SPOCK

Testing framework

Presented By
Gajraj Kalburgi

Introduction

- Spock is a testing framework.
- Highly expressive specification language.
- Inspired by Junit, Jmock, Rspec, Groovy, Scala, Vulcans.

Spec structure

- Specs are written in groovy and extends Specification from spock.

```
class MySpec extends Specification {
    // fields
    // fixture methods
    // feature methods
    // helper methods
}
```

Fields

- Declaration of objects.
- Every feature methods gets it’s own object at runtime.

```
def obj1, obj2
```

@Shared

- It will allow to share objects with in features.
- Needs when object creation is huge task (memory / time).
- Required when we want to use fields in where block.

```
@Shared obj1, obj2
```

Fixture Methods

Spock	Junit
def setup	@Before
def cleanup	@After
def setupSpec	@BeforeClass
def cleanupSpec	@AfterClass

Feature Methods

- Feature methods are the heart of a specification.
- Conceptually, a feature method consists of four phases:
  1. Set up the feature's fixture
  2. Provide a *stimulus* to the system under specification
  3. Describe the *response* expected from the system
  4. Clean up the feature's fixture
- Above phases are described using blocks.

```
def “Feature name”() {
    // blocks
}
```

Blocks

`setup:` Setup feature data, object state, define mock object behavior.

`given:` Alias for setup block.

`when:` Stimulation(actual call, can be void call)

`then:` Assertion, Condition testing

`expect:` It's combination of when and then block (Make actual call and do result assertion, Function must return some value for assertion)

`cleanup:` Cleanup feature data, object state

`where:` Streamline data used in feature, Feature will be executed multiple time depended on where block data.

Helper Methods

- Helper methods used to write common logic for features.
- It can have conditions for asserting in then block.
- It can have mock object behavior to use in given or setup block.
- It can be used to give name for common code block.

Data driven testing

Data table

```
def “Math.max”() {
    expect:
        Math.max(a, b) == c
    where:
        a  | b  || c
        8  | 5  || 8
        10 | 14 || 14
}
```

Data driven testing

Data pipe

```
def “Math.max”() {
    expect:
        Math.max(a, b) == c
    where:
        a  << [8, 10, 25]
        b  << [3, 23, 18]
        c  << [8, 23, 25]
}
```

@Unroll

- It’s defines how feature is reported.
  - Each data defined in where will be reported as separate test.
- It allows dynamic naming for features.
  - We can use members used in features to change feature name.

```
@Unroll
def “Math.max(#a, #b) == #c”() {
    expect:
        Math.max(a, b) == c
    where:
        a  | b  || c
        8  | 5  || 8
        10 | 14 || 14
}
```

Extensions

`@Ignore` : Ignores feature.

`@IgnoreRest` : Ignores all features except this. (useful to test only one feature.

`@IgnoreIf` : Ignore feature with condition.

`@Require` : Ignore feature with condition (Inverted @IgnoreIf)

`@PendingFeature` : To suggest feature is pending. (Skips test when at least one iteration fails, If all iteration fails test will be reported as failed.)

`@Stepwise` : Execute all feature in a sequence as they written. If any test fails it will skip remaining tests in class.

`@Timeout` : Timeout time for feature to execute.

`@FailsWith` : Some feature fails due to bugs in production code. This annotation will abruptly complete execution.

```
@FailsWith("XYZ api not working now")
```

Specification

```
thrown             : Asserts that exception thrown.
notThrown          : Asserts that specified exception is not thrown.
noExceptionThrown  : Asserts that no exception is thrown.
old                : Used to get field value before when block’s execution.
with               : To make multiple condition/operation with object.
verifyAll          : Asserts all conditions.
```

Mock

Mock factory method creates mock object.

```
class MySpec extends Specification {

      def mockObj = Mock(MockClass)
      def targetObj = new TargetClass(mockObj)

      def “Hello World Feature”() {
            given: “Hello World Feature”
                  mockObj.mockFunction(“Hello”) >> “Hello World !!”
            when: “Hello World Feature”
                  def result = targetObj.call(“Hello”)
            then: “Hello World Feature”
                   result == “Hello World !!”
      }
}
```

Stub

We can define mock object behavior in setup or given block. It’s called stubbing.

```
given:
    mockobject.function(“Hello”) >> “Hello world !”
    mockobject.function(_ as String) >> [“Hello”,“world !”]
    mockobject._(_) >> { throw new Exception() }
    mockobject./f.*n/(!null) >>> [“ONE”,“TWO”] >> {throw new Exception() } >> “ALL”
    mockobject.function({ it.size() > 3 })
    mockobject.function({ [“hello”].contains(it) })
```

Verify

We can verify mock interactions in then block.

```
then:
    1 * mockobject.function(“hello”)
    (1..2) * _.function(_ as String)
    (_..5) * mockobject._(_)
    (2.._) * mockobject./f.*n/(!null)
    _ * mockobject.function({ it.size() > 3 })
    0 * mockobject.function({ [“hello”].contains(it) })
```

Combine Stub and Verify

```
given:
    1 * mockobject.function(“Hello”) >> “Hello world !”
    (1..2) * mockobject.function(_ as String) >> [“Hello”,“world !”]
    (_..5) * mockobject._(_) >> { throw new Exception() }
    (2.._) * mockobject./f.*n/(!null) >>> [“ONE”,“TWO”] >> {throw new Exception() } >> “ALL”
    _ * mockobject.function({ it.size() > 3 })
    0 * mockobject.function({ [“hello”].contains(it) })
```

Hamcrest Support

We can use hamcrest for assert.

```
then:
HamcrestSupport.that(result,
IsIterableContaininInAnyOrder.containsInAnyOrder( element1, element2 ))
```