Alternativen zum Criteria-API

Nicolai Mainiero, sidion GmbH

Auch wenn es mit Spring Data oder Panache von Quarkus gut funktionierende Abstraktionen für den Zugriff auf die Datenbank gibt, kommt es manchmal zu Situationen, bei denen diese in dem, was über ihre APIs möglich ist, an ihre Grenzen stoßen. Sobald die where-Clause dynamisch zusammengebaut werden soll, muss in Spring Data auf Specifications und in Panache auf ungetypte Stringkonkatination ausgewichen werden. Falls noch komplexere Abfragen benötigt werden, kann auf das Criteria-API ausgewichen werden. Beides keine befriedigende Option, vor allem da das Criteria-API die Komplexität sowohl beim Schreiben als auch beim Lesen des Codes unnötig erhöht.

Beim lesenden Zugriff auf eine Datenbank ist es einfach, die Abfrage in SQL zu schreiben beziehungsweise sich durch SQL iterativ an die richtige Abfrage anzunähern. Die Umwandlung von SQL in Java-Code stellt dann aber eine große Herausforderung dar. Während einfache Abfragen mit den APIs von Spring Data oder Panache auch schnell umgesetzt werden können, kommt man bei komplexeren Abfragen mit mehreren Joins oder sogar Unions schnell an die Grenze dessen, was diese APIs unterstützen. An dieser Stelle muss dann auf die Grundlagen des Java Persistence API (JPA) *[1]* zurückgegriffen werden. Dieses ist nach wie vor das Standard-API in Java-SE- und Java-EE-Anwendungen für die Verwaltung der Persistenz und der Objekt-/Relationalzuordnung (ORM). Sie steht damit auch in allen Applikationen zur Verfügung, die zum Beispiel Spring Data oder Panache verwenden oder einen JPA Provider wie Hibernate *[2]* direkt integrieren.

Das bedeutet, innerhalb dieser Applikationen gibt es zwei zusätzliche Möglichkeiten, um Datenbankabfragen zu erstellen, zum einen JPQL (Java Persistence Query Language) und zum zweiten das Criteria-API. Bei JPQL kommt es ähnlich wie bei Panache schnell zu dem Problem, dass dynamische Abfragen durch Stringkonkatination zusammengebaut werden müssen und man Typsicherheit verliert. Außerdem eröffnet dies die Möglichkeit für Injection-Angriffe *[3]*, die es zu verhindern gilt. Als zweite Möglichkeit gibt es das Criteria-API, um sicher dynamische und komplexe Datenbankabfragen zu erzeugen. Im Folgenden sollen verschiedene Alternativen mit dem Criteria-API verglichen werden. Dazu wird folgendes Datenbankschema *(siehe Abbildung 1)* verwendet. Es handelt sich um ein stark vereinfachtes Modell eines Händlers mit Kunden, Produkten und Bestellungen.

Die folgenden beiden in *Listing 1* beschriebenen SQL-Abfragen sollen über das jeweilige API ausgeführt werden.
Listing 2 zeigt die Implementierung der Abfrage in Beispiel 1 mit dem Criteria-API. Zunächst wird ein `CriteriaBuilder` vom `EntityManager` geholt, dann eine `CriteriaQuery` für die Entität Customer erzeugt, um dann die where-Clause mithilfe des `CriteriaBuilder` (1) zu erzeugen. Am Ende wird die Abfrage erstellt und dann ausgeführt.

Zu erkennen ist, dass viel Setup-Code notwendig ist, um diese einfache Abfrage auszuführen. Außerdem ist es beim Lesen des Codes schwierig herauszufinden, welches SQL-Statement gegen die Datenbank schlussendlich ausgeführt wird. Die Implementierung von Bei-

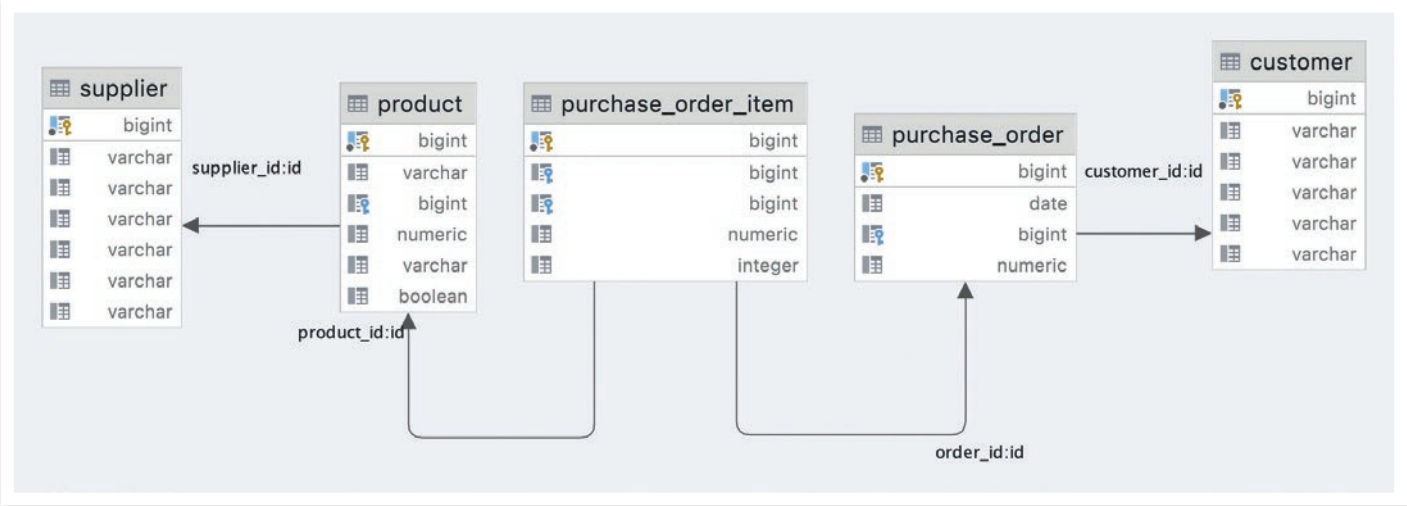

Abbildung 1: Datenbankschema (© Nicolai Mainiero)

```
SELECT * from customer WHERE first_name = :first_name and last_name = :last_name; Beispiel 1

SELECT * from purchase_order p JOIN customer c on p.customer_id = c.id WHERE last_name = :last_name; Beispiel 2
```

Listing 1: Beispiel 1 und 2 der SQL-Abfragen

```
public Customer findByName(String firstName, String lastName) {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
  Root<Customer> customerRoot = cq.from(Customer.class);
  Predicate where = cb.and(cb.equal(customerRoot.get(Customer_.FIRST_NAME), firstName), cb.equal(customerRoot.
get(Customer_.LAST_NAME), lastName)); // 1
  CriteriaQuery<Customer> customerCriteriaQuery = cq
      .select(customerRoot)
      .where(where);
  return em.createQuery(customerCriteriaQuery).getSingleResult();
}
```

Listing 2: Beispiel 1 mit Criteria-API (zweiteilige where-Clause)

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<PurchaseOrder> query = cb.createQuery(PurchaseOrder.class);
  Root<PurchaseOrder> order = query.from(PurchaseOrder.class);
  Join<PurchaseOrder, Customer> customer = order.join(PurchaseOrder_.CUSTOMER);
  query.select(order).where(cb.equal(customer.get(Customer_.LAST_NAME), lastName));
  return em.createQuery(query).getSingleResult();
}
```

Listing 3: Beispiel 2 mit Criteria-API (join)

```
public Customer findByName(String firstName, String lastName) {
  SQLDialect configuration = SQLDialect.POSTGRES;
  SelectConditionStep<Record> query = DSL.using(configuration)
      .select()
      .from(CUSTOMER)
      .where(CUSTOMER.FIRST_NAME.eq(firstName).and(CUSTOMER.LAST_NAME.eq(lastName))); // 1
  return nativeQuery(em, query, Customer.class);
}
```

Listing 4: Beispiel 1 mit jOOQ

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  SQLDialect configuration = SQLDialect.POSTGRES;
  SelectConditionStep<Record> query = DSL.using(configuration)
      .select()
      .from(PURCHASE_ORDER)
      .join(CUSTOMER).on(PURCHASE_ORDER.CUSTOMER_ID.eq(CUSTOMER.ID))
      .where(CUSTOMER.LAST_NAME.eq(lastName)); // 1
  return nativeQuery(em, query, PurchaseOrder.class);
}
```

Listing 5: Beispiel 2 mit jOOQ

spiel 2 ist in *Listing 3* zu sehen. Zusätzlich zu der where-Clause ist hier auch noch ein Join notwendig, um die entsprechende Abfrage zu realisieren. Auch hier ist zu erkennen, dass viel Setup-Code benötigt wird, bis die eigentliche Abfrage formuliert und ausgeführt werden kann.

Anhand dieser beiden einfachen Beispiele werden drei Alternativen zum Criteria-API genauer betrachtet.

jOOQ

Die jOOQ-Bibliothek *[4]* verfolgt im Gegensatz zu JPA den Ansatz, dass zunächst die Datenbank modelliert und dann auf dieser Basis Java-Code erzeugt wird, mit dem Datenbankabfragen mithilfe eines Fluent-API beschrieben werden können. Dieses Vorgehen spielt vor allem bei bestehenden Datenbanken seine Vorteile aus, um dann in der weiteren Entwicklung nur mit dem jOOQ-API zu arbeiten. jOOQ bietet aber auch die Möglichkeit, aus bestehenden JPA-Entitäten das Metamodell von jOOQ zu erstellen, um dann sowohl JPA als auch das API von jOOQ zu kombinieren.

Dadurch können Abfragen typsicher und dynamisch erzeugt werden. Eine beliebte Kombination ist, die Daten mithilfe von JPA in die Datenbank zu schreiben und mit jOOQ oder einer der Alternativen zu lesen. Indem man das Schreiben der Daten an den Persistence-Provider delegiert, kann man sich den Aufwand, korrekte Insert- beziehungsweise Update-Statements zu schreiben, sparen. JPA kümmert sich darum, dass sowohl alle Felder der Entitäten berücksichtigt werden als auch die Reihenfolge der Inserts/Updates bei komplexen Objektgraphen korrekt ist. *Listing 4 und 5* zeigen die Implementierung der bekannten Beispielabfragen aus *Listing 1* mithilfe von jOOQ.

Wie bereits erwähnt, bietet jOOQ eine DSL, die SQL typsicher in Java nachbildet. Dadurch wird es sehr einfach, die Abfrage (1) in Java zu formulieren. Auch beim Lesen des Codes ist, wenn man SQL beherrscht, schnell klar, welche Abfrage ausgeführt werden wird.

Auch im zweiten Beispiel in *Listing 4* ist die Abfrage sofort erkennbar und erinnert stark an das äquivalente SQL-Statement. Sowohl die Tabelle, mit der das Join stattfindet, als auch die on-Bedingung werden explizit aufgeführt.

JPAStreamer

JPAStreamer ist noch eine recht junge Bibliothek, Version 1.0 erschien erst im Januar 2021, die einen interessanten Ansatz verfolgt. Datenbankabfragen werden mithilfe des Java-Stream-API beschrieben und können dann direkt mit den bekannten Methoden wie `filter()`, `sort()` oder `limit()` manipuliert werden.

Natürlich wird dabei nicht die gesamte Datenbank in den Speicher geladen. Der Ausdruck wird in ein entsprechendes SQL-Statement um-

gewandelt und dann zur Datenbank übermittelt. So wird zum Beispiel aus dem Prädikat in filter() eine where-Clause im SQL-Statement. *Tabelle 1* zeigt, welche Methode in welches SQL äquivalent umgewandelt wird. Für die Beispiele 1 und 2 ergeben sich dann mit dem JPAStreamer-API die Implementierungen, die in den *Listings 6 und 7* abgebildet sind. Die Abfrage nach Vor- und Nachnamen lässt sich direkt auf einen `filter()`-Aufruf abbilden. Hierbei ist zu beachten, dass das Prädikat über das von JPAStreamer erzeugte Metamodell erzeugt und keine Lambdaexpression verwendet wird, da sonst die Filter-Operation (1) nicht auf eine where-Clause abgebildet wird.

In *Listing 7* ist zu sehen, dass dennoch eine Lambdaexpression (1) verwendet worden ist. Das bedeutet, dass bei dieser Abfrage jede `PurchaseOrder` geladen wird, bis ein passendes Ergebnis gefunden worden ist, da die Filterung auf gejointe Tabellen wie von JPAStreamer noch nicht vollständig unterstützt wird. Es existiert aber bereits ein Issue *[5]*, das sich diesem Problem widmet.

Wer Gefallen an JPAStreamer gefunden hat, aber vorrangig mit Scala programmiert, hat mit Slick *[6]* und Quill *[7]* gleich zwei ähnliche Alternativen, die in Scala implementiert worden sind und sich besser in das Ökosystem einpassen.

QueryDSL

Die dritte Alternative ist QueryDSL, eine schon etwas ältere Bibliothek, die mehr als zwei Jahre kaum gepflegt wurde. Im vergangenen Jahr fand sich allerdings eine Gruppe Freiwilliger, die die Pflege übernommen hat. So wurde im Juli 2021 dann Version 5 veröffentlicht *[8]*, die nicht nur Abhängigkeiten aufgeräumt hat, sondern auch eine Menge Bugfixes und sogar neue Features mitgebracht hat. Wie die beiden zuvor beschriebenen Alternativen setzt auch

SQL	Java Stream
FROM	stream()
WHERE	filter() (before collecting)
ORDER BY	sorted()
OFFSET	skip()
LIMIT	limit()
COUNT	count()
GROUP BY	collect(groupingBy())
HAVING	filter() (after collecting)
DISTINCT	distinct()
SELECT	map()
UNION	concat(s0, s1).distinct()
JOIN	flatmap()

Tabelle 1: JPAStreamer Stream-API und SQL-Entsprechungen

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  StreamConfiguration<PurchaseOrder> joining = of(PurchaseOrder.class).joining(PurchaseOrder$.customer);
  Optional<PurchaseOrder> order = jpaStreamer
      .stream(of(PurchaseOrder.class).joining(PurchaseOrder$.customer)) // 1
      .filter(po -> po.getCustomer().getLastName().equals(lastName)) // 1
      .findFirst();
  return order.orElse(null);
}
```

Listing 7: Beispiel 2 mit JPAStreamer

```
public Customer findByName(String firstName, String lastName) {
  QCustomer customer = QCustomer.customer;
  JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(em);
  return jpaQueryFactory.selectFrom(customer)
      .where(customer.firstName.eq(firstName).and(customer.lastName.eq(lastName)))
      .fetchOne(); // 1
}
```

Listing 8: Beispiel 1 mit QueryDSL

```
public Customer findByName(String firstName, String lastName) {
  Optional<Customer> customer = jpaStreamer
      .stream(Customer.class)
      .filter(Customer$.firstName.equal(firstName).and(Customer$.lastName.equal(lastName))) // 1
      .findFirst();
  return customer.orElse(null);
}
```

Listing 6: Beispiel 1 mit JPAStreamer

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  QCustomer customer = QCustomer.customer;
  QPurchaseOrder purchaseOrder = QPurchaseOrder.purchaseOrder;
  JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(em);
  return jpaQueryFactory.selectFrom(purchaseOrder)
      .innerJoin(purchaseOrder.customer, customer)
      .where(customer.lastName.eq(lastName))
      .fetchOne(); // 1
}
```

Listing 9: Beispiel 2 mit QueryDSL

QueryDSL auf die Generierung eines Metamodells, dass wie bei JPAStreamer durch einen Annotation-Prozessor erzeugt wird. Die Abfrage wird dann ähnlich zu jOOQ mithilfe einer DSL beschrieben und ausgeführt.

In *Listing 8* ist wieder die Suche nach einem Kunden anhand des Vor- und Nachnamens implementiert. Dank der DSL lässt sich die Abfrage (1) fast wie SQL schreiben. Zunächst wird die Tabelle ausgewählt (selectFrom), anschließend die where-Clause beschrieben (where) und dann genau ein Datensatz geladen (fetchOne).

Für das Beispiel 2 wird noch ein Join benötigt, die Implementierung ist in *Listing 9* zu sehen. Auch hier ist wieder zu erkennen, wie ähnlich die DSL zu SQL ist. Und wie direkt sich SQL-Statements nach Java übertragen lassen und typsichere Datenbankabfragen ermöglichen. Damit ist auch beim Lesen des Quellcodes direkt klar, welche Abfrage ausgeführt werden wird.

Fazit

Aus meiner Sicht ist JPAStreamer eine kreative Art, das Streams-API zu verwenden, um daraus SQL-Statements zu erzeugen. Die kognitive Last bei der Umwandlung zwischen Stream-API und SQL ist nicht zu vernachlässigen, vor allem bei komplexeren Abfragen. Dennoch denke ich, dass der Großteil der typischen Datenbankabfragen in einer Applikation sehr einfach mit filter() und sorted() abgebildet werden kann. Bei QueryDSL gefällt vor allem die sehr direkte Umwandlung zwischen Java und SQL. Bestehende SQL-Abfragen können praktisch 1:1 nach Java umgeschrieben werden. Das Gleiche gilt auch für jOOQ. Auch hier bietet die DSL die vollständige Kontrolle über das erzeugte SQL-Statement.

Sowohl QueryDSL als auch jOOQ punkten zusätzlich durch die Möglichkeit, die verwendeten Metamodelle aus bestehenden Datenbanken, ohne den Umweg über JPA, zu erzeugen. Damit bilden sie auch ein großartiges Bindeglied in Anwendungen, die sowohl Legacy- als auch JPA-basierte Datenbanken bedienen müssen. Für mich haben sich alle drei Bibliotheken als Möglichkeit erwiesen, komplexe Datenbankabfragen verständlich zu implementieren, ohne den Vorteil von einfachen Updates durch JPA aufzugeben. Der vollständige Quellcode der Beispiele, inklusive der Generierung der Metamodelle, findet sich auf GitHub *[9]*.

Quellen

[1] The Eclipse EE4J Project, „Jakarta Persistence project," [Online]. Available: *https://github.com/eclipse-ee4j/jpa-api*.

[2] Red Hat, „Hibernate ORM," [Online]. Available: *https://hibernate.org/orm/*.

[3] OWASP, „A03:2021 – Injection," [Online]. Available: *https://owasp.org/Top10/A03_2021-Injection/*.

[4] Data Geekery GmbH, „Great Reasons for Using jOOQ," [Online]. Available: *https://www.jooq.org/*.

[5] JPAStreamer, „Optimize Streams that yield Join-operations," [Online]. Available: *https://github.com/speedment/jpa-streamer/issues/43*.

[6] Slick, „ Functional Relational Mapping for Scala," [Online]. Available: *https://scala-slick.org/*.

[7] Quill, „Free/Libre Compile-time Language Integrated Queries for Scala," [Online]. Available: *https://getquill.io/*.

[8] QueryDSL, „QueryDSL Version 5.0.0," [Online]. Available: *https://github.com/querydsl/querydsl/releases/tag/QUERYDSL_5_0_0*.

[9] N. Mainiero, „Comparison of 3 alternatives to the Criteria API," [Online]. Available: *https://github.com/nicolaimainiero/criteria-api-alternatives*.

Nicolai Mainiero
sidion GmbH
nicolai.mainiero@sidion.de

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der sidion GmbH. Er entwickelt seit über vierzehn Jahren Geschäftsanwendungen in Java, Kotlin und Clojure für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen und schaut sich auch abseits des Mainstreams nach interessanten Technologien um.