Unified Query

QueryDSL als Alternative zum JPA Criteria API

Christoph Schmidt-Casdorff

QueryDSL bietet ein einheitliches Konzept, um Abfragen in SQL-nahen Sprachen zu formulieren. Dieser Artikel widmet sich der Integration von QueryDSL mit JPA.

Java Persistence API – Brücke zwischen Java und Datenbank

JPA [JPARef] ist im Rahmen der JEE-Spezifikationen der Standard für ORM (*Object Relational Mapping* – siehe [Hibernate]). ORM stellt eine Brücke zwischen Daten aus relationalen Datenbanken und objektorientierten Programmiersprachen (in unserem Fall also Java) dar. Daten der relationalen Datenbanken werden auf vielfältige Weise in Objekte überführt, sogenannte *Entities*.

JPA sieht zwei Abfragesprachen vor, mit deren Hilfe Abfragen gegen Entities formuliert werden können, nämlich JPQL und JPA Criteria API.

JPQL ist eine statische Abfragesprache, deren Syntax stark der von SQL gleicht. In Java werden Abfragen in JPQL als Zeichenkette aufgebaut und anschließend durch JPA-Provider (wie beispielsweise Hibernate oder EclipseLink) ausgeführt.

Der Aufbau eines JPQL-Statements genügt im Wesentlichen denselben Prinzipien wie ein SQL-Statement. Neben den naturgegebenen Unterschieden durch die abweichende Syntax und einigen abweichenden Konzepten zwischen SQL und JPQL bleiben die Prinzipien zur Formulierung einer Abfrage dieselben.

Da JPQL genau wie SQL von Haus aus nicht geeignet ist, typsicher dynamische Abfragen aufzubauen, spezifiziert JPA eine weitere Abfragesprache, nämlich *JPA Criteria API*. Criteria API liefert eine typsichere Java-Programmierschnittstelle, um dynamisch Abfragen aufzubauen. Kritik gibt es an Criteria API allerdings wegen schlechter Lesbarkeit, fehlender Kompaktheit und der Fülle an Statements, die es benötigt, um komplexere Abfragen zu erstellen.

QueryDSL bietet ein typsicheres API an, um Abfragen in JPQL programmatisch zu erstellen. QueryDSL ist sowohl für JPA Criteria API als auch JPQL eine Alternative, da es die Flexibilität von Criteria API mit der Sprachmächtigkeit und Kompaktheit von JPQL kombiniert.

Rückblick

QueryDSL bietet ein einheitliches Konzept, um Abfragen für SQL und SQL-nahe Abfragesprachen in Java zu formulieren. Eine ausführliche Beschreibung der Integration von QueryDSL mit SQL findet sich in [Schm15]. Es wurde auf grundlegende Konzepte von QueryDSL und auf die Programmierschnittstelle zum Aufbau von Abfragen eingegangen. Die dort beschriebenen Konzepte lassen sich im Wesentlichen auf die Integration von QueryDSL mit JPA übertragen.

Grundlegende Konzepte in QueryDSL

Die große Ähnlichkeit in der Struktur der Abfragen zwischen JPQL und SQL nutzt QueryDSL, um ein gemeinsames Konzept zum Aufbau von Abfragen in einer der beiden Sprachen anzubieten.

Die größten Unterschiede stellen allerdings die Objekte dar, auf die sich die Abfragen beziehen. Während sich in SQL die Abfragen auf Tabellen und Spalten beziehen, stehen bei JPQL die Entity-Klassen, -Attribute und deren Beziehung untereinander im Fokus. Ausdrücke, welche sich auf diese Objekte beziehen, heißen sowohl in JPA als auch in QueryDSL *Path Expression*.

Die Aufgabe von QueryDSL ist es, die Abfragen textuell zusammenzusetzen und gegebenenfalls Parameter zuzuweisen. Die syntaktische Prüfung und letztlich die Ausführung wird den JPA-Providern überlassen. Diese unterscheiden sich durchaus in Nuancen. Daher ist jede mit QueryDSL erzeugte Abfrage immer im Kontext des zugrunde liegenden JPA-Providers zu sehen.

Query Types und Domänen

Die Domäne der Anbindung an JPQL sind Entity-Klassen und -Attribute sowie deren Beziehung untereinander. QueryDSL benötigt eine eigene Darstellung (sogenannter *Query Type*), um diese Domänenobjekte zugänglich zu machen. Dazu stellt Query-DSL einen Generator bereit, welcher zu jeder *persistence entity* (im Wesentlichen Entity-Klassen) einen entsprechenden Query Type erstellt. Die Standardkonvention für die Namen stellt vor jeden Namen einer Entity-Klasse ein Q. So wird aus der Entity-Klasse `TestPerson` die generierte Query Type-Klasse `QTestPerson`. Dieser Generator kann mittels des Buildprozesses (maven, ant) oder explizit aufgerufen werden.

Die Query Types sind der Schlüssel zur Typsicherheit von QueryDSL. Sie beschreiben die Typen der einzelnen Attribute und definieren sowohl die Entity-Klassen als auch die Zielklassen der Relationen. All diese Informationen sind natürlich bereits in den Entity-Klassen zu finden und werden aus diesen durch Query-DSL-Generatoren extrahiert.

```
select testPerson
  from TestPerson testPerson
  where testPerson.name = ?1
```

Listing 1: Beispiel einer einfachen Abfrage in JPQL

```
JPAQuery<TestPerson> query= new JPAQuery<TestPerson>(entityManager);
query.select(QTestPerson.testPerson)
  .from(QTestPerson.testPerson)
  .where(QTestPerson.testPerson.name.eq("Solo"));
TestPerson person = query.fetchOne();
```

Listing 2: Formulierung der Abfrage aus Listing 1 in QueryDSL

In Listing 1 ist eine einfache JPQL-Abfrage und in Listing 2 deren Umsetzung mit QueryDSL beschrieben. Listing 3 zeigt einen Ausschnitt eines generierten Query Types. Dort finden sich dessen Attribute (siehe `name` oder `sidTestPerson`). Eine Relation zwischen den Entity-Klassen `TestPerson` und `TestAdresse` wird im Generat durch `testAdresses` repräsentiert.

```
public class QTestPerson extends EntityPathBase<TestPerson> {
  public static final QTestPerson testPerson =
    new QTestPerson("testPerson");
  public final NumberPath<Long> sidTestPerson =
    createNumber("sidTestPerson", Long.class);
  public final StringPath name = createString("name");
  protected QTestAdresse testAdresse;
  public final SetPath<TestAdresse, QTestAdresse> testAdresses =
    this.<TestAdresse, QTestAdresse>createSet(
      "testAdresses",TestAdresse.class,QTestAdresse.class,...);
...
```

Listing 3: Beispiel eines generierten Query Types für die Entity-Klasse TestPerson

Erläuterungen und Beispiele, wie SQL-Abfragen mit QueryDSL aufgebaut werden, finden Sie in [Schm15]. Die dort getroffenen Aussagen lassen sich auf JPQL übertragen.

Abfragen in JPQL

In Listing 4 repräsentieren die beiden Query Types `QTestPerson` und `QTestAdresse` ihre entsprechenden Entity-Klassen. Der INNER JOIN beschreibt explizit die Navigation zwischen einer Person und den ihr zugeordneten Adressen und wird durch `testPerson.testAdresses` beschrieben. Genauso intuitiv werden Bedingungen an Attribute formuliert (hier `testAdresse.strasse.startsWith("N")`).

JPQL-Abfragen werden als Instanzen der Klasse `JPAQuery` repräsentiert (s. Listing 4). Diese ermittelt den zugrunde liegenden JPQL-Dialekt aus dem übergebenen Entitymanager. JPAQuery stellt ein *fluent API* zum Aufbau einer JPQL-Abfrage bereit.

```
JPAQuery<TestPerson> query= new JPAQuery<TestPerson>(entityManager);
QTestPerson testPerson = QTestPerson.testPerson;
QTestAdresse testAdresse = QtestAdresse.testAdresse;
query.select(testPerson)
  .from(testPerson)
    innerJoin(testPerson.testAdresses, testAdresse)
  .where(testAdresse.strasse.startsWith("N"));
List<TestPerson> result = query.fetch();
```

Listing 4: Aufbau eines JPQL-Ausdrucks mit QueryDSL

Durch `select` wird die Ergebnismenge der Abfrage beschrieben. Im Allgemeinen besteht diese aus einzelnen oder einer Menge an Entity-Objekten. Darüber hinaus unterstützt JPQL weitere Konzepte zur Beschreibung der Ergebnismenge wie Auswahl bestimmter Attribute der Ergebnisobjekte (Projektionen), Überführung von Ergebnissen in Objekte anderer Klassen (*constructor expressions*) oder Aggregation von Ergebnissen (*aggregate expressions*). Auch diese Konzepte werden durch QueryDSL unterstützt.

Um unterschiedliche Navigationspfade über die gleichen Entity-Klassen zu unterscheiden, können unterschiedliche Instanzen einer Entity-Klasse in einer JPQL-Abfrage eingesetzt werden. Entsprechend werden in einer QueryDSL-Abfrage mehrere Instanzen eines Query Types eingesetzt. In Listing 5 werden zwei unterschiedliche Instanzen von `QTestPerson` eingesetzt. Ebenfalls kann man an Listing 5 den Einsatz von impliziten JOINS erkennen, welche durch den Path-Ausdruck der zugehörigen Relation angegeben werden (z. B. `person.testAdresses`).

Soweit eine kurze Vorstellung, wie mit QueryDSL Abfragen in JPQL formuliert werden können. Es finden sich alle grundlegenden Konzepte von JPQL im QueryDSL-API wieder, sodass eigene Abfragen schnell zu erstellen sind. Zusätzliche Beispiele finden Sie auf [QRef] und dem Github-Projekt [CSC15], welches diesen Artikel begleitet.

```
QTestPerson testPerson = QTestPerson.testPerson;
QTestAdresse testAdresse = QTestAdresse.testAdresse;
QTestPerson person = new QTestPerson("TP");
queryFactory.select(person.name, person.vorname)
  .from(testPerson)
    .innerJoin(person.testAdresses, testAdresse)
    .innerJoin(testAdresse.testPerson(),testPerson)
  .where(testPerson.testName().name.eq("Gordon"));
```

Listing 5: Formulierung eines JOINs über mehrere Tabellen

Erweiterung von QueryDSL

Das Erweiterungskonzept von QueryDSL weist in zwei Richtungen.

- Zum einen wird ein Mechanismus bereitgestellt, mit dessen Hilfe neue Ausdrücke definiert werden können. Mit Templates lässt sich der Sprachumfang erweitern. Dieser Mechanismus erlaubt es auch, datenbank- oder dialektspezifische Ausdrücke einzusetzen.
- Ein weiteres Erweiterungsverfahren ermöglicht es, auf die generierten Query Types Einfluss zu nehmen und diese um Funktionalität oder Path-Ausdrücke zu erweitern.

Im Weiteren werden diese beiden Mechanismen vorgestellt.

TemplateExpressions

Das Konzept der TemplateExpressions erlaubt es, eigene Ausdrücke zu definieren. Diese Ausdrücke beziehen sich selbst wiederum auf andere Ausdrücke, in der Regel Path-Ausdrücke. TemplateExpressions bietet quasi einen Baukasten, um eigene Ausdrücke aufzubauen.

```
QTestPerson testPerson = QTestPerson.testPerson
StringExpression nameTemplate =
  Expressions.stringTemplate("FUNCTION('UPPER',{0})", testPerson.name);
```

Listing 6: Aufruf einer Datenbankfunktion mit Hilfe einer TemplateExpression

TemplateExpressions werden wahrscheinlich am häufigsten eingesetzt, um Datenbankfunktionen oder Stored Procedures anzubinden.

Ab JPA 2 wird ein Aufruf einer Datenbankfunktion in JPQL mit dem Schlüsselwort `FUNCTION` eingeleitet. Beispielsweise wird durch den Ausdruck `FUNCTION('UPPER',testPerson.name)` die Datenbankfunktion `UPPER` auf `testperson.name` angewandt. Diese Ausdrücke sind datenbankspezifisch und werden daher als TemplateExpression repräsentiert.

Listing 6 zeigt, wie dieser Ausdruck in QueryDSL formuliert wird, und Listing 7 stellt dar, wie dieser Ausdruck so-

wohl in der SELECT- als auch in der WHERE-Clause eingesetzt wird.

```
queryFactory
  .select(nameTemplate)
  .from(QTestPerson.testPerson)
  .where(nameTemplate.eq("GORDON"))
```

Listing 7: Einsatz einer TemplateExpression

Erweiterung mit @QueryDelegate

Die Path-Ausdrücke wurden in Query Types generiert. Die Generierung stützt sich auf die Informationen aus den Entity-Klassen. QueryDSL-JPA bietet die Möglichkeit, die Query Types durch zusätzliche Funktionalität anzureichern. So können statische Methoden annotiert werden, sodass Generatoren diese Methode aufgreifen und die Generate der Query Types erweitern.

Eine detaillierte Darstellung dieses Konzepts sprengt den Rahmen des Artikels. Beispiele und Erläuterungen finden sich unter [LUISFPG] und [CSC15].

Dieser Mechanismus steht leider für die Integration mit SQL nicht zur Verfügung.

Weiterführende Konzepte

Neben den grundlegenden Konzepten in JPQL werden auch weiterführende Konzepte durch die Klasse **JPAQuery** unterstützt. Für interessierte Leser sind dies im Einzelnen: *Built-in Functions*, *Fetch Joins*, *JPQL Collection Member Declarations*, *JPQL Aggregate Functions*, *JPQL Subqueries*, *JPQL All* und *Any Expressions* [JPARef].

Auch Konzepte wie Vererbung oder *Embeddables* der Domänenobjekte setzt QueryDSL konsistent um, und sie stehen wie erwartet bei der Formulierung der Abfragen zur Verfügung.

Darüber hinaus erlaubt es QueryDSL auch, Abfragen gegen *Native SQL* zu formulieren [JPARef]. Native SQL ist neben JPQL und dem Criteria API die dritte Möglichkeit, Abfragen in JPA zu formulieren. Native SQL erlaubt eine deutlich datenbanknähere Formulierung der Abfragen und es kommt ins Spiel, wenn datenbankspezifische Sprachelemente wie Hints genutzt werden müssen.

Unterstützung von JPQL-Dialekten

Aufgrund des im Wesentlichen standardisierten Sprachumfangs von JPQL ist der Unterschied zwischen den einzelnen JPA-Providern nicht so groß wie der Unterschied der einzelnen SQL-Dialekte. Allerdings existieren auch in JPQL einzelne Dialekte. So bietet Hibernate eine Erweiterung von JPQL namens *HQL* [HQL] an. Derzeit unterstützt QueryDSL neben Standard-JPQL noch diesen Dialekt.

Datenmanipulationen

Es überrascht nicht, dass auch Datenmanipulationen unterstützt werden. So können Statements für DELETE, INSERT und UPDATE erzeugt werden.

QueryDSL und Spring Data

Den Abschluss des Überblicks über die Integration mit JPA bildet ein Ausflug in das Spring-Projekt *spring-data-(jpa)*. Dort wird QueryDSL eingesetzt, um dynamische Suchkriterien für JPA-basierte Suchen zu formulieren.

Zugriffsmethoden auf die Domäne werden lediglich in Interfaces namens *Repositories* deklariert. spring-data interpretiert diese Methoden und führt entsprechende Zugriffe mittels JPA aus [SpringData]. Repositories erweitern spezifische Marker-Interfaces vom Typ **Repository**. Spring durchsucht gegebene Packages nach diesen Interfaces und stellt zu jedem gefundenen Interface eine Implementierung bereit (sogenannte *Executoren*). Diese Executoren sind in der Lage, die Methodendeklarationen der Repositories zu interpretieren.

Neben den Operationen zum Schreiben, Löschen und Verändern bieten Repositories die Möglichkeit, nach beliebigen Kriterien zu suchen. Die Suchkriterien können per Namenskonvention sowie per parametrisierter JPQL-Statements (via Annotation) oder NamedQueries formuliert werden. Selbst Prädikate des JPA Criteria API können genutzt werden.

QueryDSL besitzt eine vergleichbare Ausdruckskraft wie das JPA Criteria API, lässt sich aber deutlich einfacher formulieren. Ein spezielles Interface namens **QueryDslPredicateExecutor** erweitert ein Repository um die QueryDSL-Unterstützung (Listing 8). Analog zur Unterstützung des JPA Criteria API in spring-data-jpa kann eine Abfrage durch ein QueryDSL-Prädikat parametrisiert werden (Listing 9, Listing 10). Ein solches Prädikat ist im Prinzip eine Expression vom Typ **Boolean**. Mit Hilfe der Klasse **BooleanBuilder** kann ein Prädikat aus mehreren Prädikaten zusammengebaut werden.

```
public interface PersonenRepository extends
  JpaRepository<TestPerson,Long>,
  QueryDslPredicateExecutor<TestPerson>
{
}
```

Listing 8: Deklaration des Repositories inklusive der Unterstützung für QueryDSL-Prädikate

```
public class QueryDslPrecidates {
  public static BooleanExpression nameNotNull =
    QTestPerson.testPerson.name.isNotNull();
}
```

Listing 9: Definition der QueryDSL-Prädikate

```
@Inject PersonenRepository repository;
@Test
public void testReposityAccess() throws Exception {
  Assert.assertFalse(
    repository.findAll(QuerydslPrecidates.nameNotNull).isEmpty());
}
```

Listing 10: Zugriff auf das Repository mittels eines QueryDSL-Prädikats

Wie auch die Integration in spring-data-jdbc-core basiert die Integration spring-data-jpa auf der Version 3.6.2 und noch nicht auf der aktuellen Version 4.0.1.

Fazit

QueryDSL wird seinem Anspruch gerecht, eine leicht erlernbare Lösung zur Formulierung von typsicheren Abfragen für JPQL (und anderen SQL-nahe Sprachen) zu sein. Sehr schnell sind die ersten Abfragen in JPQL formuliert. Das Arbeiten mit QueryDSL ist intuitiv und auch weiterführende Konzepte sind stimmig.

QueryDSL muss den Vergleich mit anderen Frameworks nicht scheuen. Die Unterstützung der Features des Standards JPQL (JPA 2.2) lässt keine Wünsche offen. Die dynamische Formulierung einer JPQL-Abfrage ist mittels QueryDSL deutlich einfacher und intuitiver als mit dem JPA Criteria API. Ein weiteres Plus ist die Integration mit spring-data.

In diesem Artikel kann der Umfang von QueryDSL-JPA nur angerissen werden. Weiterführende Informationen finden Sie auf [QRef]. Zu Fragen, die nicht in der Dokumentation beantwortet werden, findet sich oft Hilfe auf `http://stackoverflow.com/`.

Begleitend zu diesem Artikel existiert ein GitHub-Projekt [CSC15], in dem sich zu allen angesprochenen Konzepten Codebeispiele finden.

Links

[CSC15] Projekt mit begleitenden Codebeispielen, `https://github.com/csc19601128/misc-examples.git`

[Fow05] `http://martinfowler.com/bliki/FluentInterface.html`

[Hibernate] `http://hibernate.org/orm/`

[HQL] `http://docs.jboss.org/hibernate/orm/5.0/userGuide/en-US/html/ch13.html`

[LUISFPG] Luis Fernando Planella Gonzalez, `http://luisfpg.blogspot.de/2013/02/the-beauty-of-Querydsl-in-sorting.html`

[JPARef] JSR-000338 Java™ Persistence 2.1, `https://jcp.org/aboutJava/communityprocess/final/jsr338/index.html`

[MYSEMA1] `http://blog.mysema.com/2010/04/querydsl-as-alternative-to-jpa-2.html`

[Querydsl] `http://www.querydsl.com/`

[QRef] `http://www.querydsl.com/static/querydsl/4.0.1/reference/html_single/`

[Schm15] Ch. Schmidt-Casdorff, Just Married – SQL und JPA: QueryDSL, in: JavaSPEKTRUM, 5/2015

[SpringData] `http://docs.spring.io/spring-data/jpa/docs/current/reference/html`

Christoph Schmidt-Casdorff ist als IT-Berater bei der IKS Gesellschaft für Informations- und Kommunikationssysteme tätig. Er beschäftigt sich seit mehreren Jahren in großen Kundenprojekten mit dem Thema der modellgetriebenen Softwareentwicklung und mit flexiblen Softwarearchitekturen mit JEE, Spring und OSGi.
E-Mail: c.schmidt-casdorff@iks-gmbh.com