Micro Batch Services powered by Spring Boot ▸ 58

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 10.2014

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Java EE 7

4 194586 709801 10

Mapping von Entitäten und Relationen als Java-Interfaces

Persistente Objekte in Zeiten von NoSQL

In letzter Zeit wird viel über Entwicklungen im Bereich der Datenbanken gesprochen, die behandelten Themen liegen dabei oft jenseits der relationalen Welt – NoSQL lautet das Zauberwort. Neben Performanz und Skalierbarkeit ist ein wichtiger Aspekt die höhere Flexibilität bei der Modellierung von Datenstrukturen. Das Framework eXtended Objects zeigt unter Beteiligung der Graphdatenbank Neo4j auf, wie sich beim Mapping persistenter Objekte interessante neue Möglichkeiten jenseits des relational geprägten Java Persistence API (JPA) ergeben können.

von Dirk Mahler

In den letzten Jahren gab es im Bereich der Persistenz eine erfreuliche Tendenz zu verzeichnen, welche die Auswahl einer oder mehrerer (!) Datenbanken für ein Projekt betrifft: weniger relationale Gewohnheit, dafür mehr Orientierung an der Lösung bestehender fachlicher oder technischer Probleme. Die Rede ist von der Bereitschaft zum Einsatz von Lösungen wie Cassandra, MongoDB und Co., wenn von diesen bestehende Anforderungen an Verteilung, Ausfallsicherheit oder schlicht Flexibilität bei der Datenmodellierung besser erfüllt werden. Allerdings sieht sich der Anwender mit einer ihm ungewohnten Situation konfrontiert: Während im Bereich relationaler Datenbanken das Java Persistence API mit seinen gereiften Vertretern wie Hibernate oder EclipseLink für das Mapping auf Java-Objekte zur Verfügung steht, erinnert das Programmiermodell für die Kommunikation mit NoSQL-Datenbanken sehr oft an längst überwunden geglaubte JDBC-Zeiten. Mit Spring Data [1] und seinem Repository-Ansatz steht zwar eine sehr interessante Lösung zur Verfügung, jedoch erfolgt auch hier das Mapping von Entitäten analog zu JPA auf der Basis von POJOs, also „einfachen" Java-Klassen mit Feldern und entsprechenden Zugriffsmethoden – verbunden mit Einschränkungen in der Flexibilität der Modellierung.

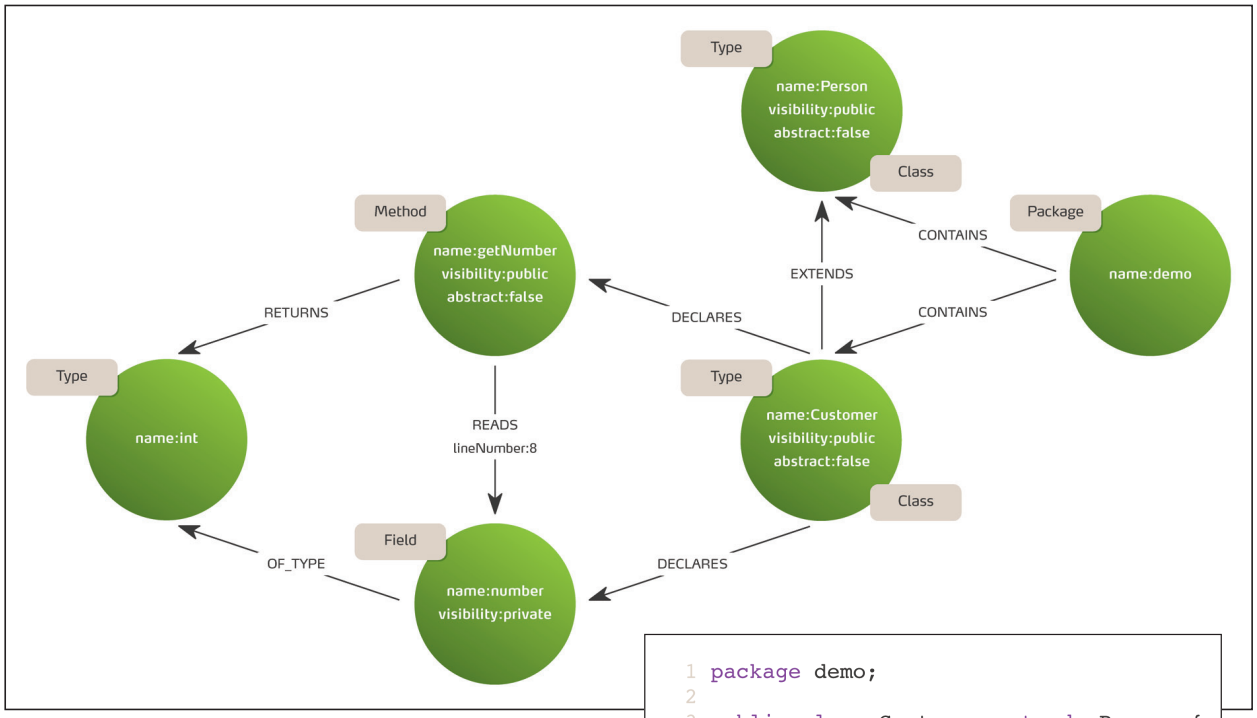
Das POJO-Problem

Bei der Entwicklung des Open-Source-Projekts „jQAssistant" [2], das es sich zur Aufgabe macht, Softwarestrukturen auf die Einhaltung von definierten Konventionen und Beschränkungen zu überprüfen (z. B. Namensregeln, Abhängigkeitsbeziehungen), besteht eine grundlegende Funktionalität darin, die Strukturen einer Java-Anwendung (also Packages, Klassen, Felder, Methoden etc.) einzulesen und als entsprechende Datenstrukturen in Neo4j [3] abzulegen. Beim ersten Entwurf der Modellierung dieser Domäne unter Zuhilfenahme von POJOs traten jedoch folgende Unannehmlichkeiten zutage:

1. Hohe Redundanz bzw. Boilerplate-Code: Deklaration der Felder und zugehöriger Zugriffsmethoden, verbunden mit sich stets wiederholenden Typdeklarationen und der Implementierung trivialer Zugriffslogik (kursive Hervorhebungen in Listing 1).
2. Fehlende Mehrfachvererbung: Diese ist zwar über Interfaces realisierbar, deren Methoden müssen jedoch immer wieder ausimplementiert werden und verschärfen damit das erste Problem deutlich.

Listing 1: Redundanter Code in POJOs

```
@Entity // JPA
public class Type {
  private String name;
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```




```
1  package demo;
2
3  public class Customer extends Person {
4
5    private int number;
6
7    public int getNumber() {
8        return this.number;
9    }
10
11 }
```

Abb. 1: Codebeispiel (rechts) und passende grafische Darstellung (oben)

Insbesondere die unzureichende Mehrfachvererbung bereitete Kopfzerbrechen. In Tabelle 1 wird anhand von Beispielen verdeutlicht, dass sich eine Menge gleichartiger Eigenschaften (*name*, *visibility*, *abstract*) an verschiedenen Entitäten (*Package*, *Type*, *Field*, *Method*) in unterschiedlichen Kombinationen wiederfinden und es daher nicht oder nur schwer möglich ist, sinnvolle Vererbungshierarchien aufzubauen. Es wäre also wünschenswert, einmal definierte Eigenschaften einfach wiederverwenden zu können.

Neo4j, Labels und Interfaces

In dieser Aussage versteckt sich bereits der Lösungsansatz, der durch das Open-Source-Framework „eXtended Objects“ [4] realisiert wird: Die Eigenschaften eines persistenten Objekts werden nicht mehr in Klassen *implementiert*, sondern lediglich als Zugriffsmethoden in Interfaces *deklariert*. Instanzen werden grundsätzlich über das Framework unter Zuhilfenahme dynamischer Proxys erzeugt und verwaltet. Diese Idee eröffnet neben erheblicher Codereduktion und Mehrfachvererbung weitere interessante Möglichkeiten, auf die im Folgenden eingegangen werden soll.

Zunächst muss aber noch ein Blick auf das Datenmodell von Neo4j geworfen werden, da sich hier ein interessanter Zusammenhang offenbart. In aller Kürze beschrieben werden Daten als Graph abgelegt, dessen grundlegende Elemente *Knoten* (z. B. für Typen, Methoden) und getypte *Beziehungen* (z. B. DECLARES, INVOKES) sind. An diesen beiden Grundelementen können *Eigenschaften* abgelegt werden (z. B. *name*, *lineNumber*). In **Abbildung 1** wird ein Beispiel im wahrsten Sinn des Wortes „grafisch“ veranschaulicht. An den hier dargestellten Knoten kommt ein weiteres wichtiges Element zum Tragen: so genannte *Labels*. Einem von Natur aus typlosen Knoten können mit ihrer Hilfe eine oder mehrere Rollen zugewiesen werden – im konkreten Fall des Knotens mit dem Namen *Customer* sind dies *Type* und *Class*. Näher betrachtet impliziert die Präsenz eines Labels auch das Vorhandensein spezifischer Eigenschaften (Class → visibility) oder Beziehungen (Type → DECLARES).

Übersetzt in die Java-Welt und im Sinne eines Mappings repräsentiert ein Knoten eine „persistente“ Instanz. Er kann über beliebig viele Labels mit entsprechenden Eigenschaften und Beziehungen verfügen – die Instanz also eine entsprechend dynamische Anzahl von Java-Typen abbilden. Die Verwendung von Interfaces für diese Typen ist also naheliegend, da sie – ohne die Voraussetzung einer Vererbungshierarchie – ebenfalls

	name	visibility	abstract
Package	X		
Type	X	X	X
Field	X	X	
Method	X	X	X

Tabelle 1: Eigenschaften von Java-Elementen

Listing 2: Mapping eines Interface auf einem Neo4j-Label

```
@Label
public interface Type {
  String getName();
  void setName(String name);
}
```

Listing 3: Einbindung des Neo4j-Datastores (Maven)

```
<dependency>
  <groupId>com.buschmais.xo</groupId>
  <artifactId>xo.neo4j</artifactId>
  <version>0.4.0</version>
</dependency>
```

Listing 4: XO-Descriptor: META-INF/xo.xml

```
<xo:xo version="1.0" xmlns:xo ="http://buschmais.com/xo/schema/v1.0">
  <xo-unit name="demo">
    <url>file:///C:/demo/db</url> // use an embedded Neo4j instance
    <provider>com.buschmais.xo.neo4j.api.Neo4jXOProvider</provider>
    <types>
      <type>com.buschmais.jqassistant.Type</type>
    </types>
  </xo-unit>
</xo:xo>
```

Listing 5: eXtended-Objects-API

```
XOManagerFactory xmf = XO.getXOManagerFactory("demo"); // as declared in xo.xml
XOManager xm = xmf.createXOManager();
xm.currentTransaction().begin();
Type type = xm.create(Type.class); // Creates and persists an instance
type.setName("Customer");
xm.currentTransaction().commit();
xm.close();
xmf.close();
```

Listing 6: Template

```
public interface NameTempl { // a template interface declaring name accessors
  String getName();
  void setName(String name);
}

@Label
public interface Type extends NameTempl, AbstractTempl, VisibilityTempl {
}

@Label
public interface Field extends NameTempl, VisibilityTempl {
}
```

beliebig miteinander kombinierbar sind. Ein Blick über den Gartenzaun zum relationalen Nachbarn zeigt, dass er dieses Dynamikproblem nur bedingt kennt – der starren 1:1-Abbildung von Java-Typen auf Tabellen in festen Schemata sei Dank. Im Umkehrschluss stellt sich aber die Frage, ob die Modellierung von Datenstrukturen mittels POJOs nicht grundsätzlich für „schemafreie" Datenbanken eher eine Einschränkung darstellt (z. B. für MongoDB-Dokumente).

In Listing 2 ist ein auf einem Interface basierendes Mapping veranschaulicht. Im Vergleich zu Listing 1 hat sich bei vergleichbarem Informationsgehalt der Umfang des notwendigen Codes halbiert.

Das Erzeugen einer Instanz dieses Interface zieht die Erzeugung eines Knotens mit dem Label *Type* in der Graphdatenbank nach sich. Die Annotation *@Label* wird durch den Neo4j-Datastore von eXtended Objects (XO) bereitgestellt; dessen Einbindung erfolgt über die in Listing 3 angegebene Maven-Abhängigkeit. Das notwendige Bootstrapping und die Verwendung des API von eXtendend Objects ist in Listing 4 und 5 skizziert – beides orientiert sich an Konzepten, wie sie bereits von JPA bekannt sind. Grundlage sind benannte XO-Units, die als Mindestanforderung neben einem Datenbank-URL den entsprechenden Provider sowie eine Menge von Entitäts- und Relationstypen deklarieren.

Templates, Komposition und Migration

Das in Tabelle 1 beschriebene Problem gleichartiger Eigenschaften an verschiedenen persistenten Typen lässt sich nun elegant lösen, indem jeweils zusammengehörige Eigenschaften in Templateinterfaces ohne Labelannotation ausgelagert und mittels (Mehrfach-)Vererbung einfach wiederverwendet werden. Dieses Vorgehen ist in Listing 6 skizziert.

Vererbungsbeziehungen können natürlich auch zwischen Interfaces bestehen, die mit *@Label* annotiert sind. In der fachlichen Domäne von jQAssistant ist dies z. B. zur Repräsentation von Klassen der Fall – es werden an einem Knoten die Labels *Type* und *Class* benötigt, dementsprechend leitet das Interface *Class* vom Interface *Type* ab (Listing 7). eXtended Objects bezeichnet diesen Fall als *statische Komposition*, da die Zusammengehörigkeit der Labels bereits zum Compile-Zeitpunkt festgelegt wird.

Die Nutzung von Interfaces schafft aber auch die Möglichkeit einer *dynamischen Komposition*, d. h. die Festlegung der zu verwendenden Labels bzw. Typen zur Laufzeit, so wie es Listing 8 skizziert. Die Typen *Class* und *File* stehen in keinerlei Vererbungsbeziehung zueinander. Es kann jedoch ein Knoten erzeugt werden, der beide Labels trägt. Damit muss auch die ihn repräsentierende Instanz zu beiden Typen zuweisungskompatibel sein. Durch das API von eXtended Objects wird in diesen Fällen ein *CompositeObject* zurückgegeben, das mittels *as()*-Methode in den jeweiligen Rollen verwendet werden kann.

Der Dynamikaspekt lässt sich noch weiter ausdehnen: Es ist denkbar, dass zur Laufzeit für einen bereits erzeugten Knoten die Menge seiner Labels bzw. Typen geändert, d. h. migriert werden müssen. Listing 9 zeigt den entsprechenden Methodenaufruf am Interface des XOManagers. Die bestehende Instanz (*type*) wird dabei invalidiert, und es kann anschließend nur noch mit *classType* weitergearbeitet werden. Es handelt sich im Hintergrund aber nach wie vor um denselben Knoten in der Datenbank, der lediglich um ein weiteres Label ergänzt wurde.

Derartige Anwendungsfälle entsprechen den natürlichen Modellierungsmöglichkeiten von Neo4j, sind mit existierenden Mapping-Ansätzen auf POJO-Basis aber nicht ohne Weiteres abbildbar.

Beziehungen

Es gibt wohl keine fachliche Domäne, in der nicht zwischen Entitäten irgendeine Form von Beziehung existiert. Eine gute Unterstützung hierfür ist das Kernmerkmal von Graphdatenbanken wie Neo4j. Dem sollte natürlich auch das verwendete Mappingframework Rechnung tragen. Der einfachste Fall hierfür ist die transparente Behandlung einer Beziehung, wenn eine Eigenschaft (d. h. ein Getter/Setter-Paar) deklariert wird, deren Typ einer Entität oder einer Collection derselben entspricht. So würde beispielsweise der Aufruf einer Methode *void setExtends(Class superClass)* eine Beziehung *EXTENDS* zwischen zwei Knoten erzeugen. Interessanter sind jedoch zwei andere Fälle, die näher betrachtet werden sollen.

Beim ersten handelt es sich um die Abbildung bidirektionaler Beziehungen: Diese werden von eXtended Objects im Gegensatz zu JPA grundsätzlich explizit als eigenständige Java-Typen behandelt. Der Vorteil besteht darin, dass eine größere Sicherheit gegenüber Refactorings gewährleistet werden kann und alle Mappinginformationen, welche die Beziehung betreffen, direkt daran spezifiziert werden können.

Für einfache bidirektionale Beziehungen ohne Eigenschaften kommen hierfür mit *@Relation* annotierte Annotationstypen zum Einsatz, die in Anlehnung an CDI als Qualifier bezeichnet werden (Listing 10). Die Zugriffsmethoden in den Entitäten werden mit dem entsprechenden Qualifier und der gewünschten Richtung – also *@Outgoing* oder *@Incoming* – versehen. Zur Sicherung der Konsistenz aktualisiert eXtendend Objects transparent beim Aufruf eines Setters der einen Seite den entsprechenden Wert der Gegenseite.

Der zweite zu betrachtende Fall sind Beziehungen, die selbst über Eigenschaften verfügen. Im Datenmodell von jQAssistant betrifft dies u. a. READS zwischen einer Methode und einem Feld: Die Eigenschaft *lineNumber* gibt an, in welcher Zeilennummer der Methode der Lesezugriff erfolgt. In diesem Fall wird der Beziehungstyp – wenig überraschend – auf ein Interface abgebildet, das wiederum die benötigten Eigenschaften sowie Getter mit den gewünschten Richtungen auf die umgebenden Entitäten (*Method* bzw. *Field*) deklariert (Listing 11). Der Lebenszyklus einer solchen Beziehung wird über den XOManager verwaltet. Es können hier selbstverständlich auch Eigenschaften aus Templatetypen verwendet werden, eine Vererbung zwischen Beziehungstypen ist jedoch nicht möglich.

Listing 7: Statische Komposition

```
@Label
public interface Class extends Type {
  // an instance will be labeled with Type and Class
}
```

Listing 8: Dynamische Komposition

```
@Label
public interface File {
  String getFileName();
  void setFileName();
}

CompositeObject type = xoManager.create(Type.class, File.class);
 // type.getClass().getInterfaces() reports CompositeObject, Type and File.
type.as(Type.class).setName("Customer");
type.as(File.class).setFileName("/com/buschmais/demo/Customer.class");
```

Listing 9: Migration

```
CompositeObject type = xoManager.create(Type.class, File.class);
// class.getClass().getInterfaces(): CompositeObject, Type and File
CompositeObject classType = xoManager.migrate(type, Class.class, File.class);
// classType.getClass().getInterfaces(): CompositeObject, Type, Class and File
```

Listing 10: Qualifizierte Beziehung

```
@Relation("DECLARES") // Defines a qualifier for relations
// DECLARES is optional and defines the name to be used in the database
@Retention(Runtime) @Target(Method)
public @interface Declaration {}

@Label
public interface Type {
  @Declaration
  @Outgoing List<Method> getDeclaredMethods()
}

@Label
public interface Method {
  @Declaration
  @Incoming Type getDeclaringType()
}
```

Abfragen

Abfragen stellen einen weiteren Aspekt dar, der durch ein Mappingframework unterstützt werden muss. JPA bietet hierfür eine eigene Sprache an: JP-QL. Eine ihrer Aufgaben ist es, eine Brücke zwischen relationalen Fremdschlüsseln zu objektorientierten Navigationen zu bauen, sie bleibt in ihrer Syntax dabei relativ nah an SQL. Es könnte der Versuch unternommen werden, ein entsprechendes Äquivalent für eXtended Objects zu schaffen, allerdings müsste dann die Sinnfrage gestellt werden: Das Neo4j zugrunde liegende Datenmodell weist bereits eine sehr große Nähe zu objektorientierten Strukturen auf und Cypher ist als Abfragesprache bereits sehr ausdrucksstark. Da jede Übersetzung und Abstraktion einen Verlust an Flexibilität bedeutet, liegt es nahe, die native Abfragesprache der Datenbank direkt zu nutzen (Listing 12) und dafür anderen Aspekten die ihnen gebührende Aufmerksamkeit zu schenken.

Ein interessantes Problem entsteht beispielsweise, wenn das Ergebnis mehr als eine Spalte in der Return-Klausel liefert. Daher erlaubt eXtended Objects die Verwendung von Interfaces zur Projektion von Ergebnissen – eine durch die Abfrage gelieferte Spalte entspricht dabei einer deklarierten *Get*-Methode. Das Framework geht allerdings noch einen Schritt weiter: Der Projektionstyp kann mit einer Annotation *@Cypher* versehen werden, die als Parameter einen Query-String entgegennimmt. Es entsteht eine getypte Abfrage, die dem Abfrage-API übergeben werden kann und damit ihr Ergebnis selbst beschreibt (Listing 13). Der Ansatz hat Ähnlichkeiten zu Named Queries von JPA, umgeht aber das Problem, dass Abfragen in vielen Fällen nicht eindeutig einer Entität zugeordnet werden können und schafft wiederum mehr Sicherheit bei Refactorings.

Dynamische Eigenschaften

In den vorangegangen Beispielen wurde bei der Definition eines persistenten Typs (also Entität oder Relation) immer davon ausgegangen, dass deklarierte Methoden entweder eine Rolle als Getter oder Setter für Eigenschaften bzw. Beziehungen einnehmen. Das deklarative Vorgehen kann aber genauso gut auch die Ausführung von nutzerdefinierten Abfragen bei Aufruf der einer Methode ermöglichen. Dazu dient die Annotation *@ResultOf*, ihre Funktionsweise ist in Listing 14 demonstriert. Bei Aufruf von *getNumberOfMethodsByName* wird die durch die Annotation *@Cypher* definierte Abfrage ausgeführt. Dabei werden der Wert des Methodenparameters *n* unter dem angegebenen Namen *name* explizit und *this* als Identität der Instanz implizit als Parameter übergeben. Der Rückgabewert entspricht dem Ergebnis der Abfrage. Ähnlichkeiten zu Ansätzen aus Frameworks

Listing 11: Getypte Beziehung

```
@Relation // no explicit name provided thus the name of the type is used
public interface Reads {
  int getLineNumber();
  void setLineNumber(int lineNumber)

  // managed by eXtended Objects
  @Outgoing Method getMethod();
  @Incoming Field getField();
}

@Label
public interface Method {
  List<Reads> getReads()
}

@Label
public interface Field {
  List<Reads> getReads()
}

Reads reads = xoManager.create(method, Reads.class, field);
reads.setLineNumber(8);
...
xoManager.delete(reads);
```

Listing 12: Abfragen

```
QueryResult<Type> result = xm.createQuery("match (type:Type) return
                                type", Type.class).execute(); // Return all types
for (Type type : result) { ... }
types.close(); // The result is a transactional iterator and must be closed
```

Listing 13: Getypte Abfragen

```
@Cypher("match (type:Type)-[:DECLARES]->(method:Method)
  return type, count(method) as numberOfMethods")
// returns two columns: type and numberOfMethods
public interface TypeWithNumberOfMethods {
  Type getType(); // maps to column type
  long getNumberOfMethods(); // maps to column numberOfMethods
}

QueryResult<TypeWithNumberOfMethods> result = xm.createQuery(
                                TypeWithNumberOfMethods.class).execute();
for (TypeWithNumberOfMethods row : result) {
  Type type = row.getType();
  long numberOfMethods = row.getNumberOfMethods();
}
result.close();
```

Listing 14: Dynamische Eigenschaften

```
public interface Type {
  @ResultOf
  @Cypher("match (type:Type)-[:DECLARES]->(method:Method) where
        id(type)={this} and method.name={name} return count(method)")
  long getNumberOfMethodsByName(@Parameter("name") String n);
}
```

wie Ruby on Rails sind dabei nicht unbeabsichtigt, wenngleich die Verwendung dieser Möglichkeit sicherlich in die Kategorie „Geschmackssache“ fällt.

Zusammenfassung und Ausblick

Am Anfang stand ein aus der Not heraus geborenes Experiment: die Schaffung einer Mappinglösung, die durch den Einsatz von Interfaces zur Abbildung von Datenstrukturen in Neo4j den Einsatz von Mehrfachvererbung ermöglichen sollte. Dieser kleine gedankliche Schritt eröffnete eine ungemeine Flexibilität und Leichtigkeit bei der Modellierung persistenter Typen sowie einige weitere interessante Möglichkeiten, die es nahelegten, daraus ein eigenständiges Framework zu entwickeln und es als Open-Source-Projekt zu veröffentlichen. eXtended Objects ist unter der Apache Software License 2.0 lizensiert und liegt momentan in der Version 0.4.0 vor. Es hat durch den Einsatz u. a. in jQAssistant bereits einen erheblichen Grad an Stabilität erreicht – Ausprobieren lohnt sich also, auch wenn die Projektdokumentation an einigen Stellen noch etwas lückenhaft ist.

Die im Vorangegangenen beschriebenen Funktionalitäten – von Templates, über statische oder dynamische Komposition, Migration, typsichere Beziehungen bis hin zu dynamischen Eigenschaften – lassen sich sehr gut auf andere Datenbanken übertragen. So existieren bereits erste Communityanbindungen für OrientDB [5] oder Titan [6]. Konzeptionell können neben Graphdatenbanken auch Vertreter anderer Familien (z. B. Column bzw. Document Stores) von derartigen Konzepten profitieren – entsprechende Implementierungen stehen daher auf der mittelfristigen Roadmap des Projekts. Die Zeiten JDBC-ähnlicher Programmiermodelle sollten also auch im Umfeld von NoSQL zukünftig der Vergangenheit angehören.

Dirk Mahler ist Senior-Consultant bei der buschmais GbR, einem Beratungshaus mit Sitz in Dresden. Der Schwerpunkt seiner mittlerweile mehr als zehnjährigen Tätigkeit liegt im Bereich Architektur für Java-Applikationen im Unternehmensumfeld. Den Fokus setzt er dabei auf die Umsetzung von Lösungen, die im Spannungsfeld zwischen Pragmatismus, Innovation und Nachhaltigkeit liegen. In diesem Rahmen engagiert er sich für die Open-Source-Projekte jQAssistant und eXtended Objects.

Links & Literatur

[1] http://jqassistant.org

[2] http://projects.spring.io/spring-data/

[3] http://neo4j.org

[4] http://github.com/buschmais/extended-objects

[5] http://github.com/BluWings/xo-orientdb

[6] http://github.com/PureSolTechnologies/extended-objects-titan

JAVA³

Jetzt abonnieren!
www.javamagazin.de

Jetzt 3 TOP-VORTEILE sichern!

1

- Alle Printausgaben frei Haus erhalten
- Intellibook-ID kostenlos anfordern (www.intellibook.de)

2 Abo-Nr. (aus Rechnung oder Auftragsbestätigung) eingeben

3 Zugriff auf das komplette PDF-Archiv mit der Intellibook-ID

S&S Media Group

www.javamagazin.de