Just Married – SQL und JPA

QueryDSL

Christoph Schmidt-Casdorff

In der derzeitigen IT-Landschaft spielen relationale Datenbanksysteme immer noch die entscheidende Rolle. Die Sprache zur Datenabfrage und -manipulation ist SQL. QueryDSL tritt an, SQL und weitere SQL-nahe Abfragesprachen unter einem gemeinsamen Konzept in Java zu integrieren. Dieser Artikel beschäftigt sich mit der Integration von QueryDSL und SQL.

Konzepte

Mit *QueryDSL* lassen sich nach eigenem Anspruch typsichere, SQL-nahe Abfragen formulieren. Diese Abfragen werden nicht – wie häufig immer noch gängige Praxis – durch Konkatenierung von Zeichenketten gebaut, sondern durch ein geschickt entworfenes API (*Application Programming Interface* – Schnittstelle der Anwendung) intuitiv zusammengestellt. QueryDSL bedient sich der Form des *fluent API* [Fow05].

QueryDSL bezieht sich also nicht nur auf SQL, sondern auch auf SQL-nahe Abfragen, wie Abfragen in Lucene, MongoDB, Hibernate Search oder Collections [Querydsl]. Für all diese Einsatzmöglichkeiten bietet QueryDSL eine einheitliche, domänenspezifische Sprache (DSL) an.

Obwohl die Abfragesprachen aus unterschiedlichsten Bereichen (*Domänen*) stammen, ist es durch eine geschickte Abstraktion gelungen, alle Abfragesprachen unter ein gemeinsames DSL-Konzept zu bringen.

Um diese ersten abstrakten Aussagen zu konkretisieren, ist nichts zuträglicher, als sich Beispiele anzuschauen.

Abfragen in SQL

Eine erste SQL-Abfrage

Listing 1 spricht für sich selbst und es ist offensichtlich, welches SQL-Statement erzeugt wird (s. Listing 2). Es zeigt, wie intuitiv sich das zu erzeugende SQL-Statement in dem API wiederfindet.

```
List<String> names = queryFactory.select(testPerson.name)
  .from(testPerson)
  .where(testPerson.name.isNotNull())
  .fetch();
```

Listing 1: Abfrage in QueryDSL

```
select "TEST_PERSON"."NAME"
  from "TEST_PERSON"
  where "TEST_PERSON"."NAME" is not null
```

Listing 2: SQL-Statement, welches in Listing 1 erzeugt wird

Mit dem Aufruf der Methode `fetch` ist der Aufbau der Abfrage abgeschlossen und das Statement wird aufgebaut. Die syntaktische Prüfung übernimmt der zuständige JDBC-Treiber. Grundsätzlich ist QueryDSL nur für den Aufbau des Statements zuständig. Syntaxprüfungen werden immer durch das Zielsystem vorgenommen, in diesem Fall dem JDBC-Treiber.

Konfiguration

Die Struktur eines SQL-Statements wird durch die sogenannten *SQL Clauses* wie FROM, WHERE, GROUP BY usw. bestimmt. Ein QueryDSL-Statement wird durch eine Instanz von `AbstractSQLQuery` (im Weiteren kurz `Query`) repräsentiert. Diese Instanz stellt den Rahmen für die SQL Clauses bereit und bietet das API für den Aufbau des SQL-Statements an.

SQL ist leider nicht in dem Maß standardisiert, dass alle SQL-Datenbanken den gleichen SQL-Befehlsumfang unterstützen. Der SQL-Dialekt wird daher bei der Konfiguration einer Abfrage angegeben (`SQLTemplates`). In Listing 3 wird der Dialekt für *H2* eingesetzt. Für die gängigsten Datenbanken existieren entsprechende `SQLTemplates`.

Zusammen mit einer *DataSource*, welche den Zugriff auf die Datenbankverbindung bereitstellt, wird eine Umgebung zur Generierung von SQL-Statements für einen bestimmten SQL-Dialekt vorbereitet (s. Listing 3).

```
DataSource dataSource = ...
SQLTemplates dialect =
  H2Templates.builder().printSchema().quote().build();
Configuration configuration = new Configuration(dialect);
SQLQueryFactory queryFactory =
  new SQLQueryFactory(configuration, dataSource);
```

Listing 3: Umgebung zur Generierung von SQL-Statements für einen bestimmten SQL-Dialekt

Query Types – Anbindung an die Domäne

Für die Integration von SQL in QueryDSL spielen Spalten und Tabellen die Rolle der Domänenobjekte. Grundsätzlich werden Verweise auf Domänenobjekte als *Path* bezeichnet. Überall dort, wo in einem SQL-Statement eine Spalte oder Tabelle zu finden ist, muss ein entsprechender Path-Ausdruck angegeben werden. Path ist hierarchisch ausgebaut. So ist eine Spalte einer Tabelle und diese wiederum (falls angegeben) einem Schema untergeordnet.

Datenbankspalten sind typisiert und der Typ kann unmittelbar aus der Definition der Spalten in der Datenbank abgeleitet werden. QueryDSL setzt einen Generator ein, um diese Informationen aus der Datenbank zu extrahieren und in eigenen Klassen zusammenzufassen. Der zugehörige Codegenerator [Querydsl] kann mittels des Buildprozesses (maven, ant) oder explizit aufgerufen werden.

Es wird für jede Tabelle eine Klasse erzeugt, welche die Metadaten der Domänenobjekte enthält. Die Standardkonvention für die Benennung stellt ein Q vor den Tabellennamen. So wird aus der Tabelle `TEST_PERSON` die generierte Klasse `QTestPerson`. Diese generierten Klassen zur Anbindung an die Domäne heißen *Query Type*.

Zu jeder Spalte wird ein passend typisierter Path-Ausdruck erzeugt und zum Primärschlüssel wird ein Path vom Typ `PrimaryKey` (Listing 4) generiert.

```
public class QTestPerson extends RelationalPathBase<QTestPerson> {
  public static final QTestPerson testPerson =
    new QTestPerson("TEST_PERSON");
  public final StringPath name = createString("name");
  public final com.querydsl.sql.PrimaryKey<QTestPerson> constraint8 =
    createPrimaryKey(sidTestPerson);
```

Listing 4: Generierte Klasse der Metadaten für die Tabelle TEST_PERSON

JOINs

Die Nutzung von QueryDSL zum Aufbau einfacher Statements ist in [Querydsl] gut beschrieben. Komplexer ist da schon der

Aufbau von JOINs, denn hier spielen naturgemäß die Path-Ausdrücke eine wichtige Rolle.

Um SQL-JOINs formulieren zu können, muss die relationale Verbindung zwischen zwei Tabellen bekannt sein. In Query-DSL werden dazu natürlich Query Types genutzt. Diese Verbindung kann explizit über die konkreten Kriterien angegeben werden (s. Listing 5).

```
QTestPerson testPerson = QTestPerson.testPerson;
QTestAdresse testAdresse = QTestAdresse.testAdresse;
queryFactory.select(testAdresse)
  .from(testAdresse)
  .innerJoin(testPerson)
    .on(testPerson.sidTestPerson.eq(testAdresse.sidTestAdresse));
```

Listing 5: Formulierung eines INNER JOINs

Es werden INNER JOIN, LEFT und RIGHT OUTER JOIN sowie FULL OUTER JOIN unterstützt (zur Erläuterung siehe [TECH]).

Jede Instanz einer Q*-Klasse repräsentiert eine Tabelleninstanz im SQL-Statement. In der Regel reichen die durch den Generator erzeugten statischen Instanzen aus. Sollen aber mehrere Instanzen der gleichen Tabelle genutzt werden oder soll Tabellen ein Alias zugewiesen werden, so werden unterschiedliche Instanzen benötigt.

Um einen JOIN zu beschreiben, in dem die Tabelle **`TEST_PERSON`** mehrfach verjoint wird, muss diese Tabelle mehrfach instanziert werden. Daher muss auch die Query Type-Klasse mehrfach instanziert werden (s. Listing 6).

```
QTestPerson testPerson = QTestPerson.testPerson;
QTestAdresse testAdresse = QTestAdresse.testAdresse;
QTestPerson person = new QTestPerson("TP");
queryFactory.select(person.name, testPerson.vorname)
  .from(testPerson)
  .innerJoin(person)
    .on(person.sidTestPerson.eq(testAdresse.sidTestPerson))
  .innerJoin(testAdresse).
    .on(testAdresse.sidTestPerson.eq(testPerson.sidTestPerson));
```

Listing 6: Formulierung eines JOINs über mehrere Tabellen

Projektionen

Neben der Beschreibung der Suchkriterien ist die Definition der Ergebnisspalten der zweite wichtige Abschnitt einer SQL-Abfrage.

Von einem objektorientierten Framework zur Formulierung von SQL-Abfragen darf man erwarten, dass es in diesem Punkt eine umfangreiche Unterstützung anbietet. Neben der Aufzählung der erwarteten Spalten (Projektion) muss auch eine Abbildung aus der Datenbank in die objektorientierte Welt möglich sein (ORM – *Object Relational Mapping*). Ergebnisse einzelner Spalten werden ihrem Java-Typ zugewiesen, oder Mengen von Ergebnisspalten können auf Beans abgebildet werden. Ganz allgemein gesprochen findet eine Transformation der Abfrageergebnisse statt.

Das Abfrageergebnis enthält immer eine Projektion auf die Domänenobjekte (Auswahl der Spalten) und eine Transformation in die Java-Welt .

Ein Blick auf die Arbeitsweise der **`Query`** hilft, die Umsetzung dieses Konzepts aus Projektion und Transformation besser zu verstehen.

Die Methode **`select`** erzeugt eine durch den Ergebnistyp der Abfrage parametrisierte Instanz der Klasse **`Query`** (s. Listing 7). Liefert die Abfrage genau eine Spalte, so wird die **`Query`** durch den Typ der Spalte parametrisiert.

Liefert sie eine Menge an Spalten, so wird sie durch den Typ **`Tuple`** dargestellt. **`Tuple`** ermöglicht einen Zugriff auf die einzelnen Spalten über deren Namen (s. Listing 7). Durch die Parametrisierung der **`Query Types`** sind auch die Ergebnisse des Zugriffs auf **`Tuple`** typisiert. Wie oben beschrieben, ist **`testPerson.name`** ein typisierter Path-Ausdruck.

```
Query<Tuple> query = queryFactory.select(Projections.tuple(
  testPerson.vorname,testPerson.name));
List<Tuple> result = query.fetch();
for (Tuple tuple : result) {
  String name = tuple.get(testPerson.name);
  String vorname = tuple.get(testPerson.vorname);
}
```

Listing 7: Projektion der Ergebnismenge

In Listing 7 wird eine Projektion der angegeben Spalten, gefolgt von einer Transformation in den Ergebnistyp **`Tuple`** beschrieben. In der Klasse **`Projections`** finden sich weitere Projektionen, welche den Ergebnistyp auf Listen, Maps oder andere Java-Typen abbilden.

Weitere Projektionen ermöglichen eine Abbildung der Spalten auf Attribute von Beans (mittels **`Projections.beans`**) oder ermöglichen die Bereitstellung eines sogenannten *ResultTransformers*, welche die Möglichkeit bieten, beliebige Transformationen auf der Ergebnismenge durchzuführen.

Subqueries

Subqueries sind ein integraler Bestandteil von SQL und kommen dort zum Einsatz, wo eine Abfrage die Rolle eines anderen SQL-Ausdrucks einnimmt.

Der häufigste Einsatz von *Subqueries* in einer SQL-Abfrage ist das sogenannte *Subselect* oder auch Unterabfrage. In Listing 8 (Beispiel 1) wird eine Subquery als Unterabfrage genutzt, um den Wert des Feldes **`testPerson.sidTestPerson`** einzuschränken.

Beispiel 2 aus Listing 8 zeigt den Einsatz einer Subquery als Stellvertreter einer FROM-Clause (sogenannter *Inline View*). Statt einer Tabelle wird dort die Subquery referenziert.

Im dritten Beispiel in Listing 8 wird der Aufbau einer UNION-Abfrage dargestellt. In QueryDSL sind Subqueries Bausteine für Unions. In **`query.union`** werden mehrere Subqueries zu einem UNION-Statement zusammengefasst.

Auch weiterführende Konzepte wie Inline Views oder WITH-Clause (sogenannte *common table expressions*) werden unterstützt, soweit der entsprechende SQL-Dialekt diese zulässt.

```
SQLQuery<Long> subQuery = new SQLQuery<Long>(dialect)
  .select(testAdresse.sidTestPerson)
    .from(testAdresse).where(testAdresse.sidTestPerson.isNotNull());

// Bsp 1 : Subquery als Subselect
List<Long> result1 =
  query.select(testPerson.sidTestPerson)
    .from(testPerson).where(testPerson.sidTestPerson.in(subQuery))
      .fetch();

// Bsp 2 : Subquery als Inline View
QTestAdresse t = new QTestAdresse("T");
List<Long> result2 =
  query.select(t.sidTestPerson).from(subQuery, t).fetch();

// Bsp 3 : Subquery als UNION
Union<Long> union = query.<Long>union(subQuery, subQuery);
List<Long> result3 = union.list();
```

Listing 8: Beispiele zur Formulierung von Subqueries

Integration in Spring

Im Springprojekt `spring-data-jdbc-core` wird QueryDSL in Spring integriert. Im Wesentlichen wird ein Template bereitgestellt, welches den Rahmen zur Ausführung von QueryDSL-Abfragen oder Datenmanipulationen bietet. Die Technik ist bereits etabliert und weist große Ähnlichkeit mit dem allseits bekannten JDBCTemplate aus Spring auf. Ein Wermutstropfen bleibt allerdings: QueryDSL ist mit der Version 3.6.2 integriert und noch nicht in der aktuellen Version 4.0.1.

Datenmanipulationen

Es überrascht nicht, dass auch Datenmanipulationen unterstützt werden. So können Statements für DELETE, INSERT und UPDATE erzeugt werden. Es wird sogar der Batch-Mechanismus von JDBC unterstützt [Querydsl].

Erweiterbarkeit der SQL-Abfragen

QueryDSL verfolgt nicht den Ansatz, alle SQL-Dialekte von Haus aus vollständig zu unterstützen. Um aber den unterschiedlichen Dialekten gerecht zu werden, bietet QueryDSL verschiedene Erweiterungsmechanismen an.

Dialektunterschiede, die keinen Einfluss auf das API haben, wie Abbildung der Datenbanktypen auf Java-Typen, werden in `SQLTemplates` beschrieben (s. Unterkapitel „Konfiguration“).

Um das API zu erweitern, bietet QueryDSL dialektspezifische Implementierungen von `AbstractSQLQuery` an. *OracleQuery* unterstützt beispielsweise die Oracle-spezifischen sogenannten hierarchischen Abfragen [Oracle]. Die entsprechenden SQL Clauses wie CONNECT BY, START WITH werden nur in OracleQuery durch entsprechende API-Methoden unterstützt. Durch eine Erweiterung dieser Klasse kann das API dialektspezifisch erweitert werden.

Ein weiterer Erweiterungsmechanismus wird durch *TemplateExpression* beschrieben. Dieses Konzept erlaubt es, eigene Ausdrücke zu beschreiben und in Statements zu nutzen. Listing 9 zeigt, wie die Funktion UPPER als TemplateExpression formuliert und eingesetzt werden kann.

```
OracleQuery<String> query =...
StringExpression upper = Expressions.stringTemplate("UPPER({0})",
                         QTestPerson.testPerson.name);
List<String> names = query.select(upper)
  .from(QTestPerson.testPerson)
  .where(QTestPerson.testPerson.name.eq("Gordon"))
  .fetch();
```

Listing 9: Einsatz einer TemplateExpression, um eine Funktion aufzurufen

Der Mechanismus der TemplateExpression ist ein Schlüsselkonzept zur Erweiterung. Er bietet die Möglichkeit, neue Ausdrücke durch beliebige Kombinationen bestehender zu beschreiben. Am häufigsten werden TemplateExpressions eingesetzt, um Funktionen oder *Stored Procedures* aufzurufen.

Ausblicke

QueryDSL bietet ein einheitliches Konzept, um unterschiedliche SQL-nahe Sprachen zu integrieren.

Die offene Architektur von QueryDSL ermöglicht neben SQL auch eine Unterstützung weiterer Abfragen gegen JPQL (JPA), Lucene, MongoDB, Hibernate Search oder Collections (siehe [Querydsl]). Das QueryDSL-API bleibt im Wesentlichen für die unterschiedlichen Integrationsszenarien gleich. Die unterschiedlichen Sprachanbindungen unterscheiden sich in der Implementierung der `Templates` und der `Query`.

In diesem Artikel kann der Umfang von QueryDSL nur angerissen werden. Leider ist die Dokumentation zwar für die wichtigsten Anwendungsfälle ausreichend, aber weiterführende Konzepte sind nur stiefmütterlich beschrieben. Zu Fragen, die nicht in der Dokumentation beantwortet werden, findet sich oft Hilfe auf `http://stackoverflow.com/`. Eine weitere gute Quelle zum tieferen Verständnis sind die ausführlichen Tests der einzelnen Komponenten.

Begleitend zu diesem Artikel existiert ein GitHub-Projekt [CSC15], in dem sich zu allen angesprochenen Konzepten (inkl. der Integration mit JPA) Codebeispiele finden.

Fazit

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

Alles in allem ist QueryDSL eine gute Lösung zur Formulierung von SQL-nahen Abfragen und muss auch den Vergleich mit anderen Frameworks nicht scheuen. Im Gegensatz zum Platzhirsch der SQL-Frameworks jOOQ [jOOQ] unterstützt QueryDSL nicht alle SQL-Dialekte von Hause aus. Dort ist also eventuell QueryDSL zu erweitern.

QueryDSL ist ausschließlich Open Source (Apache License 2.0) und damit im Vorteil gegenüber Bewerbern wie jOOQ.

Falls Sie sich für Frameworks zur Formulierung von Abfragen in SQL-nahen Sprachen interessieren, so sollten Sie QueryDSL auf jeden Fall in die engere Wahl nehmen.

Referenzen

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

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

[jOOQ] `http://www.jooq.org/`

[Oracle] Hierarchical Queries in Oracle 11g, `http://docs.oracle.com/cd/B28359_01/server.111/b28286/queries003.htm`

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

[SpringDataJDBC] `http://docs.spring.io/spring-data/jdbc/docs/current/reference/html/core.querydsl.html`

[TECH] Erläuterung zu den unterschiedlichen JOIN-Typen, `http://www.techonthenet.com/oracle/joins.php`

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