Data Driven Testing

Oftentimes, it is useful to exercise the same test code multiple times, with varying inputs and expected results. Spock's data driven testing support makes this a first class feature.

Introduction

Suppose we want to specify the behavior of the `Math.max` method:

```
class MathSpec extends Specification {
    def "maximum of two numbers"() {
        expect:
        // exercise math method for a few different inputs
        Math.max(1, 3) == 3
        Math.max(7, 4) == 7
        Math.max(0, 0) == 0
    }
}
```

Although this approach is fine in simple cases like this one, it has some potential drawbacks:

- Code and data are mixed and cannot easily be changed independently
- Data cannot easily be auto-generated or fetched from external sources
- In order to exercise the same code multiple times, it either has to be duplicated or extracted into a separate method
- In case of a failure, it may not be immediately clear which inputs caused the failure
- Exercising the same code multiple times does not benefit from the same isolation as executing separate methods does

Spock's data-driven testing support tries to address these concerns. To get started, let's refactor above code into a data-driven feature method. First, we introduce three method parameters (called *data variables*) that replace the hard-coded integer values:

```
class MathSpec extends Specification {
    def "maximum of two numbers"(int a, int b, int c) {
        expect:
        Math.max(a, b) == c

        ...
    }
}
```

We have finished the test logic, but still need to supply the data values to be used. This is done in a `where:` block, which always comes at the end of the method. In the simplest (and most common) case, the `where:` block holds a *data table*.

Data Tables

Data tables are a convenient way to exercise a feature method with a fixed set of data values:

```
class Math extends Specification {
    def "maximum of two numbers"(int a, int b, int c) {
        expect:
        Math.max(a, b) == c

        where:
        a | b | c
        1 | 3 | 3
        7 | 4 | 4
        0 | 0 | 0
    }
}
```

The first line of the table, called the *table header*, declares the data variables. The subsequent lines, called *table rows*, hold the corresponding values. For each row, the feature method will get executed once; we call this an *iteration* of the method. If an iteration fails, the remaining iterations will nevertheless be executed. All failures will be reported.

Data tables must have at least two columns. A single-column table can be written as:

```
where:
a | _
1 | _
7 | _
0 | _
```

Isolated Execution of Iterations

Iterations are isolated from each other in the same way as separate feature methods. Each iteration gets its own instance of the specification class, and the `setup` and `cleanup` methods will be called before and after each iteration, respectively.

Sharing of Objects between Iterations

In order to share an object between iterations, it has to be kept in a `@Shared` or static field.

> **Note**
>
> Only `@Shared` and static variables can be accessed from within a `where:` block.

Note that such objects will also be shared with other methods. There is currently no good way to share an object just between iterations of the same method. If you consider this a problem, consider putting each method into a separate spec, all of which can be kept in the same file. This achieves better isolation at the cost of some boilerplate code.

Syntactic Variations

The previous code can be tweaked in a few ways. First, since the `where:` block already declares all data variables, the method parameters can be omitted. [1]. Second, inputs and expected outputs can be separated with a double pipe symbol (`||`) to visually set them apart. With this, the code becomes:

```
class DataDriven extends Specification {
     def "maximum of two numbers"() {
         expect:
         Math.max(a, b) == c

         where:
         a | b || c
         3 | 5 || 5
         7 | 0 || 7
         0 | 0 || 0
     }
 }
```

Reporting of Failures

Let's assume that our implementation of the `max` method has a flaw, and one of the iterations fails:

```
maximum of two numbers   FAILED
```

```
Condition not satisfied:

Math.max(a, b) == c
     |     |  |  |  |
     |     7  0  |  7
     42          false
```

The obvious question is: Which iteration failed, and what are its data values? In our example, it isn't hard to figure out that it's the second iteration that failed. At other times this can be more difficult or even impossible [2]. In any case, it would be nice if Spock made it loud and clear which iteration failed, rather than just reporting the failure. This is the purpose of the `@Unroll` annotation.

Method Unrolling

A method annotated with `@Unroll` will have its iterations reported independently:

```
@Unroll
def "maximum of two numbers"() { ... }
```

Note that unrolling has no effect on how the method gets executed; it is only an alternation in reporting. Depending on the execution environment, the output will look something like:

```
maximum of two numbers[0]   PASSED
maximum of two numbers[1]   FAILED

Math.max(a, b) == c
     |     |  |  |  |
     |     7  0  |  7
     42          false

maximum of two numbers[2]   PASSED
```

This tells us that the second iteration (with index 1) failed. With a bit of effort, we can do even better:

```
@Unroll
def "maximum of #a and #b is #c"() {
```

This method name uses placeholders, denoted by a leading hash sign (#), to refer to data variables `a`, `b`, and `c`. In the output, the placeholders

Why isn't `@Unroll` the default?

One reason why `@Unroll` isn't the default is that some execution environments (in particular IDEs) expect to be told the number of test methods in advance, and have certain problems if the actual number varies. Another reason is that `@Unroll` can drastically change the number of reported tests, which may not always be desirable.

will be replaced with concrete values:

```
maximum of 3 and 5 is 5   PASSED
maximum of 7 and 0 is 7   FAILED

Math.max(a, b) == c
     |     |  |  |  |
     |     7  0  |  7
     42          false

maximum of 0 and 0 is 0   PASSED
```

Now we can tell at a glance that the `max` method failed for inputs `7` and `0`. See More on Unrolled Method Names for further details on this topic.

The `@Unroll` annotation can also be placed on a spec. This has the same effect as placing it on each data-driven feature method of the spec.

Data Pipes

Data tables aren't the only way to supply values to data variables. In fact, a data table is just syntactic sugar for one or more *data pipes*:

```
...
where:
a << [3, 7, 0]
b << [5, 0, 0]
c << [5, 7, 0]
```

A data pipe, indicated by the left-shift (`<<`) operator, connects a data variable to a *data provider*. The data provider holds all values for the variable, one per iteration. Any object that Groovy knows how to iterate over can be used as a data provider. This includes objects of type `Collection`, `String`, `Iterable`, and objects implementing the `Iterable` contract. Data providers don't necessarily have to *be* the data (as in the case of a `Collection`); they can fetch data from external sources like text files, databases and spreadsheets, or generate data randomly. Data providers are queried for their next value only when needed (before the next iteration).

Multi-Variable Data Pipes

If a data provider returns multiple values per iteration (as an object that Groovy knows

how to iterate over), it can be connected to multiple data variables simultaneously. The syntax is somewhat similar to Groovy multi-assignment but uses brackets instead of parentheses on the left-hand side:

```
@Shared sql = Sql.newInstance("jdbc:h2:mem:", "org.h2.Driver"

def "maximum of two numbers"() {
    ...
    where:
    [a, b, c] << sql.rows("select a, b, c from maxdata")
}
```

Data values that aren't of interest can be ignored with an underscore (_):

```
...
where:
[a, b, _, c] << sql.rows("select * from maxdata")
```

Data Variable Assignment

A data variable can be directly assigned a value:

```
...
where:
a = 3
b = Math.random() * 100
c = a > b ? a : b
```

Assignments are re-evaluated for every iteration. As already shown above, the right-hand side of an assignment may refer to other data variables:

```
...
where:
row << sql.rows("select * from maxdata")
// pick apart columns
a = row.a
b = row.b
c = row.c
```

Combining Data Tables, Data Pipes, and Variable Assignments

Data tables, data pipes, and variable assignments can be combined as needed:

```
...
where:
a | _
3 | _
7 | _
0 | _

b << [5, 0, 0]

c = a > b ? a : b
```

Number of Iterations

The number of iterations depends on how much data is available. Successive executions of the same method can yield different numbers of iterations. If a data provider runs out of values sooner than its peers, an exception will occur. Variable assignments don't affect the number of iterations. A `where:` block that only contains assignments yields exactly one iteration.

Closing of Data Providers

After all iterations have completed, the zero-argument `close` method is called on all data providers that have such a method.

More on Unrolled Method Names

An unrolled method name is similar to a Groovy `GString`, except for the following differences:

- Expressions are denoted with `#` instead of `$` [3], and there is no equivalent for the `${...}` syntax
- Expressions only support property access and zero-arg method calls

Given a class `Person` with properties `name` and `age`, and a data variable `person` of type `Person`, the following are valid method names:

```
def "#person is #person.age years old"() { ... } // property
def "#person.name.toUpperCase()"() { ... } // zero-arg method
```

Non-string values (like `#person` above) are converted to Strings according to Groovy semantics.

The following are invalid method names:

```
def "#person.name.split(' ')[1]" { ... } // cannot have metho
def "#person.age / 2" { ... } // cannot use operators
```

If necessary, additional data variables can be introduced to hold more complex expression:

```
def "#lastName"() {
    ...
    where:
    person << ...
    lastName = person.name.split(' ')[1]
}
```

Footnotes

[1] The idea behind allowing method parameters is to enable better IDE support. However, recent versions of IntelliJ IDEA recognize data variables automatically, and even infer their types from the values contained in the data table.

[2] For example, a feature method could use data variables in its `setup:` block, but not in any conditions.

[3] Groovy syntax does not allow dollar signs in method names.