Nummer 1 - 2018

JAVA MAGAZINE

.nl. jug

Onafhankelijk tijdschrift voor de Java-professional

Maak virtual reality werkelijkheid met Java

Features

Terugblikken J-Fall, Masters of Java en meer

Cassandra 'onder de motorkap'

ScalaTest met Selenium

ScalaTest met Selenium

Selenium is een tool waarmee je browsertests kunt automatiseren. Meer wist ik er niet van. Toevallig wist ik dat ScalaTest ondersteuning heeft voor Selenium en op een dag besloot ik om het toch eens uit te proberen. Het was verrassend gemakkelijk en leuk!

Op ons project voor de Stichting Kerkelijk Geldbeheer, een financiële dienstverlener, hadden we alles: een back-end in Scala en het Play! Framework, een front-end met de nieuwste Angular, continuous deployment met Docker. Weinig management en nergens een Sharepoint te bekennen. Het enige wat we nog niet hadden, was een goede suite end-to-end integratietests en geen van de ontwikkelaars in ons team had ervaring met Selenium. Wel waren we allemaal bang voor de reputatie van dit soort tests: erg onderhoudsgevoelig. Zoals gezegd, wist ik dat ScalaTest (het testframework dat we gebruikten) Selenium ondersteunde. Proberen dus!

Java

Nu is het niet erg behulpzaam een artikel te schrijven over hoe mooi Scala integreert met Selenium, in een tijdschrift waarvan de lezers voornamelijk in Java-projecten werken. Daarom heb ik een demoproject gemaakt in een bestaande Spring Boot voorbeeld-applicatie die ik heb geforkt van de GitHub van Spring zelf. Via link 1 kan je alle Java- en Scalacode zien, inclusief hoe ik de Maven-pom heb uitgebreid om ScalaTest te kunnen draaien binnen een bestaand project. In het artikel zelf gebruik ik, als simpel voorbeeld, de site van Google.

ScalaTest

ScalaTest is het bekendste testframework voor Scala. Het staat bekend om zijn fraaie DSL, waarbij het lijkt alsof je gewoon Engelse zinnen schrijft, terwijl het eigenlijk Scala is. Zie **Listing 1** voor een voorbeeld.

Het grote voordeel hiervan is dat je niet meer gebonden bent aan de regels voor Java identi-fiers om je test een naam te geven. Je kunt gewoon een string gebruiken. In Java moet je namelijk een camel-cased (of eventueel snake-cased) identifier gebruiken. Dat leidt ertoe dat testnamen vaak moeilijk leesbaar zijn of te kort om goed uit te kunnen afleiden wat de test doet. In ScalaTest kan je jouw test veel uitgebreider beschrijven en die beschrijving toch leesbaar houden.

Een andere unieke eigenschap van ScalaTest is dat ScalaTest erg plug-and-play is door het gebruik van traits (een soort interfaces) die je kunt 'in-mixen'. Wil je traditionele tests schrijven of liever in een BDD-stijl werken? Daar zijn traits voor. Gebruik je graag *should* of vind je dat te zwak en gebruik je liever *must*? Kies dan de bijbehorende trait. Code uitvoeren voor of na elke test? Je raadt het al: trait. En inderdaad, ook voor de Selenium-DSL is een trait.

```
class StringTest extends FlatSpec with Matchers {

  behavior of "A string"

  it should "return its length" in {
    "hello world".length shouldBe 11
  }
}
```

Listing 1: Een voorbeeld van ScalaTest

```
go to "http://www.google.com"
click on "q"
textField("q").value = "ScalaTest"
submit()
title shouldBe "ScalaTest - Google Search"
```

Listing 2: Selenium DSL

Jan Ouwens is senior Scala-ontwikkelaar bij Codestar. Hij houdt zich graag bezig met Java, Scala, softwarekwaliteit en zijn open source-tool EqualsVerifier.

De DSL

Hoe ziet die Selenium-DSL er dan uit? De website van ScalaTest (link 2) geeft het volgende voorbeeld in **Listing 2**.

De eerste regel ligt voor de hand. In `click on "q"` gaat ScalaTest op zoek naar een HTML-element waarbij het id-attribuut of het name-attribuut "q" is. In mijn ervaring kan het soms traag zijn als er geen id attribuut is, want er wordt eerst op id gezocht en daarna pas op name. Je kunt ook explicieter zijn:

```
click on name("q")
```

Ook kun je klikken op andere attributen, zoals linkText, cssSelector en xpath. Dat ziet er bijvoorbeeld zo uit:

```
click on linkText("I'm Feeling Lucky")
```

Daarnaast kun je elementen opvragen met de functies find en findAll:

```
val box = find(name("q"))
```

find geeft een Option met het eerstgevonden element; findAll geeft een collectie met alle matchende elementen. Ook hier kan je weer zoeken op name, id, cssSelector, etc.

Eventually

Ik heb het bovenstaande voorbeeld expres een beetje versimpeld. Na het submitten van de pagina, zal Google de zoekresultaten renderen via Javascript. Het duurt dus even voordat de pagina geladen is. In de tussentijd moet je wachten. Hiervoor kan je Seleniums explicit wait gebruiken, maar veel eenvoudiger is het om er nog een trait van ScalaTest bij te pakken: `Eventually` (zie **Listing 3**).

Dit zorgt ervoor dat de code in het blok herhaald wordt uitgevoerd, net zolang tot hij slaagt of totdat een time-out is verstreken. Zowel de timeout als de intervalperiode kan je configureren. Het is belangrijk je te realiseren dat de code in een `eventually`-blok geen side-effects mag hebben. Anders kan het side-effect een onbekend aantal keren achter elkaar worden uitgevoerd en daarmee het verdere verloop van de test beïnvloeden.

Het nut van `eventually` is enorm: negen van de tien keer is een timing-probleem in de test op te lossen door er gewoon `eventually` omheen te zetten! Overigens moet `eventually` wel geconfigureerd worden. Standaard blijft

```
eventually {
  title shouldBe "ScalaTest - Google Search"
}
```

Listing 3: `eventually`

```
override implicit val patienceConfig: PatienceConfig =
    PatienceConfig(timeout = Span(5, Seconds), interval = Span(50, Millis))
```

Listing 4: Timeouts

```
def never[T](fn: => T): Assertion =
  try {
    eventually(fn)
    Assertions.fail("Condition is completed successfully, which is not allowed")
  }
  catch {
    case _: TestFailedDueToTimeoutException => Assertions.succeed
  }
```

Listing 5: `never`

hij namelijk maar maximaal 150ms proberen, wat voor Selenium veel te kort is. Een configuratie die voor ons project goed werkte, is die in **Listing 4**.

Never

Soms wil je juist het tegenovergestelde van `eventually`. Bijvoorbeeld: na het veranderen van een veld mag de oude waarde niet meer in een overzicht te zien zijn. Of: na het klikken op een knop mag geen foutmelding verschijnen. ScalaTest heeft hier geen trait voor, maar een never-functie die dit doet, is eenvoudig te maken: zie **Listing 5**.

Nu kan je het volgende aan de test toevoegen:

```
never {
  title shouldBe "JUnit - Google Search"
}
```

De never functie gebruikt dezelfde time-out en interval als `eventually`. Hij zal de code in het blok volgens het interval opnieuw blijven uitvoeren en zal slagen als de time-out wordt bereikt of falen als dat niet het geval is. Je zou kunnen zeggen dat never niet de juiste naam is, aangezien hij stopt na de time-out. Maar bij het lezen van de code is de intentie wel meteen lekker duidelijk!

Maven

Het inrichten van al dit moois in een bestaand Java-project, is eenvoudig. Ik raad aan dit in een nieuwe Maven-module te doen. Scala en ScalaTest hebben elk een dependency en een Maven-plugin nodig. De Maven-plugin voor ScalaTest zorgt ervoor dat alle in Scala geschreven tests automatisch meedraaien als je `mvn test` runt.

NEGEN VAN DE TIEN KEER IS EEN TIMING-PROBLEEM OP TE LOSSEN MET EVENTUALLY

Voor Selenium zijn twee dependencies nodig. Ten eerste natuurlijk die van Selenium zelf. Ten tweede kan je webdrivermanager gebruiken, die automatisch de meest recente driver voor je browser(s) downloadt en laadt. Daar hoef je dus niet meer naar om te kijken.

Let wel op: omdat je altijd de meest recente driver hebt, moet je zorgen dat je build-machine ook altijd de meest recente versie van je browser heeft. Als dit niet mogelijk is, kun je webdrivermanager ook configureren om de browserversie te fixeren. De details kun je vinden in de POM in het demo-project; zie link 1.

Plumbing

Ook in de code is een klein beetje configuratie nodig. Aangezien deze code precies één keer moet worden uitgevoerd, is het handig om ergens een singleton object te maken waarin de WebDriver wordt geconfigureerd: zie **listing 6**.

```
object Plumbing {
  ChromeDriverManager.getInstance.setup()
  implicit val driver: WebDriver = new ChromeDriver

  driver.manage.window.setSize(new SeleniumDimension(1290, 1024))
  implicitlyWait(Span(1, Millis))
  sys.addShutdownHook { quit() }
}
```

Listing 6: Configuratie

De WebDriver is *implicit*, omdat deze door vrijwel alle calls in de Selenium-DSL gebruikt wordt. Door hem implicit te laten, hoef je hem nooit expliciet als parameter mee te geven. Dit is wat de DSL leesbaar maakt:

```
click on "q"            // implicit :)
click.on("q")(driver)   // explicit :(
```

Ik heb gemerkt dat het handig is om de grootte van het browservenster te hard-coden, omdat het anders soms niet werkt op machines met andere beeldscherm-resoluties. Dit voorkomt ongemakkelijke momenten als je een laptop op een beamer aansluit om aan je collega's te demonstreren hoe cool ScalaTests Selenium-DSL is.

We zetten implicitlyWait op 3 seconden en vertrouwen op eventually om eventuele verdere timing issues op te lossen.

Tot slot willen we dat de browser weer wordt afgesloten wanneer alle tests voltooid zijn. Daarvoor voegen we een shutdown hook toe aan de JVM, zodat we zeker weten dat alle tests helemaal klaar zijn, voordat we de browser sluiten.

Een eigen trait

Er is altijd een vast setje traits dat je door de hele testsuite gebruikt. Ten eerste de trait die vastlegt in welke stijl je wilt werken: traditioneel, BDD of een andere? Ten tweede de trait die bepaalt of je should of must gebruikt. Verder wil je natuurlijk altijd Eventually gebruiken. En laten we vooral ook WebBrowser niet vergeten, die de Selenium-DSL activeert.

Al deze traits, en meer, kun je verzamelen in je eigen trait, bijvoorbeeld met de naam EndToEndTest. Ook kun je hier extra configuratie en helpers in opnemen, zoals de implicit WebDriver, de never functie die we hierboven besproken hebben en de time-outconfiguratie voor Eventually. Zie **Listing 7** voor een voorbeeld.

Op deze manier hoef je voor elke testsuite nog maar één enkele trait te extenden:

```
class GoogleTest extends EndToEndTest {
  ...
}
```

Screenshots

Wanneer een Selenium-test faalt tijdens de nightly build, is het handig om te kunnen zien wat is misgegaan. Gelukkig maakt ScalaTest het ook eenvoudig om screenshots te maken wanneer een test is gefaald. In **Listing 8** staat code die je kunt toevoegen aan je EndToEndTest trait. Als je dat doet,

```
trait EndToEndTest extends FlatSpec with Matchers with WebBroser with Eventually {

  implicit val driver: WebDriver = Plumbing.driver

  override implicit val patienceConfig: PatienceConfig = ...

  def never[T](fn: => T): Assertion = ...
}
```

Listing 7: Eigen trait

VOOR IEMAND DIE NIET BEKEND IS MET SELENIUM, MAAKT DE DSL VAN SCALATEST HET HEEL EENVOUDIG OM SNEL UP-AND-RUNNING TE ZIJN

```
override def withFixture(test: NoArgTest): Outcome = {
  val outcome = test()
  if (outcome.isExceptional) {
    setCaptureDir("target")
    val timestamp = LocalDateTime.now.format(
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH.mm.ss"))
    capture to s"$timestamp - ${test.name}.png"
  }
  outcome
}
```

Listing 8: Screenshots

zal bij elke gefaalde test een screenshot worden gemaakt. Het bestand wordt in de target map van het project geplaatst en bevat in de naam het precieze tijdstip en de naam van de test die faalde.

Page objects

ScalaTest heeft ook ondersteuning voor Page objects. Door een klasse te schrijven die de trait Page implementeert, kun je eenvoudig naar een pagina navigeren in je test:

```
val google: GooglePage = ...
go to google
```

Het page object zelf moet in een `val url` definiëren wat de url is van de pagina.

Als je dit design pattern gebruikt, is het handig om streng onderscheid te maken tussen Selenium-code en test-code. Tot nu toe hebben we, om dingen eenvoudig te houden, alles op één hoop gegooid in de `EndToEndTest` trait. We creëren nu een nieuwe trait, `AbstractPage`, die naast Page ook `WebBrowser` implementeert, en daarmee de Selenium-DSL erft.

Bij `EndToEndTest` halen we `WebBrowser` weer weg, want in de testklassen willen we juist geen expliciete Selenium-code meer zien; dat moet nu immers via de Page object gaan.

Dit heeft ook gevolgen voor de code in `EndToEndTest` en `AbstractPage`, die op sommige punten naar elkaar gaan verwijzen. Bijvoorbeeld, de code voor het nemen van de screenshot moet nu naar `AbstractPage` verplaatst worden, maar de `withFixture`-method moet nog steeds in `EndToEndTest` worden geïmplementeerd. `Eventually` is in beide traits nuttig, dus de configuratie daarvan moet nu gedeeld worden. Hoe je dit alles het beste kunt inrichten, kan je zien in het demo-project op link 1.

Conclusie

Voor iemand die niet bekend is met Selenium, maakt de DSL van ScalaTest het heel eenvoudig om snel up-and-running te zijn. Ik was voorheen altijd een beetje huiverig voor Selenium, maar met deze DSL vond ik het zelfs leuk om GUI-tests te schrijven.

In dit artikel besteedde ik veel tijd aan het beschrijven van de set-up, maar in de praktijk is die set-up langzaam gegroeid tijdens het bouwen van de testsuite. Je kan dus prima met een eenvoudige versie hiervan starten. Of je kan de set-up kopiëren uit het demo-project (link 1) en nog sneller up-and-running zijn dan ik was. Wil je meer weten over de Selenium-DSL? Kijk dan op link 2.

Veel testplezier! ■

REFERENTIES

[1] **https://github.com/jqno/selenium-scala-java-demo**
[2] **http://www.scalatest.org/user_guide/using_selenium**