower case'
    }

}
```

When we look at the output we now have more descriptive method names:

Test	Duration	Result
'A' is not lower case	0.090s	passed
'Ab' is not lower case	0.001s	passed
'a' is lower case	0.001s	passed
'aB' is not lower case	0s	passed
'ab' is lower case	0s	passed

This post is inspired by the great Idiomatic Spock talk by Rob Fletcher at Gr8Conf 2014 Europe.

Code written with Spock 0.7 for Groovy 2.

Original blog post written on June 16, 2014.

Ignore Specifications Based On Conditions

We can use the `@Ignore` and `@IgnoreRest` annotation in our Spock specifications to not run the annotated specifications or features. With the `@IgnoreIf` annotation we can specify a condition that needs to evaluate to `true` to not run the feature or specification. The argument of the annotation is a closure. Inside the closure we can access three extra variables: `properties` (Java system properties), `env` (environment variables) and `javaVersion`.

In the following Spock specification we have a couple of features. Each feature has the `@IgnoreIf` annotation with different checks. We can use the extra variables, but we can also invoke our own methods in the closure argument for the annotation:

```
package com.mrhaki.spock

import spock.lang.*

class SampleRequiresSpec extends Specification {

    private static boolean isOsWindows() {
        System.properties['os.name'] == 'windows'
    }

    @IgnoreIf({ Boolean.valueOf(properties['spock.ignore.longRunning']) })
    def "run spec if Java system property 'spock.ignore.longRunning' is not set or fals\
e"() {
        expect:
        true
    }

    @IgnoreIf({ Boolean.valueOf(env['SPOCK_IGNORE_LONG_RUNNING']) })
    def "run spec if environment variable 'SPOCK_IGNORE_LONG_RUNNING' is not set or fal\
se"() {
        expect:
        true
    }

    @IgnoreIf({ javaVersion < 1.7 })
    def "run spec if run in Java 1.7 or higher"() {
        expect:
        true
    }

    @IgnoreIf({ javaVersion != 1.7 })
    def "run spec if run in Java 1.7"() {
        expect:
        true
```

```
    }

    @IgnoreIf({ isOsWindows() })
    def "run only if run on non-windows operating system"() {
        expect:
        true
    }

}
```

When we run our specification with Java 1.8 and do not set the Java system property *spock.ignore.longRunning* or we set the value to `false` and we do not set the environment variable *SPOCK_IGNORE_LONG_RUNNING* or give it the value `false` we can see that some features are ignored:

Test	Duration	Result
run only if run on non-windows operating	0s	passed
run spec if Java system property 'spock.ignore.longRunning' is not set or false	0.011s	passed
run spec if environment variable 'SPOCK_IGNORE_LONG_RUNNING' is not set or false	0.001s	passed
run spec if run in Java 1.7	-	ignored
run spec if run in Java 1.7 or higher	0s	passed

Now we run on Java 1.7, Windows operating system and set the Java system property *spock.ignore.longRunning* with the value `true` and the environment variable *SPOCK_IGNORE_LONG_RUNNING* with the value `true`. The resulting report shows the specifications that are ignored and those that are executed:

Test	Duration	Result
run only if run on non-windows operating	-	ignored
run spec if Java system property 'spock.ignore.longRunning' is not set or false	-	ignored
run spec if environment variable 'SPOCK_IGNORE_LONG_RUNNING' is not set or false	-	ignored
run spec if run in Java 1.7	0s	passed
run spec if run in Java 1.7 or higher	0.013s	passed

Code written with Spock 0.7-groovy-2.

Original blog post written on June 20, 2014.

Indicate Class Under Test with Subject Annotation

If we write a specification for a specific class we can indicate that class with the `@Subject` annotation. This annotation is only for informational purposes, but can help in making sure we understand which class we are writing the specifications for. The annotation can either be used at class level or field level. If we use the annotation at class level we must specify the class or classes under test as argument for the annotation. If we apply the annotation to a field, the type of the field is used as the class under test. The field can be part of the class definition, but we can also apply the `@Subject` annotation to fields inside a feature method.

In the following example Spock specification we write a specification for the class `Greet`. The definition of the `Greet` class is also in the code listing. We use the `@Subject` annotation on the field `greet` to indicate this instance of the `Greet` class is the class we are testing here. The code also works with the `@Subject` annotation, but it adds more clarity to the specification.

```
package com.mrhaki.spock

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import spock.lang.*

// The @Subject annotation can also be applied at class level.
// We must specify the class or classes as arguments:
// @Subject([Greet])
class GreetSpec extends Specification {

    // The greet variable of type Greet is the
    // class we are testing in this specification.
    // We indicate this with the @Subject annotation.
    @Subject
    private Greet greet = new Greet(['Hi', 'Hello'])

    // Simple specification to test the greeting method.
    def "greeting should return a random salutation followed by name"() {
        when:
        final String greeting = greet.greeting('mrhaki')

        then:
        greeting == 'Hi, mrhaki' || greeting == 'Hello, mrhaki'
    }

}

/**
 * Class which is tested in the above specification.
 */
@groovy.transform.Immutable
```

```
class Greet {

    final List<String> salutations

    String greeting(final String name) {
        final int numberOfSalutations = salutations.size()
        final int selectedIndex = new Random().nextInt(numberOfSalutations)
        final String salutation = salutations.get(selectedIndex)

        "${salutation}, ${name}"
    }

}
```

Code written with Spock 0.7-groovy-2.0 and Groovy 2.3.7.

Original blog post written on October 13, 2014.

Support for Hamcrest Matchers

Spock has support for Hamcrest matchers and adds some extra syntactic sugar. In an `expect:` block in a Spock specification method we can use the following syntax *value Matcher*. Let's create a sample Spock specification and use this syntax with the Hamcrest matcher `hasKey`:

```
// File: SampleSpecification.groovy
package com.mrhaki.spock

@Grab('org.hamcrest:hamcrest-all:1.3')
import static org.hamcrest.Matchers.*

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import spock.lang.Specification

class SampleSpecification extends Specification {

    def "sample usage of hamcrest matcher hasKey"() {
        given:
        final sampleMap = [name: 'mrhaki']

        expect:
        sampleMap hasKey('name')
        sampleMap not(hasKey('name')) // To show assertion message.
    }

}
```

We can run the code (`$groovy SampleSpecification.groovy`) and see in the output a very useful assertion message for the second matcher in the `expect:` block. We directly see what went wrong and what was expected.

```
$ groovy SampleSpecification.groovy
JUnit 4 Runner, Tests: 1, Failures: 1, Time: 210
Test Failure: sample usage of hamcrest matcher hasKey(com.mrhaki.spock.SampleSpecificat\
ion)
Condition not satisfied:

sampleMap not(hasKey('name'))
|         |
|         false
[name:mrhaki]

Expected: not map containing ["name"->ANYTHING]
     but: was <{name=mrhaki}>
```

```
 at com.mrhaki.spock.SampleSpecification.sample usage of hamcrest matcher hasKey(Sample\
Specification.groovy:18)
```

With Spock we can rewrite the specification and use the static method `that()` in `spock.util.matcher.HamcrestSupport` as a shortcut for the Hamcrest `assertThat()` method. The following sample shows how we can use `that()`. With this method we can use the assertion outside an `expect:` or `then:` block.

```
// File: SampleSpecification.groovy
package com.mrhaki.spock

@Grab('org.hamcrest:hamcrest-all:1.3')
import static org.hamcrest.Matchers.*

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import static spock.util.matcher.HamcrestSupport.*
import spock.lang.Specification

class SampleSpecification extends Specification {

    def "sample usage of hamcrest matcher hasKey"() {
        given:
        final sampleMap = [name: 'mrhaki']

        expect:
        that sampleMap, hasKey('name')
    }

}
```

Finally we can use the `expect()` method in `spock.util.matcher.HamcrestSupport` to add the assertion in a `then:` block. This improves readability of our specification.

```
// File: SampleSpecification.groovy
package com.mrhaki.spock

@Grab('org.hamcrest:hamcrest-all:1.3')
import static org.hamcrest.Matchers.*

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import static spock.util.matcher.HamcrestSupport.*
import spock.lang.Specification

class SampleSpecification extends Specification {

    def "sample usage of hamcrest matcher hasKey"() {
        when:
        final sampleMap = [name: 'mrhaki']
```

```
        then:
        expect sampleMap, hasKey('name')
    }

}
```

Code written with Spock 0.7-groovy-2.0

Original blog post written on May 14, 2013.

Using a Custom Hamcrest Matcher

In a previous blog post we learned how we can use Hamcrest matchers. We can also create a custom matcher and use it in our Spock specification. Let's create a custom matcher that will check if elements in a list are part of a range.

In the following specification we create the method `inRange()` which will return an instance of a `Matcher` object. This object must implement a `matches()` method and extra methods to format the description when the matcher fails. We use Groovy's support to create a Map and turn it into an instance of `BaseMatcher`.

```
// File: SampleSpecification.groovy
package com.mrhaki.spock

@Grab('org.hamcrest:hamcrest-all:1.3')
import static org.hamcrest.Matchers.*
import org.hamcrest.*

@Grab('org.spockframework:spock-core:0.7-groovy-2.0')
import static spock.util.matcher.HamcrestSupport.*
import spock.lang.Specification

class SampleSpecification extends Specification {

    def "sample usage of custom hamcrest matcher"() {
        given:
        final list = [2,3,4]

        expect:
        that list, inRange(0..10)
        that list, not(inRange(0..3))
    }

    /**
     * Create custom Hamcrest matcher to check if a list has elements
     * that are contained in the define range.
     *
     * @param range Range to check if elements in the list are in this range.
     * @return Hamcrest matcher to check if elements in the list are part of the range.
     */
    private inRange(final range) {
        [
            matches: { list -> range.intersect(list) == list },
            describeTo: { description ->
                description.appendText("list be in range ${range}")
            },
            describeMismatch: { list, description ->
```

```
                description.appendValue(list.toListString()).appendText(" was not in ra\
nge ${range}")
            }
        ] as BaseMatcher
    }

}
```

We can run the specification (`$ groovy SampleSpecification.groovy`) and everything should work and all tests must pass.

We change the code to see the description we have added. So we change `that list, inRange(0..10)` to `that list, inRange(0..3)`. We run the specification again (`$ groovy SampleSpecification.groovy`) and look at the output:

```
JUnit 4 Runner, Tests: 1, Failures: 1, Time: 200
Test Failure: sample usage of custom hamcrest matcher(com.mrhaki.spock.SampleSpecificat\
ion)
Condition not satisfied:

that list, inRange(0..3)
|    |
|    [2, 3, 4]
false

Expected: list be in range [0, 1, 2, 3]
     but: "[2, 3, 4]" was not in range [0, 1, 2, 3]

    at com.mrhaki.spock.SampleSpecification.sample usage of custom hamcrest matcher(Sam\
pleSpecification.groovy:18)
```

Notice the output shows the text we have defined in the `describeTo()` and `describeMismatch()` methods.

Code written with Spock 0.7-groovy-0.2.

Original blog post written on May 14, 2013.