Making Shell Scripts Groovy

Georg Berky, Valtech Mobility

Groovy ist eine dynamische Sprache auf der JVM und glänzte schon einige Jahre vor Java mit Features wie funktionaler Programmierung oder Literals für oft benutzte Datenstrukturen wie `List` *und* `Map`*. Weniger bekannt ist, dass Groovy auch mächtige Werkzeuge zum Arbeiten auf der Kommandozeile mitbringt, mit denen man – allein durch Kenntnis von Java – schon eigene Skripte schreiben kann, ohne sich gleich auf die komplizierte Syntax mancher Shells einlassen zu müssen.*

Getting Groovy – Installation und Setup

Der einfachste Installationsweg für Groovy ist `sdkman`, ein CLI-Tool, mit dem man eine Menge SDKs installieren und zwischen ihren Versionen hin- und herschalten kann. Der Installationsvorgang von `sdkman` wird in *Listing 1* gezeigt. Die Installation von Groovy ist in *Listing 2* zu finden.

A Groovy Kind of Java – Erste Schritte

Der erste Schritt in einer neuen Programmiersprache ist in unserer Zunft das „Hello World". Groovy ist eine syntaktische Obermenge von Java. Das bedeutet, dass wir einfach Java-Code schreiben können und es dadurch schon valides Groovy ist *(siehe Listing 3)*. Statt `.java` verwenden wir `.groovy` als Dateiendung *(siehe Listing 4)*.

```
curl -s "https://get.sdkman.io" | bash
```

Listing 1: sdkman installieren

```
sdk install groovy
```

Listing 2: Groovy installieren

Groovy wäre nicht „groovy", wenn es uns hier nicht einiges an Schreibarbeit abnehmen würde: Wir brauchen weder die Definition der Klasse noch die Main-Methode, sondern können einfach schreiben, als wären wir direkt in „main". Beide werden von Groovy generiert. Semikola können wir ebenfalls weglassen *(siehe Listing 5)*.

Um unser Skript zu einem First-Class-Citizen auf der Shell zu machen, muss es sich in die Unix-Philosophie der kleinen, zusammensetzbaren Programme einfügen. Dazu muss es wie andere Programme ausführbar sein und Daten über Pipes als Input akzeptieren.

Ausführbarkeit bekommen wir, indem wir eine Shebang `#!` an den Anfang unseres Skripts setzen. Damit signalisieren wir der Shell, dass sie einen neuen Prozess mit dem hinter der Shebang angegebenen Interpreter starten soll. Der Rest der Datei wird zum Input für den Interpreter *(Listing 6)*.

Danach machen wir die Datei noch für die Shell ausführbar. Um gänzlich zu verbergen, dass wir es mit Groovy zu tun haben, können wir noch die Dateiendung entfernen. Wir brauchen sie nicht mehr, wenn unser Skript mit einer Shebang anfängt *(Listing 7)*.

Pipes

Wir sind jetzt so weit, dass wir unsere Skripte auf der Kommandozeile starten können, aber wie interagieren wir mit anderen Prozessen auf der Shell? Das Unix-Bordmittel der Wahl sind Pipes.

Eigenschaften von Pipes

Pipes sind zeilengepufferte Kanäle zur Interprozesskommunikation. Sie verhalten sich wie reguläre Unix-Dateien mit einem schreibbaren und einem lesbaren Ende. Beide Enden haben einen eigenen Dateideskriptor. Zeilen, die am schreibbaren Ende geschrieben werden, kommen in derselben Reihenfolge am lesbaren Ende auch wieder aus der Pipe (FIFO: „First In, First Out").

Pipes haben eine interne Kapazität. Schreibt man zu viel, blockiert der Schreibvorgang, bis am anderen Ende gelesen wird. Ebenso blockieren lesende Operationen, wenn die Pipe leer ist.

Durch die Zeilenpufferung kann man aus einer Pipe erst lesen, wenn das Schreiben einer Zeile mit `\n` abgeschlossen wurde. Wenn sich der letzte schreibende Prozess von der Pipe trennt, empfangen die Konsumenten ein `EOF` (End of File). Auch das schließt die Verarbeitung einer Zeile ab.

Die Standard-Pipes: stdin, stdout, stderr

Die bekanntesten Pipes sind `stdin`, `stdout` und `stderr`. Bevor wir ein Programm starten, ist `stdin` auf der schreibbaren Seite der Pipe mit unserem Keyboard verbunden. Das Gegenstück `stdout` ist auf der lesbaren Seite mit dem Terminal verbunden, ebenso wie `stderr`, das zur Ausgabe von Fehlermeldungen getrennt vom normalen Output verwendet wird. Zwischen `stdin` und `stdout`/`stderr` sitzt verarbeitend die interaktive Shell.

Pipes und Groovy

In der JVM ist `stdin` mit dem `InputStream` unter `System.in` verbunden. Für die Ausgabe sind `stdout` und `stderr` mit einem `PrintStream` unter `System.out` und `System.err` verbunden.

```
public class Hello {
    public static void main(String [] args) {
        System.out.println("Hello World!");
    }
}
```

Listing 3: Hello World in Java und Groovy

```
➜ groovy HelloWorld.groovy
Hello World!
```

Listing 4: Hello World mit Groovy starten

```
println "Hello World!"
```

Listing 5: Hello World in Groovy

```
#!/usr/bin/env groovy
println "Hello World!"
```

Listing 6: Hello World mit Shebang

```
➜ chmod +x HelloWorld.groovy
➜ mv HelloWorld.groovy hello_world
➜ ./hello_world
Hello World!
```

Listing 7: Hello World ausführbar auf der Shell

Wir wollen als erstes Beispiel grob die Funktionalität von `grep` nachbauen. Unsere Eingabedatei `README.md` hat den in *Listing 8* angegebenen Inhalt.

Gebaut für die Unendlichkeit

Pipes können unendlich lange Streams von Daten sein. Wir wählen einen funktionalen Ansatz, um nicht in Speicherprobleme durch Zwischenspeichern zu laufen *(siehe Listing 9)*.

Die Methode `eachLine` ist eine Erweiterung des API von `InputStream` und gibt uns den Inhalt des Streams Zeile für Zeile. Die Variable `it` kommt ebenfalls von Groovy. Sie ist der Standardname bei Closures, die nur einen Parameter haben.

Dadurch, dass wir die Ergebnisse weiter nach `stdout` schreiben, können andere Programme auf der Shell diese ohne zusätzlichen Aufwand weiterverarbeiten, wie in *Listing 10* gezeigt wird. Hier geben wir die Ausgabe unseres Skripts mit einer Pipe an das Unix-Programm `wc` weiter, das mit dem Schalter `-l` die Anzahl der Zeilen zählt, die es liest. Die Eingabe bekommen wir über den Umleitungsoperator `<` für `stdin` aus der Datei `README.md`. Das funktioniert, weil sich Pipes wie reguläre Dateien verhalten. Alle Groovy-Erweiterungen des JDK können übrigens unter *[2]* nachgelesen werden.

Ausblick: Named Pipes

Will man mehr als zwei Prozesse miteinander kommunizieren lassen, verwendet man dafür normalerweise „Named Pipes". Details zu dieser Thematik sind im Blog des Autors zu finden.

```
# groovy-cli #
Examples demonstrating how to use groovy on the command line
This is a line without our keyword
this line is groovy
```

Listing 8: Eingabedatei für den Nachbau von grep

```
#!/usr/bin/env groovy
System.in.eachLine {
    if(it.contains("groovy")) {
        println it
    }
}
```

Listing 9: Unser Nachbau von grep

```
➜ ./3_pipes.groovy < README.md | wc -l
3
```

Listing 10: Interaktion mit anderen Shell Tools

```
.
├── hello
│   └── Hello.groovy
└── hello.groovy
```

Listing 11: Repository mit einem Package

```
#!/usr/bin/env groovy
import hello.Hello
def hello = new Hello()
hello.doSomething()
```

Listing 12: Klasse im Package verwenden

```
.
├── src
│   └── bar
│       └── Foo.groovy
└── test
    └── bar
        └── FooTest.groovy
```

Listing 13: Getrennte Test- und Produktionsklassen

```
➜ groovy -cp test:src test/bar/FooTest.groovy
JUnit5 launcher: passed=1, failed=0, skipped=0, time=51ms
```

Listing 14: Test- und Produktionscode auf dem Classpath

```
#!/usr/bin/env groovy
import org.junit.jupiter.api.Test

class MyTest {

    @Test
    void "simple JUnit 5 Test"() {
        assert 1 + 1 == 3
    }
}
```

Listing 15: Erster Test mit Groovy und JUnit 5

Test Driven Groovy Scripting

Skripte werden oft für Massen-Tasks oder kritische Administration verwendet. In solchen Fällen ist es sinnvoll, ihre Logik zu testen. Als Simulation echter Logik wollen wir ein Skript schreiben, das Zeilen aus einer Pipe liest, diese als Zahl interpretiert und sie inkrementiert. Dies wollen wir testen – einmal in einem isolierten und einmal in einem integrierten Test *[3]*. Groovy bringt dafür bereits viele Werkzeuge mit. Bevor wir mit dem Testing loslegen, möchte ich zunächst die grundlegenden Hilfsmittel für unsere Tests vorstellen.

Source Code organisieren

Die meisten meiner Skripte sind recht klein und gehen nicht über wenige Klassen hinaus. Ab einer gewissen Größe kann es jedoch sinnvoll sein, den Code in Packages aufzuteilen.

Packages

Angenommen, mein Repository sähe so aus, wie in *Listing 11* gezeigt, dann könnte ich mit diesem Skript-Layout `hello.groovy` auf die Klasse `Hello`, wie in *Listing 12* beschrieben, zugreifen.

Tests von Produktionsklassen trennen

Wenn das Repository größer wird, lohnt es sich, Test- und Produktionscode zu trennen, um nicht aus Versehen Test-Code in Produktionsklassen zu verwenden.

Unser Produktionscode liegt in Packages unter `src`. Wie in Maven und Gradle bekommt jedes Package ein eigenes Verzeichnis. Unter-Packages werden zu Unterverzeichnissen. Mit den Tests verfahren wir genauso.

Angenommen unser Repository hätte den Inhalt, der in *Listing 13* zu sehen ist, dann können wir den Test **aus dem Root-Verzeichnis** des Repository starten *(siehe Listing 14)*. Der Aufruf mit `-cp path1:path2` sagt Groovy, wo nach Klassen gesucht werden soll. Wenn wir testen, brauchen wir sowohl den Baum unter `test` als auch den unter `src`. Wenn wir nur Produktionscode starten wollen, nehmen wir `test` einfach nicht in den Classpath auf. So ist dann sichergestellt, dass sich kein Test-Code unter den Produktionscode gemischt hat.

Batteries Included: JUnit + Power Assertions

Groovy bringt inzwischen ganze drei Versionen von JUnit mit: 3, 4 und 5. Man muss keine zusätzlichen Bibliotheken einbinden, sondern kann sofort loslegen. Wir beschränken uns hier auf JUnit 5.

JUnit 5

Auf der Shell ist der einzige Unterschied zu regulären JUnit-Tests, dass wir die bekannte Shebang an den Anfang der Datei setzen müssen. Danach folgen die bekannten Imports und eine Testklasse mit den Annotations von JUnit 5 *(siehe Listing 15)*. Groovy erlaubt Strings als Methodennamen. Dies machen wir uns bei Tests zunutze, um besonders sprechende Namen zu vergeben.


ORAWORLD
Das e-Magazine für alle Oracle-Anwender!
EOUC
EMEA ORACLE USERGROUP COMMUNITY
• Spannende Geschichten aus der Oracle-Welt
• Technologische Hintergrundartikel
• Leben und Arbeiten heute und morgen
• Einblicke in andere User Groups weltweit
• Neues (und Altes) aus der Welt der Nerds
• Comics, Fun Facts und Infografiken
Jetzt Artikel einreichen oder Thema vorschlagen!
Jetzt e-Magazine herunterladen
www.oraworld.org

```
➜ ./5_junit5.groovy
JUnit5 launcher: passed=0, failed=1, skipped=0, time=51ms

Failures (1):
  JUnit Jupiter:MyTest:simple JUnit 5 Test()
    MethodSource [className = 'MyTest', methodName = 'simple JUnit 5 Test', methodParameterTypes = '']
    => Assertion failed:

assert 1 + 1 == 3
         |   |
         2   false
```

Listing 16: Fehlgeschlagener Test

Wenn wir den Test starten, bekommen wir eine Fehlermeldung *(Listing 16)*. Die sprechenden Fehlermeldungen sind ein Feature von Groovy, die „Power Assertions". Wir sehen alle Teile der fehlgeschlagenen Assertion auf einen Blick: die ganze Expression, ihre Teile und das Ergebnis ihrer (Teil-)Auswertungen. Bei komplexeren Evaluationen hilft uns das enorm bei der Analyse des Problems.

Stubs und Mocks

Groovy kann sowohl interpretiert als auch kompiliert ausgeführt werden. Skripte und auch Tests in Skripten werden vorkompiliert. Deswegen können sie dynamische Features wie `StubFor` und `MockFor` nicht verwenden. Stattdessen werden wir das Feature „Map Coercion" benutzen, das uns erlaubt, Maps mit dem Coercion-Operator `as` zu Implementierungen von Interfaces zu machen. Dies funktioniert auch im kompilierten Kontext. Jetzt haben wir alle Hilfsmittel zusammen, um mit dem Testing zu starten.

Unit-Tests

Unit-Tests sollen isoliert von der Außenwelt laufen und nichts von Dateien, Netzwerk, Datenbanken oder – frei nach Robert Martin – anderen „Implementierungsdetails" *[4]* aus der Außenwelt unseres Systems wissen. Um unsere Komponenten isoliert testbar zu machen, müssen wir das Design unseres Skripts entsprechend wählen. Zum Schluss werden wir das ganze System in einem integrierten Test prüfen.

Incrementer

Die reine Logik unseres Skripts beschränken wir deswegen darauf, eine Zahl zu nehmen und die inkrementierte Zahl zurückzuliefern. Woher die Zahl kommt, ist irrelevant. Die Außenwelt ist ein Implementierungsdetail. Das entkoppelt das Holen der Zahl vom Inkrementieren. Wir beachten das Single Responsibility Principle und unser Skript wird dadurch ohne einen anderen Prozess oder eine Datei als Input für den Stream testbar *(Listing 17)*. Auch TDD funktioniert auf der Kommandozeile gut. Getrieben von unseren Tests sieht die Logik der Produktionsklasse zum Schluss wie in *Listing 18* beschrieben aus.

Sie haben vielleicht bemerkt, dass ich keine Shebang an die Klassen geschrieben habe, obwohl dies möglich wäre. Shebangs schreibe ich normalerweise nur an den Einstiegspunkt unseres Systems, um diesen als solchen zu markieren. Den Test starte ich auf traditionelle Weise *(siehe Listing 19)*.

Deserializer

Der Deserialisierer soll eine Zeile lesen, die Zeile in eine Zahl umwandeln und diese an den Incrementer weitergeben. Letzteres werden wir durch einen Interaktionstest gegen einen Mock verifizieren. Die Zeile selbst wird später aus der Pipe kommen. Den Mock erzeugen wir durch die oben erwähnte Map Coercion: Wir weisen dem Key `increment` der Map eine Closure zu. Durch die Coercion der Map wird sie zu einer Interface-Implementierung mit einer Methode `increment(int number)`.

Der Mock nimmt das übergebene Argument an und speichert es im Field `receivedDeserialized`. Dieses prüfen wir in unseren Tests. Wenn nichts gespeichert wird, bleibt das Field `null` und ist damit nach „Groovy Truth" `false` *(siehe Listing 20)*. Der Produktionscode sieht getrieben von unseren Tests so aus, wie in *Listing 21* gezeigt.

```
import org.junit.jupiter.api.*

class IncrementerTest {

    private def sut = new Incrementer()

    @Test
    void "increment 1"() {
        assert sut.increment(1) == 2
    }

    //gleicher Test für 2 und 3 ...

}
```

Listing 17: Tests für den Incrementer

```
class Incrementer {
    public int increment(int number) {
        return number + 1
    }
}
```

Listing 18: Testgetriebener Produktionscode des Incrementer

```
➜ groovy IncrementerTest.groovy
JUnit5 launcher: passed=3, failed=0, skipped=0, time=13ms
```

Listing 19: Tests des Incrementer starten

```
import org.junit.jupiter.api.*

class DeserializerTest {

    private def receivedDeserialized

    private def incrementerMock
    private def sut

    @BeforeEach
    void "setup"() {
        //map coercion!
        incrementerMock = [increment: { int number ->
            receivedDeserialized = number }] as Incrementer

        sut = new Deserializer(incrementerMock)
    }

    @Test
    void "deserialize 1 on one line"() {
        def line = "1\n"

        sut.deserialize(line)

        assert receivedDeserialized == 1
    }

    //gleicher Test für 2 und 3...

    @Test
    void "deserialize 1 on one line without newline"() {
        def line = "1"

        sut.deserialize(line)

        assert receivedDeserialized == 1
    }

    @Test
    void "deserialize non-numbers does not call increment"() {
        def garbage = "lkajdfoiaer"

        sut.deserialize(garbage)

        assert !receivedDeserialized //groovy truth
    }
}
```

Listing 20: Testcode Deserializer

```
class Deserializer {

    private def incrementer

    Deserializer(Incrementer incrementer) {
        this.incrementer = incrementer
    }

    Integer deserialize(String line) {
        try {
            def deserialized = Integer.parseInt(line?.trim())

            return incrementer.increment(deserialized)
        }
        catch(NumberFormatException e) {
            System.err.println("Malformed message: '${line}'")
        }
    }
}
```

Listing 21: Produktionscode Deserializer

```
#!/usr/bin/env groovy
import java.util.stream.*

def numbers = Stream.iterate(0G) { i ->
    i.add(1G)
}

numbers.each {
    println it
    sleep 1000
}
```

Listing 22: Produktionscode Producer

```
import org.junit.jupiter.api.*

class PipeConsumerTest {

    @Test
    void "consumer increments text from stdin"() {
        def process = "./pipe_consumer.groovy".execute()
        def stdin = new PrintStream(
            process.getOutputStream(), true) //autoFlush
        def stdout = process.getInputStream()

        stdin.println("1")
        stdin.println("2")
        stdin.println("3")
        stdin.close() //EOF

        assert stdout.readLines() == ["2", "3", "4"]
    }
}
```

Listing 23: Tests des Consumer

```
#!/usr/bin/env groovy
def deserializer = new Deserializer(new Incrementer())

System.in.eachLine {
   println deserializer.deserialize(it)
}
```

Listing 24: Produktionscode Consumer

```
➜ ./pipe_producer.groovy | ./pipe_consumer.groovy
1
2
3
...
```

Listing 25: Producer und Consumer zusammen starten

Die Fehler geben wir auf `System.err` aus. Damit können wir sie – falls nötig – gesondert weiterverarbeiten. Sie landen also nicht in den Pipes, die wir mit dem `System.out` unseres Skripts verbinden.

Producer

Der Producer produziert durch `BigInteger` und `Stream.iterate()` einen potenziell unendlich langen Stream von Zahlen, die er direkt nach `System.out` schreibt. Das `G` hinter den Zahlen macht sie zu `BigInteger` *(siehe Listing 22)*. Da der Producer unendlich viele Zahlen produziert, ist es nicht möglich, ihn direkt zu testen.

Consumer

Um den Consumer integriert zu testen, müssen wir seine Eingaben kontrollieren und seine Ausgaben verifizieren können. Groovy hat das API von `String` um die Methode `execute()` erweitert. Diese führt den Befehl im String in einer Instanz von `Process` aus, auf der wir `stdin` und `stdout` setzen können.

Um in die Eingabe bequem schreiben zu können, verwenden wir einen `PrintStream`. Die Ausgabe lesen wir mit der Methode `readLines()` aus dem Ausgabe-Stream des Prozesses. Auch diese Methode ist eine Erweiterung von Groovy *(siehe Listing 23)*.

Nach dem Aufruf von `execute()` läuft der Prozess eigentlich schon. Da er aber keine Eingaben über `stdin` bekommt, blockiert das Lesen und wir müssen uns nicht darum kümmern, den Prozess erst dann zu starten, wenn wir in sein `stdin` geschrieben haben – ein weiterer Vorteil des Arbeitens mit Pipes.

Die Bezeichnungen der Methoden können anfangs etwas verwirrend sein: Um das `stdin` des Prozesses zu erhalten, muss man `getOutputStream` aufrufen. Die Blickrichtung ist hier aus der Sicht des aufrufenden Prozesses. Gleiches gilt in Gegenrichtung für `stdout` und `getInputStream`. Der Consumer selbst ist einfach gebaut *(siehe Listing 24)*. Der gemeinsame Start von Producer und Consumer ist in *Listing 25* zu sehen.

Dependencies

Das letzte und gleichzeitig mächtigste Werkzeug in unserem Scripting-Koffer ist Groovys Dependency Management Tool „Grape". Es erlaubt uns, zur Laufzeit Bibliotheken nachzuladen und sie in unsere Skripte einzubinden.

Kernstück von Grape ist die Annotation `@Grab`, die als Parameter entweder (wie in Gradle) einen String oder (wie in Maven) die drei Teile der aus Maven bekannten GAV-Parameter nimmt, das Artefakt hinter den Parametern aus Maven Central oder einem

```
//alternativ: @Grab('org.apache.commons:commons-lang:3.8.1')
@Grab(
    group='org.apache.commons',
    module='commons-lang3',
    version='3.8.1')
import org.apache.commons.lang3.*

println StringUtils.joinWith(" ", "Hello", "Java", "Aktuell")
```

Listing 26: Dependencies einbinden

konfigurierten Repository zwischenspeichert und es zur Laufzeit auf den Classpath legt.

In *Listing 26* binden wir die Bibliothek `commons-lang3` aus der Gruppe `org.apache.commons` in Version `3.6` als Dependency ein, um daraus die Klasse `StringUtils` verwenden zu können.

`@Grab` kann überall verwendet werden, wo Annotations erlaubt sind, also insbesondere an Imports, Fields und Klassen. Man kann beliebig viele `@Grab` hintereinander verwenden. Wir können jetzt jede beliebige Bibliothek aus dem JVM-Ökosystem in unseren Skripten verwenden.

Fazit

Groovy ist ein mächtiges Werkzeug für Skripte auf der Kommandozeile. Durch seine starke Ähnlichkeit zu Java hat es eine sehr flache Lernkurve und man kann auch als Shell-Neuling schnell Skripte in einer für Java-Teams les- und wartbaren Sprache entwickeln. Die Skripte sind durch die in Groovy eingebauten Werkzeuge leicht zu testen und können sich mit Grape auch die große Anzahl von Bibliotheken im JVM-Ökosystem zunutze machen.

Quellen und Referenzen

[1] Georg Berky (2018): „Groovy Shell Scripting - Pipes und FIFOs". Never Code Alone

[2] *http://groovy-lang.org/gdk.html*

[3] J.B. Rainsberger (2009): „Integrated tests are a scam". The Code Whisperer

[4] Robert C. Martin (2012): „NO DB". Clean Coder Blog

Georg Berky

georg.berky@valtech-mobility.com

Georg Berky ist leidenschaftlicher Programmierer, hauptsächlich unterwegs in JVM-Sprachen wie Java, Groovy oder Clojure. Seine ersten Schritte hat er in QBASIC, dann unter Linux auf der Kommandozeile und mit C gemacht und konnte bis heute nicht aufhören zu programmieren. Im Arbeitsleben entwickelt er Connected-Car-Dienste bei der Valtech Mobility GmbH.
In seiner Freizeit ist er Co-Organisator der Softwerkskammergruppen Düsseldorf und Ruhrgebiet. Wenn er mal nicht programmiert, praktiziert er Aikido oder spielt Trompete.