Eine Groovy-DSL zum Erzeugen von Testdaten über JPA

Daniel Behrwind, TRIOLOGY GmbH

Beim automatisierten, integrativen Testen von Software, die mit einem komplexen JPA-Datenmodell arbeitet, steht man unweigerlich früher oder später vor der Frage, wie sich semantisch sinnvolle Testdaten ohne großen Aufwand erzeugen lassen. Dieser Artikel zeigt, wie man mit Groovy eine Domain Specific Language (DSL) definieren kann, die es erlaubt, Testdaten leicht lesbar, modular und getrennt vom eigentlichen Test-Code festzulegen.

Automatisierte Software-Tests gehen von bestimmten Annahmen aus, bevor sie sicherstellen, dass das getestete Feature unter den gegebenen Bedingungen korrekt funktioniert. Testet man auf integrativer Ebene, manifestieren sich diese Annahmen oft in einem bestimmten Datenstand einer relationalen Datenbank. Vor Durchführung eines Tests sind also entsprechende Testdaten zu erzeugen.

Häufig widersprechen sich die für die einzelnen Testfälle benötigten Daten aber gegenseitig, sodass es nicht möglich ist, auf einem einmalig definierten Stand aufzusetzen. Vielmehr müssen die Daten programmatisch pro Testfall definiert und in die unterliegende Datenbank eingefügt werden. Dabei wird der Testcode schnell unübersichtlich, wenn große Hierarchien oder große Datenmengen erforderlich sind.

Es liegt also nahe, die Erzeugung der Testdaten auszulagern, sodass sich der eigentliche Test auf das Wesentliche konzentrieren kann. Dazu drängen sich zunächst zwei Alternativen auf: native SQL-Skripte oder die Test-Bibliothek DBUnit. Beide orientieren sich allerdings an Tabellen und Spalten statt an Java-Objekten. Eine im Produktiv-Code durch JPA gewonnene Abstraktion der relationalen Datenbank würde damit im Test wieder verloren gehen. Insbesondere beim Abbilden von Referenzen durch Fremdschlüssel macht sich das schmerzlich bemerkbar. Die Generierung der Daten lässt sich zwar vom Testcode trennen, in puncto Übersichtlichkeit ist aber nichts gewonnen. Der Fokus sonstiger Alternativen liegt eher darin, große Mengen an Zufallsdaten zu erzeugen. Sie zielen auf Performance-Tests ab, sind jedoch für einen fachlichen Test ungeeignet.

Anforderungen

So sähe eine Lösung aus, die dem Datenbank-scheuen Java-Entwickler zusagt:

- Die Lösung müsste nahtlos aus dem Java-Test-Code heraus aufrufbar sein
- Testdaten sollten wiederverwendbar und modular definierbar sein
- Die Lösung sollte vollständig objektorientiert arbeiten
- Gespeicherte Entities sollten dem aufrufenden Code zugänglich gemacht werden
- Testdaten sollten in einer gut lesbaren Form komfortabel definiert werden können

Das Grails-Fixtures-Plug-in *(siehe „http://www.grails.org/plugin/fixtures")* erfüllt überwiegend die genannten Anforderungen bereits mit einer Domain Specific Language (DSL). Es ist allerdings eng mit dem Grails-Framework verwoben. Im Folgenden wird gezeigt, wie sich die Anforderungen mit wenig Aufwand in einer Groovy-DSL umsetzen lassen, die direkt in herkömmlichen Java-Projekten nutzbar ist. Der vollständige Code steht unter *„https://github.com/triologygmbh/test-data-loader"* zur Verfügung.

Groovy to the Rescue

Die JVM-Sprache Groovy bietet mit ihrer dynamischen Natur und vielen syntaktischen Möglichkeiten beste Voraussetzungen, auf einfache Weise eine eigene DSL zu definieren. Unter *„http://www.*

```
create User, 'Peter', {
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 1

```
class EntityBuilder {
  void buildEntities(String entityDefinitionFile) {
    DelegatingScript script = createExecutableScriptFr
omEntityDefinition(entityDefinitionFile)
    script.setDelegate(this)
    script.run()
  }
  // …
}
```

Listing 2

groovy-lang.org/documentation.html" stehen weiterführende Informationen zu Groovy bereit. Sprach-Features, die in der vorgestellten Lösung Verwendung finden, sind im Fall des Einsatzes kurz erklärt.

Die größte Herausforderung beim Entwickeln der angedachten DSL ist es, eine möglichst einfache Syntax für die Definition der Testdaten zu finden und diese Definition dann in tatsächliche JPA-Entities umzuwandeln. Um dies ein wenig plastischer zu machen, zunächst ein Beispiel: Ziel ist es, einen JPA-Entity-User durch folgendes Snippet zu instanziieren, die Felder entsprechend zu initialisieren, das Entity in der Datenbank zu speichern und den Test-Code unter dem Namen „Peter" zur Verfügung zu stellen *(siehe Listing 1)*.

Groovy-Skripte einlesen und ausführen

Bevor wir uns anschauen, wie aus der Definition ein initialisiertes Entity wird, werden zunächst Lösungen für die übrigen Anforderungen diskutiert. Groovy wird in Java-Byte-Code übersetzt. Daher lässt sich Groovy-Code direkt aus Java-Code heraus aufrufen, als wäre er in Java geschrieben. Die nahtlose Java-Integration bekommen wir also schon einmal geschenkt.

Zusätzlich ist es möglich, Groovy als Skript-Sprache einzusetzen. Dabei kann man relativ einfach Skript-Dateien mit Groovy-Bordmitteln programmatisch einlesen und ausführen. Diese Möglichkeit eignet sich ideal für unsere Zwecke: Wir können so unsere Testdaten-Definitionen in beliebige „.groovy"-Dateien auslagern und je nach Testfall die benötigten Dateien laden. Dazu wird die Klasse „EntityBuilder" eingeführt, die einen Dateinamen entgegennimmt und die in der Datei definierten Entities erzeugt *(siehe Listing 2)*. Der vollständige Code steht im eingangs erwähnten GitHub-Repository zur Verfügung.

„buildEntities" nimmt den Dateinamen eines Entity-Definitions-Skripts entgegen. Diese Definition wird eingelesen und in eine ausführbare Skript-Instanz umgewandelt. Dabei wird zur Laufzeit eine Klasse erstellt, deren „run"-Methode den Inhalt des Skripts enthält. Diese „run"-Methode können wir nun wie jede andere Methode aufrufen und so das Skript ausführen. Hierbei ist zu beachten, dass Groovy Referenzen, die im Skript nicht eindeutig sind, zur Laufzeit auflösen kann. Die eingesetzte Skript-Subklasse „DelegatingScript" erlaubt es, ein Delegate zu setzen, gegen das nicht eindeutige Referenzen aufgelöst werden. Der EntityBuilder setzt sich an dieser Stelle selbst als Delegate des Skripts. Wozu das notwendig ist, wird später deutlich.

Mit dieser Umsetzung können wir nun Testdaten modular in beliebigen Skriptdateien definieren und diese nach Bedarf laden. Gehen wir zunächst davon aus, dass tatsächlich die in den Skripten definierten Entities instanziiert und initialisiert werden, so stellt sich die Frage, wie ihre Daten in die Datenbank gelangen.

Der Glue-Code

Um die Persistierung der Entities von ihrer Erzeugung zu entkoppeln, bietet der EntityBuilder die Möglichkeit, „EntityCreatedListener" zu registrieren. So können wir im Glue-Code in der Klasse „TestDataLoader" *(siehe „https://github.com/triologygmbh/test-data-loader")* einen Listener einsetzen, der sich darum kümmert, die Daten zu speichern. „TestDataLoader" erwartet als Konstruktor-Parameter einen fertig initialisierten JPA-EntityManager. Seine Methode „loadTestData" kann dann vom Client mit Entity-Definitions-Dateien aufgerufen werden. Nach Initialisierung des Listeners reicht er die Definitionen an den EntityBuilder weiter. Dieser erzeugt die definierten Entities und übergibt sie über das Listener-Interface an den EntityPersister zum Speichern.

Definition der eigentlichen DSL

Wir sind jetzt in der Lage, Entities in beliebigen Skripten zu definieren, die Definitionen einzulesen und die erzeugten Entities zu speichern. Beim eigentlichen Erzeugen der Entities kommen gleich mehrere Groovy-Features zum Tragen: Groovy erlaubt es, beim Aufruf von Methoden die Parameter-umschließenden Klammern wegzulassen. Zusätzlich ist es möglich, mit geschweiften Klammern Closures zu definieren, also ausführbare Code-Abschnitte, ähnlich Java-8-Lambdas, die wie gewöhnliche Objekte über Variablen referenziert und übergeben werden können. Closures lassen sich dann an beliebiger Stelle ausführen.

Damit wird klar, dass der Ausdruck „create User, 'Peter', { }" nichts weiter ist als ein Aufruf der statischen Methode „create" im EntityBuilder („static import") mit drei Parametern (in Groovy kann eine Klasse einfach über ihre Namen referenziert werden; der Ausdruck „User" ist daher äquivalent zu „User.class."). *Listing 3* zeigt, was beim Aufruf von „create" aus der DSL heraus passiert.

Wir stellen fest, dass die statische „create"-Methode lediglich an „createEntity" der Singleton-Instanz des EntityBuilder delegiert. Hier wird zunächst eine neue Instanz der übergebenen Entity-Klasse erzeugt und unter dem ebenfalls übergebenen Namen in der „java.util.Map" unter „entitiesByName" registriert. Hinweis: „entitiesByName[entityName] = entity" ist analog zu „entitiesByName.put(entityName, entity)". Nachdem das neue Entity in der Methode „executeEntityDataDefinition" mit Daten initialisiert wird, werden schlussendlich die registrierten Listener informiert.

Die eigentliche Magie passiert in den zwei Zeilen der Methode „executeEntityDataDefinition". Ihr werden das neu instanziierte Entity sowie das aus dem Skript stammende Closure übergeben, in dem die Daten für das Entity definiert sind. Um zu verstehen, was passiert, müssen wir allerdings etwas weiter ausholen. Schauen wir zunächst noch einmal auf das Closure, das im Skript als letzter Pa-

rameter an die „create"-Methode übergeben wird *(siehe Listing 4)*. Es werden augenscheinlich Variablen Werte zugewiesen. Diese Variablen sind allerdings nicht deklariert, sodass hier ein weiteres Groovy-Feature zum Tragen kommt. Der Ausdruck „myObject.someProperty = 'value'" entspricht dem Aufruf eines Setters „myObject.setSomeProperty('value')". Es werden im Closure also zwei Setter aufgerufen, die Frage ist nur: Worauf? Da die gesetzten Felder zufällig genau den Properties des zu erzeugenden User-Entitys entsprechen, wäre es doch praktisch, sie würden direkt auf dem Entity aufgerufen? Die beiden Zeilen in „executeEntityDataDefinition" bewirken genau das.

Ähnlich wie bei Skripten ist Groovy in der Lage, Methoden-Aufrufe innerhalb eines Closure zur Laufzeit dynamisch aufzulösen. Dazu kann man auch für das Closure ein Delegate definieren. Die in „executeEntityDataDefinition" aufgerufene „rehydrate"-Methode erstellt eine Kopie des Closure und setzt den ersten Parameter als Delegate, in unseren Fall also das zuvor instanziierte Entity. Wird das Closure nun mit „entityDataDefinition.call()" ausgeführt, werden die Setter im Beispiel tatsächlich auf dem User-Entity aufgerufen, sodass dieses mit den entsprechenden Daten initialisiert wird.

Es ist an der Zeit, das bisher Erreichte einmal auszuprobieren, bevor wir die DSL noch weiter verfeinern. Definieren wir uns also zunächst einen User *(siehe Listing 5)*.

Die Klasse „Demo.java" führt vor, wie der TestDataLoader aus Java-Code heraus benutzbar ist und dass die erzeugten Entities tatsächlich persistiert werden. Die DSL-Snippets stammen aus der Datei „testData.groovy" *(siehe „https://github.com/triologygmbh/test-data-loader")*.

Verschachtelte Entities

So weit, so gut – wir haben den Roundtrip von der DSL über die Datenbank bis hin zum Testcode geschafft. Bleibt noch offen, wie ein komplexes Datenmodell zu bedienen ist, wie wir also mit Referenzen zwischen den Entities umgehen. Nehmen wir hierzu an, dass ein Benutzer einer Abteilung zugeordnet werden kann. Der User erhält eine „@ManyToOne"-Beziehung zum Department. In der DSL können wir die Erzeugung von Entities ganz einfach schachteln *(siehe Listing 6)*.

Was aber, wenn ein zweiter User derselben Abteilung angehört? Das Department mit der „create"-Methode ein zweites Mal anzulegen, ist offensichtlich keine Option. Wir müssen also eine Möglichkeit schaffen, bereits angelegte Entities aus der DSL heraus zu referenzieren. Da wir uns in ganz normalem Groovy-Code bewegen, wäre es möglich, sich ein von der „create"-Methode erzeugtes Entity in einer Variablen zu merken. Mit ein wenig Unterstützung vom EntityBuilder geht das aber auch einfacher *(siehe Listing 7)*.

Was geht hier nun vor sich? Wie kann die Zuweisung „department = lostBoys" funktionieren, ohne dass „lostBoys" initialisiert wurde? Hier greifen wieder verschiedene Groovy-Eigenschaften ineinander: „lostBoys" ist zunächst ein Bezeichner, der nicht aufgelöst werden kann. Groovy ruft in diesem Fall einen Getter für ein Property mit dem Namen des Bezeichners auf – ähnlich wie den Setter bei der Zuweisung von Werten zu Variablen. Analog entspricht der Ausdruck „myObject.someProperty" dem Aufruf „myObject.getSomeProperty()". Auch hier stellt sich wieder die Frage, worauf der Getter aufgerufen wird. Das

```
class EntityBuilder {
  static <T> T create(Class<T> entityClass, String
entityName, Closure entityData) {
    return instance().createEntity(entityClass, enti-
tyName, entityData);
  }

  private <T> T createEntity(Class<T> entityClass,
String entityName, Closure entityData) {
    T entity = createEntityInstance(entityName, enti-
tyClass)
    executeEntityDataDefinition(entityData, entity)
    notifyEntityCreatedListeners(entity)
    return entity
  }

  private <T> T createEntityInstance(String entity-
Name, Class<T> entityClass) {
    ensureNameHasNotYetBeenAssigned(entityName, enti-
tyClass)
    T entity = entityClass.newInstance()
    entitiesByName[entityName] = entity;
    return entity
  }

  private void executeEntityDataDefinition(Closure
entityDataDefinition, Object entity) {
    entityDataDefinition = entityDataDefinition.
rehydrate(entity, this, this)
    entityDataDefinition.call()
  }

  // …
}
```

Listing 3

```
{
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 4

```
create User, 'Peter', {
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 5

```
create User, 'Peter', {
  firstName = 'Peter'
  lastName = 'Pan'
  department = create Department, 'lostBoys', {
    name = 'The Lost Boys'
  }
}
```

Listing 6

```
create User, 'Tinker', {
    firstName = 'Tinker'
    lastName = 'Bell'
    department = lostBoys
}
```

Listing 7

Delegate des Closure kann es nicht sein. Das ist in diesem Fall eine User-Instanz, die sicherlich keine Methode „getLostBoys()" bietet.

An dieser Stelle kommt das beschriebene Delegate des Skripts ins Spiel. Erinnern wir uns, der EntityBuilder setzt sich vor dem Ausführen des Skripts selbst als Delegate. Zwar hat auch der EntityBuilder keine Methode „getLostBoys()", er implementiert jedoch die Methode „propertyMissing", die von Groovy beim Zugriff auf ein nicht existierendes Property mit dessen Namen aufgerufen wird. Auf diese Weise können wir nun das zuvor angelegte Department unter dem Namen „lostBoys" auffinden und zurückgeben, sodass es im Skript als Wert gesetzt wird *(siehe Listing 8)*. Das Ganze lässt sich sogar so weit treiben, dass wir „Peter" als Department-Head setzen, während wir das Department anlegen und ihm zuordnen *(siehe Listing 9)*.

Code Completion

Damit haben wir alles, was wir brauchen. Es geht aber noch ein bisschen komfortabler. Bislang sind wir bei der Definition der eigentlichen Entity-Daten ziemlich auf uns gestellt. Innerhalb der DSL gibt es keine Möglichkeit herauszufinden, welche Properties ein Entity hat und von welchem Typ sie sind. Code-Completion durch die IDE ist somit auch nicht möglich. Wir können der IDE allerdings mitteilen, was das Delegate des Closure sein wird. Alle Informationen sind im Aufruf der statischen „create"-Methode des EntityBuilder vorhanden. Das Delegate des Closure ist immer eine Instanz der gleichzeitig übergebenen Klasse. Ergänzen wir daher die „create"-Methode um zwei Annotationen, um diese Information bekannt zu machen *(siehe Listing 10)*. Im Resultat weiß die IDE (hier IntelliJ IDEA), dass in unserem Beispiel Aufrufe innerhalb des Closure an eine User-Instanz delegiert werden, und bietet entsprechend die Properties des Users an.

Geschafft! Unsere DSL erlaubt es, Testdaten leicht lesbar, modular und getrennt vom eigentlichen Testcode zu definieren. Hierarchien können verschachtelt und objektorientiert statt über Fremdschlüssel abgebildet werden. Dabei müssen wir die Java-Welt nicht verlassen und keinen gedanklichen Bruch hin zum relationalen Modell der Datenbank hinnehmen. Da wir uns innerhalb der DSL in Groovy-Code bewegen, genießen wir alle Freiheiten, die eine Programmiersprache so mit sich bringt. Ein denkbares Szenario wäre zum Beispiel, große Datenmengen über Schleifen zu generieren.

```
private def propertyMissing(String name) {
  if (entitiesByName[name]) {
    return entitiesByName[name]
  }
  // handle missing reference
}
```

Listing 8

```
create User, 'Peter', {
  department = create Department, 'lostBoys', {
      name = 'The Lost Boys'
      head = Peter
  }
}
```

Listing 9

```
static <T> T create(@DelegatesTo.Target Class<T> entityClass, String entityName, @DelegatesTo(strategy = Closure.DELEGATE_FIRST, genericTypeIndex = 0) Closure entityData) {
  return instance().createEntity(entityClass, entityName, entityData);
}
```

Listing 10

```
create User named 'Peter' with {
  firstName = 'Peter'
  lastName = 'Pan'
}
```

Listing 11

In diesem Artikel unbehandelt bleibt allerdings die Frage, wie die Datenbank nach einem Testfall bereinigt werden soll. In den vorgeführten Beispielen („Demo.java") machen wir es uns einfach und rollen nach jedem Test die Transaktion zurück. Für einen wirklich integrativen Test ist das natürlich keine Option. In echten Projekten haben wir bislang mittels Datenbank-Skript und „TRUNCATE TABLE" alle Tabellen geleert. Bei großen Schemata hat sich das allerdings erheblich auf die Ausführungszeit der Tests ausgewirkt. Es wäre interessant auszuprobieren, ob es effizienter wäre, die Datenbank programmatisch zu bereinigen. Über einen Stack sollte es relativ einfach möglich sein, die angelegten Entities in umgekehrter Reihenfolge wieder zu löschen.

Noch ein kleiner Wermutstropfen: Die DSL könnte sich mit einem Fluent-API noch schöner gestalten *(siehe Listing 11)*. Da sich die Definition hier jedoch über drei Methoden-Aufrufe verteilt („create", „named" und „with"), habe ich keine Möglichkeit gefunden, das Delegate des Closure über Annotationen bekannt zu machen. Grund ist, dass die Klasse, an die delegiert wird, an eine andere Methode übergeben wird als das Closure selbst. Zugunsten der Code-Completion hat sich der Autor daher für eine etwas unschönere DSL-Syntax entschieden. Vielleicht fällt ja jemandem ein, wie man beides haben kann? Pull-Requests mit Verbesserungen sind in jedem Fall willkommen.

Daniel Behrwind
d.behrwind@gmail.com

Daniel Behrwind ist als Software-Entwickler bei der TRILOGY GmbH tätig. Dort befasst er sich überwiegend mit der Entwicklung von Individual-Software, wobei er schwerpunktmäßig an Java-Webprojekten arbeitet. Als leidenschaftlicher Clean-Code-Verfechter ist er begeistert von kreativen Lösungen für komplexe Probleme, die so offensichtlich aussehen, als seien sie von selbst entstanden.