

Wie die Datenbank auch Scala spricht

Der MongoDB-Drive

MongoDB ist erwachsen geworden. Immer mehr Unternehmen setzen MongoDB in ihren geschäftskritischen Anwendungen ein. Eine aktive Community sorgt dafür, dass die Integration in zahlreiche Programmiersprachen immer effizienter wird. Wie MongoDB in Java und Scala angesprochen werden kann, erfahren Sie in diesem Artikel.

von Maximilian Weber

Im ersten Teil dieser Artikelserie wurde anhand eines Beispiels gezeigt, wie mithilfe der interaktiven MongoDB-Konsole JSON-Dokumente in einer MongoDB-Datenbank verwaltet werden können. Als Domäne für dieses Beispiel wurde ein Webshop gewählt, für dessen Kunden jeweils der Name, eine Kundennummer, der Wohnort,

die Bankverbindung und die Anzahl der bisherigen Bestellungen hinterlegt sind. Dieser Artikel behandelt die Integration von MongoDB in Java und verdeutlicht, wie mithilfe des MongoDB-Java-Treibers [1] das oben genannte Beispiel umgesetzt werden kann. Ferner wird anhand desselben Beispiels eine Scala-spezifische Erweiterung dieses Treibers vorgestellt, der bei der Interaktion mit einer MongoDB die von Scala-Bibliotheken gewohnte Eleganz bietet. Dieser MongoDB-Scala-Treiber ist nur eines von vielen Communityprojekten rund um MongoDB, einige weitere für die Java-Plattform findet

Artikelserie

Teil 1: Was ist MongoDB?

Teil 2: Java-MongoDB-Treiber

man unter [1]. Beide Implementierungen des Beispiels sollen die gleichen Ergebnisse produzieren, wie die Befehle auf der MongoDB-Konsole aus dem ersten Teil, die hier noch einmal in Listing 1 zu sehen sind.

Der MongoDB-Java-Treiber

In Listing 2 ist das Java-Äquivalent der Zeilen 1 bis 10 aus Listing 1 zu sehen. Dieses verwendet den MongoDB-Java-Treiber, um mit der MongoDB-Instanz zu interagie-

Listing 1

```

1. use webshop
2. > db.customer.save({id: 4711,
name: "Max Mustermann",
city: "Cologne", numberOfOrders: 3});
3. > db.customer.findOne();
4. {
5.   "_id" : ObjectId("4b9945fe1c23e16a3c0a77c1"),
6.   "id" : 4711,
7.   "name" : "Max Mustermann",
8.   "city" : "Cologne",
9.   "numberOfOrders" : 3
10. }
11. > db.customer.save({"id" : 1234, "name" : "Otto Normal", city: "Berlin", numberOfOrders:
4, bankData: {accountNumber : "9876543210", bankCode : "30020011", accountHolder : "Otto Normal" }});
12. > db.customer.find({city: "Berlin"});
13. { "_id" : ObjectId("4b9b7bc4d9433f42225bffb6"), "id" : 1234, "name" : "Otto Normal", "city" :
"Berlin", "numberOfOrders" : 4, "bankData" : { "accountNumber" : "9876543210",
"bankCode" : "30020011", "accountHolder" : "Otto Normal" } }
14. > db.customer.ensureIndex({id : 1}, {unique: true})
15. > db.customer.update({id: 1234}, { $inc : { numberOfOrders : 1 } });
16. > db.customer.find({ numberOfOrders : { $gt: 4 } });
17. { "_id" : ObjectId("4b9947b01c23e16a3c0a77c2"), "id" : 1234, "name" : "Otto Normal",
"city" : "Berlin", "numberOfOrders" : 5 }

```

1/3 oose

Listing 2

```

1. public class WebshopTest {
2.   private DBCollection customers;
3.   @Before public void setUp() throws Exception {
4.     Mongo mongo = new Mongo();
5.     DB db = mongo.getDB("webshop");
6.     customers = db.getCollection("customer");
7.     customers.remove(new BasicDBObject());
8.   }
9.   @Test public void shouldWorkWithMongoDB() {
10.    DBObject customerMaxMustermann = new BasicDBObject()
11.      .append("id", "4711")
12.      .append("name", "Max Mustermann")
13.      .append("city", "Cologne")
14.      .append("numberOfOrders", 3);
15.    customers.insert(customerMaxMustermann);
16.    DBObject savedCustomer = customers.findOne();
17.    assertNotNull(savedCustomer);
18.    assertEquals("4711", savedCustomer.get("id"));
19.  } // ...
20. }

```

ren. Zur besseren Übersicht wurde das Beispiel als JUnit-Test formuliert und in einzelne Testfälle unterteilt. Des Weiteren verdeutlichen die Assert-Anweisungen, welche Ergebnisse durch die Aktionen produziert werden.

Listing 3

```

1. public class Customer extends ReflectionDBObject
2. {
3.     private String id;
4.     private String name;
5.     private String city;
6.     private int numberOfOrders;
7.     private BankData bankData;
8.     /* Getter and setter skipped */
9.     //...
10. }
11. public class WebshopTest {
12.     //...
13.     @Test
14.     public void shouldSaveAndFindCustomers() {
15.         Customer customer = new Customer();
16.         customer.setId("1234");
17.         customer.setName("Otto Normal");
18.         customer.setCity("Berlin");
19.         customer.setNumberOfOrders(4);
20.         BankData bankData = new BankData();
21.         bankData.setAccountNumber("9876543210");
22.         bankData.setBankCode("30020011");
23.         bankData.setAccountHolder("Otto Normal");
24.         customer.setBankData(bankData);
25.
26.         customers.setObjectClass(Customer.class);
27.         customers.insert(customer);
28.
29.         DBCursor result = customers.find(new BasicDBObject("City", "Berlin"));
30.         assertTrue(result.hasNext());
31.         Customer savedCustomer = (Customer) result.next();
32.         assertEquals("1234", savedCustomer.getId());
33.         assertEquals("30020011", savedCustomer.getBankData().getBankCode());
34.     }
35.
36.     @Test
37.     public void shouldIncrementTheNumberOfOrdersIfTheCustomerPlaceAnOrder() {
38.         // to save the customer Otto Normal
39.         shouldSaveAndFindCustomers();
40.
41.         customers.update(new BasicDBObject("Id", "1234"),
42.             new BasicDBObject("$inc",
43.                 new BasicDBObject("NumberOfOrders", 1)));
44.         Customer savedCustomer = (Customer) customers
45.             .findOne(new BasicDBObject("NumberOfOrders", new
46.                 BasicDBObject("$gt", 4)));
47.         assertNotNull(savedCustomer);
48.         assertEquals("Otto Normal", savedCustomer.getName());
49.         assertEquals(5, savedCustomer.getNumberOfOrders());
50.     }

```

Beim Aufbau des Test-Fixtures in der *setUp*-Methode wird ein *Mongo*-Objekt instanziiert. Dieses verwaltet die Verbindungen zu einer MongoDB-Instanz und besitzt bereits ein eingebautes Connection Pooling. Werden keine Konstruktorparameter angegeben, so wird der Versuch unternommen, eine Verbindung zur lokalen MongoDB-Instanz auf dem Standard-Port herzustellen. MongoDB setzt im Normalfall eine sichere Umgebung voraus und verlangt keine Authentifizierung. Diese kann allerdings aktiviert werden [2], sodass die Angabe eines Benutzernamens und Kennworts verlangt wird.

In Zeile 5 wird ein *DB*-Objekt erzeugt, um den Zugriff auf die Webshopdatenbank zu ermöglichen. Dieses wird genutzt, um ein *DBCollection*-Objekt für die MongoDB-Collection *customer* zu erhalten. Die abstrakte Klasse *DBCollection* ist eine der zentralen Klassen des MongoDB-Java-Treibers und bietet die wesentlichen Methoden, um (BSON-)Dokumente innerhalb einer Collection zu verwalten. Ein weiteres wichtiges Puzzlestück ist das *DBObject*-Interface. Dieses besitzt ähnliche Methoden wie eine *java.util.Map* und repräsentiert jeweils ein (BSON-)Dokument. In Zeile 7 wird mit der Standardimplementierung des *DBObject*-Interface, der *BasicDBObject*-Klasse, ein leeres Dokument erzeugt. Der Aufruf der *remove*-Methode auf der *DBCollection* *customers* mit einem leeren Dokument als Filterparameter bewirkt, dass alle Dokumente aus der Collection *customer* gelöscht werden. Somit startet der Test mit einer definierten Testumgebung. Hier gelten dieselben Regeln wie bei der MongoDB-Konsole: Existiert die Datenbank und/oder die Collection noch nicht, dann wird diese durch MongoDB automatisch angelegt.

Im ersten Testfall wird der Datensatz für den Webshopkunden Max Mustermann angelegt, und mithilfe der Standardimplementierung *BasicDBObject* wird hierfür ein *DBObject* erzeugt. Durch das Fluent-Interface [3] der *append*-Methode können dem Dokument die Eigenschaften in einem Aufruf hinzugefügt werden. In Zeile 15 wird der Kundendatensatz in die MongoDB-Collection eingefügt. Da beim Aufbau des Test-Fixtures alle Dokumente aus der Collection *customer* entfernt wurden, liefert der Aufruf der Methode *findOne* in Zeile 16 das soeben in die Collection eingefügte Dokument zurück.

Bean – MongoDB – Bean

Listing 3 zeigt zwei weitere Testfälle, die notwendig sind, um mit dem Beispiel auf der MongoDB-Konsole gleichziehen zu können.

Zunächst wird eine Klasse *Customer* definiert, die einen Webshopkunden repräsentieren soll. Die Bankverbindungsdaten wurden als separate Klasse *BankData* modelliert, die sich als Komposition in der *Customer*-Klasse wiederfindet. Die Klasse *BankData* ist ebenfalls eine einfache Java Bean und wurde deshalb nicht mehr abgedruckt. Beide Klassen erben von *ReflectionDBObject*. Dieses wiederum implementiert das *DBObject*-Interface und kann somit in eine MongoDB gespeichert werden. Wie der Name bereits vermuten lässt, arbeiten

die Entwickler des MongoDB-Java-Treibers hierbei mit der Java-Reflection-API, um die Eigenschaften der Bean auszulesen bzw. zu setzen. Die Bankdaten (*BankData*) werden als Teil des *Customer*-Dokuments in der MongoDB-Collection gespeichert.

In den Zeilen 15 bis 24 wird das *Customer*-Objekt und das dafür vorgesehene *BankData*-Objekt mit den Daten des Kunden *Otto Normal* befüllt. Anschließend wird dieses in die *DBCollection customers* eingefügt. Dabei ist zu beachten, dass vorher der MongoDB-Java-Treiber die Information erhielt, welche Klasse von Objekten die *DBCollection* verwalten soll (Zeile 26). In Zeile 29 wird nach allen Dokumenten in der MongoDB-Collection *customer* gesucht, die eine Eigenschaft {„City“ : „Berlin“} haben. Als Ergebnis des Aufrufs erhält man einen *DBCursor*, der u. a. *java.util.Iterator* implementiert und dadurch ermöglicht, über die Ergebnismenge zu iterieren. Da der Kunde *Otto Normal* in eine leere *customer* Collection eingefügt und als Wohnort „Berlin“ eingegeben wurde, wird dieser auch als erster Datensatz in der Ergebnismenge erwartet. Dies wird in den Zeilen 30 bis 33 überprüft.

Atomare Operationen

Im letzten Testfall (Zeile 37) wird erneut der Test *shouldSaveAndFindCustomers* aufgerufen, damit sich der Kunde *Otto Normal* in der MongoDB-Collection

customer befindet. Die Initialisierung und das Einfügen des *Customer*-Objekts sollte in einem echten Test auf jeden Fall beim Aufbau des Test-Fixtures (*setUp*-Methode) erfolgen. Die hier gezeigte, etwas kritische Vorgehensweise ist lediglich gewählt worden, um den Beispielcode an die Reihenfolge der textlichen Erklärungen anzupassen. Da sich der Datensatz für den Kunden *Otto Normal* nun in der Collection *customer* befindet, kann anschließend (Zeile 41 bis 43) wie im Beispiel auf der MongoDB-Konsole eine atomare Inkrement-Operation für die Eigenschaft *NumberOfOrders* erzeugt und ausgeführt werden. Auch hier kommt das *DBObject* (*BasicDBObject*) zum Einsatz, um im ersten Parameter die Suchkriterien (die Kundennummer) für die Updateoperation und im zweiten Parameter die eigentliche Inkrement-Updateoperation zu beschreiben. Wie am zweiten Parameter zu sehen ist, kann ein *DBObject* wiederum ein *DBObject* aufnehmen. So lassen sich geschachtelte Strukturen erzeugen, wie man sie von JSON-Dokumenten gewöhnt ist. Um zu überprüfen, ob die Inkrement-Operation ausgeführt worden ist, wird ein Kunde gesucht, der mehr als vier Bestellungen getätigt hat (Zeile 44 und 45). Durch die In-

1/3 microtool

Listing 4

```

1. package com.flowfact
2. import org.specs._
3. import com.osinka.mongodb.Preamble._
4. import runner.JUnitSuiteRunner
5. import org.junit.runner.RunWith
6. import com.osinka.mongodb.{MongoCollection,
                                 DBOBJECTCollection}
7. import com.osinka.mongodb.shape.ShapedCollection
8. import com.mongodb.{DBObject, DB, Mongo}
9.
10. @RunWith(classOf[JUnitSuiteRunner])
11. class WebshopSpec extends SpecificationWithJUnit {
12.   "The webshop application" should {
13.     val mongo = new Mongo()
14.     val db = mongo.getDB("webshop")
15.     val dbCollection = db.getCollection("customer")
16.
17.     doBefore {
18.       dbCollection.remove(Map())
19.     }
20.     val customerCollection: DBOBJECTCollection =
                                 dbCollection.asScala
21.
22.     "work with MongoDB" in {
23.       customerCollection << Map("id" -> "4711",
24.                                "name" -> "Max Mustermann",
25.                                "city" -> "Cologne",
26.                                "numberOfOrders" -> 3)
27.       customerCollection must exist {_.get("id") ==
                                         "4711"}
28.       // {DBObject : DBOBJECT => dbObject.get("id")
         // == "4711"}
29.     }
30.     //...
31.   }
32. }

```

krement-Operation hat der Kunde *Otto Normal* nun fünf Bestellungen aufgegeben und sollte deshalb von der Abfrage des *findOne*-Aufrufs erfasst werden. Dies wird durch die Assert-Anweisungen in den Zeilen 46 bis 48 überprüft.

MongoDB-Scala-Treiber

Insgesamt sind die Möglichkeiten des MongoDB-Java-Treibers recht umfangreich. Im Vergleich zu einer JPA-Implementierung wie Hibernate, wird man allerdings einige Dinge vermissen. Hier kann der MongoDB-Scala-Treiber [4] teilweise Abhilfe schaffen. Dieser ist ein dünner Scala-Aufsatz für den MongoDB-Java-Treiber und ermöglicht, mit der von Scala-Bibliotheken gewohnten Eleganz mit einer MongoDB-Datenbank zu interagieren. Im Folgenden wird gezeigt, wie das Webshopbeispiel mithilfe dieses Scala-Aufsatzes bzw. des MongoDB-Scala-Treibers umgesetzt werden kann. In Listing 4 ist das Scala-Äquivalent des Java-Beispiels aus Listing 2 zu sehen. Dieser Artikel bietet keine Einführung für die verwendeten

Scala-Sprachelemente. Grundlegende Scala-Kenntnisse sind für das Nachvollziehen der Beispiele von Vorteil, aber nicht zwingend notwendig.

Specs und BDD

Statt des JUnit-Frameworks wurde hier Specs [5] als Testframework verwendet. Specs bietet eine Scala-DSL, mit der Software mittels Behaviour-driven Development (BDD [6]) entwickelt werden kann. Dieses kombiniert wiederum die Ansätze des Test-driven Developments und des Domain-driven Designs.

Spezifikationen erben bei Specs von der abstrakten Oberklasse *org.specs.Specification* oder einer ihrer Subklassen. Das Äquivalent zu einem Testfall wird bei

Specs als *Example* bezeichnet. *Examples* befinden sich entweder direkt im Body einer Spezifikationsklasse oder sind noch einmal in unterschiedliche Abschnitte eingeordnet, die in den meisten Fällen den Namen des zu spezifizierenden Systems tragen. In Listing 4, Zeile 22 ist ein solches *Example* definiert, das wiederum in einen Abschnitt eingeordnet ist, der in Zeile 12 beginnt.

Die Möglichkeiten des MongoDB-Java-Treibers sind recht umfangreich.

Listing 5

```

1. "The webshop application" should {
2.   //...
3.   val customers: ShapedCollection[Customer] = dbCollection of Customer
4.   val customerOttoNormal = new Customer
5.   customerOttoNormal.id = "1234"
6.   customerOttoNormal.name = "Otto Normal"
7.   customerOttoNormal.city = "Berlin"
8.   customerOttoNormal.numberOrders = 4
9.
10.  val bankData = new BankData("9876543210", "30020011", "Otto Normal")
11.  customerOttoNormal.bankData = bankData
12.
13.  "find a customer by city" in {
14.    customers << customerOttoNormal
15.    val result = Customer where {Customer.city eq_? "Berlin"}
16.                                     take 1 in customers
17.    result must haveSize(1)
18.    val savedCustomer = result.toList(0)
19.    savedCustomer.id mustEqual "1234"
20.    savedCustomer.bankData.bankCode mustEqual "30020011"
21.  }
22. //...

```

Listing 6

```

1. object Customer extends ObjectShape[Customer] {
2.   lazy val id = Field.scalar("id", _id, (x: Customer, v: String) => x.id = v)
3.   lazy val name = Field.scalar("name", _name, (x: Customer, v: String) =>
4.                                     x.name = v)
5.   lazy val city = Field.scalar("city", _city, (x: Customer, v: String) => x.city = v)
6.   lazy val numberOrders = Field.scalar("numberOrders", _numberOrders,
7.                                       (x: Customer, v: Int) => x.numberOrders = v)
8.
9.   object bankData extends EmbeddedField[BankData]("bankData", _bankData,
10.  Some((x: Customer, v: BankData) => x.bankData = v)) with BankDataIn[Customer]
11.   override lazy val * = List(id, name, city, numberOrders, bankData)
12.   override def factory(dbo: DBObject) = Some(new Customer)
13. }
14.
15. trait BankDataIn[T] extends ObjectIn[BankData, T] {
16.   lazy val accountNumber = Field.scalar("accountNumber", _accountNumber)
17.   lazy val bankCode = Field.scalar("bankCode", _bankCode)
18.   lazy val accountHolder = Field.scalar("accountHolder", _accountHolder)
19.
20.   override lazy val * = accountNumber :: bankCode :: accountHolder :: Nil
21.   override def factory(dbo: DBObject) =
22.     for{accountNumber(n) <- Some(dbo)
23.         bankCode(c) <- Some(dbo)
24.         accountHolder(h) <- Some(dbo)}
25.       yield new BankData(n, c, h)

```

Die Spezifikation in Listing 4 erbt von *SpecificationWithJUnit*, einer Subklasse von *org.specs.Specification*. Diese ermöglicht das Ausführen der Spezifikation mit JUnit. Examples werden von Specs anders verarbeitet als Testfälle (Methoden mit *@Test*-Annotation) von JUnit. Dabei unterscheidet sich vor allem die Initialisierung des Test-Fixtures, so wird z. B. keine *setUp*- oder *tearDown*-Methode benötigt. Das Test-Fixture wird bei Specs direkt im Konstruktor der Spezifikationsklasse bzw. der Abschnitte aufgebaut (vgl. Scala-Konstrukturen). Specs legt die Test-Fixtures für alle in einer Spezifikationsklasse definierten Examples im Voraus an und nicht wie JUnit erst direkt vor der Ausführung eines Examples/Testfalls. Ist man allerdings darauf angewiesen, dass z. B. jedes Example einen definierten Datenbankzustand vorfindet, so ist es nötig, bei Specs auf die *doBefore*-Methode (evtl. auch *doAfter*-Methode) zurückzugreifen. Der in der *doBefore*-Methode enthaltene Code wird vor der Ausführung eines jeden Examples aufgerufen. Listing 4 macht sich dies zunutze und leert vor jeder Ausführung eines Examples die MongoDB-Collection *customer* (Zeile 17 bis 19), um eine einheitliche Testausgangssituation zu schaffen. Für eine Einführung zu Specs sei auf das Wiki des Projekts verwiesen [7].

Magie mit Implicit Conversions

Das Test-Fixture der Scala-Version unterscheidet sich bis auf die o. g. Besonderheit der *doBefore*-Methode nicht wesentlich von dem der Java-Variante. In Zeile 13 bis 15 wird auch ein *Mongo*-Objekt erzeugt und die *DBCollection customer* aus der Datenbank *webshop* referenziert. In Zeile 20 wird die *com.mongodb.DBCollection* mit einer *com.osinka.mongodb.DBObjectCollection* des MongoDB-Scala-Treibers umhüllt. Dadurch erhält man nützliche Zusatzmethoden zur Arbeitserleichterung. Zum besseren Verständnis wurde hier auf die Type Inference von Scala verzichtet und der resultierende Typ explizit angegeben. Normalerweise könnten die angesprochenen Zusatzmethoden auch direkt auf der *dbCollection* aufgerufen werden, da Scala diese durch eine Definition einer Implicit-Konvertierung [8] automatisch in eine *com.osinka.mongodb.DBObjectCollection* umwandeln kann. Die Implicit-Definitionen für den MongoDB-Scala-Treiber sind im Objekt *Preamble* zu finden und werden im Beispiel durch „*import com.osinka.mongodb.Preamble._*“ importiert. Eine weitere dort enthaltene Implicit-Definition ermöglicht die automatische Umwandlung einer normalen Scala Map in ein *DBObject*. Dies macht sich der Aufruf in Zeile 18 zunutze. Die Methode *remove* der *DBCollection* würde normalerweise ein *DBObject* als Parameter erwarten. Durch den Implicit-Mechanismus wird die Scala Map automatisch in ein passendes *DBObject* umgewandelt.

In Zeile 23 nutzt das Example die Zusatzmethode *<<* der *customerCollection*-Variablen, die aus der *DBObjectCollection* bzw. dem dortigen *com.osinka.*

mongodb.MongoCollection Trait stammt. Diese sorgt dafür, dass das übergebene *DBObject* (hier eine Scala Map) in die MongoDB-Collection eingefügt wird. In Zeile 27 wird überprüft, ob der Datensatz für den Kunden *Max Mustermann* auch korrekt in der MongoDB-Collection *customer* gespeichert worden ist. Dazu wird der Syntax-Sugar von Specs genutzt. Mit dem Aufruf von *exist* wird überprüft, ob die anonyme Funktion für irgendein Element der *customerCollection true* zurückliefert. Dies ist der Fall, wenn ein Dokument (*DBObject*) mit der Eigenschaft { *“id“ : “4711“* } in der *customerCollection* existiert. Da das zuvor für den Kunden *Max Mustermann* in die MongoDB-Collection eingefügte Dokument eine solche Eigenschaft besitzt, ist der Testfall bzw. das vom Example spezifizierte Verhalten erfüllt. Die anonyme Funktion in Zeile 27 verwendet die Kurzschreibweise, eine ausführlichere Variante der Funktion mit Typangaben zeigt der Kommentar in Zeile 28.

Type-safe Queries

Dieses beschriebene Beispiel aus Listing 4 hat aufgezeigt, wie der MongoDB-Scala-Aufsatz die Arbeit mit

Anzeige

1/4 flowfact

den rudimentären (Low-level-)Bestandteilen des MongoDB-Java-Treibers vereinfachen kann. Seine eigentliche Stärke zeigt der MongoDB-Scala-Aufsatz aber erst bei der Abbildung eines Objekts aus dem Domänenmodell auf ein (BSON-)Dokument. Listing 5 zeigt einen weiteren Ausschnitt der Webshopspezifikation. Er zeigt, wie mit dem MongoDB-Scala-Treiber gearbeitet werden kann, sobald die Domänenklassen auf die (BSON-)Dokumente gemappt sind.

In Zeile 3 entsteht aus der *DBCollection* eine *com.osinka.mongodb.shape.ShapedCollection*, die die wirklich interessanten Zusatzfunktionalitäten des MongoDB-Scala-Treibers freilegt. In den darauffolgenden Zeilen werden noch im Rahmen des Test-Fixtures die Daten des Kunden *Otto Normal* initialisiert, der dann durch die Anweisung in Zeile 14 in die MongoDB-Collection *customer* eingefügt wird. Im Hintergrund war der MongoDB-Scala-Treiber dafür verantwortlich, dass aus dem *Customer*-Objekt und dem darin enthaltenen *BankData*-Objekt ein *DBObject* wurde und dieses in der entsprechenden MongoDB-Collection gespeichert wurde. Zeile 15 zeigt die Eleganz des MongoDB-Scala-Treibers: Hier wird vollkommen typsicher (type-safe) nach allen Kunden in der MongoDB-Collection *customer* gesucht, die in Berlin wohnen. Dabei wird die Paging-Funktionalität genutzt, um nur das erste Element aus der Ergebnismenge zu nehmen („take 1“, mit Drop N können auch die ersten N Datensätze übersprungen werden). In den Zeilen 16 bis 19 wird zunächst überprüft, ob die Ergebnismenge wirklich nur ein Element enthält, während anschließend die korrekte Datenspeicherung für die Eigenschaften *id* und *bankData.bankCode* gecheckt wird.

Mapping ohne Annotations

Damit der MongoDB-Scala-Treiber die *Customer*-Klasse auf (BSON-)Dokumente abbildet, muss das

Mapping mithilfe eines (Scala-)Objekts definiert werden. Für die Klasse *Customer* ist ein entsprechendes Companion Object – ebenfalls mit dem Namen *Customer* – definiert worden, das beispielsweise in Zeile 15 (Listing 5) in Aktion zu sehen ist. Listing 6 zeigt die beiden Objekte, die das Mapping für die Klassen *Customer* und *BankData* definieren.

Im ersten Moment wirkt die DSL für die Definition des Mappings sehr komplex. Sobald der grundlegende

Aufbau jedoch verstanden wurde, ist die Definition des Mappings nur noch reine Fleißarbeit. Für jede Eigenschaft der Klasse *Customer* wird dem MongoDB-Scala-Treiber durch das *Customer*-Objekt mitgeteilt, wie und auf welche Eigenschaft des BSON-Dokuments (*DBObject*) diese abgebildet werden soll. Folgende Zeile ist für das Mapping der Eigenschaft *id* der Klasse *Customer* verantwortlich: *lazy val id = Field.scalar("id", _id, (x: Customer, v: String) => x.id = v)*. Das *Field*-Objekt des MongoDB-Scala-Treibers bietet Fabrikmethoden an, um die für das Mapping notwendigen Objektinstanzen zu erzeugen. Die Fabrikmethode *scalar* wird für elementare Datentypen verwendet und verlangt mindestens zwei Parameter. Der erste Parameter ist der Schlüssel der Eigenschaft des BSON-Dokuments, auf die die Eigenschaft der Klasse abgebildet werden soll. Im Beispiel wird die *id* der Klasse *Customer* auf eine BSON-Eigenschaft mit dem Schlüssel „id“ abgebildet. Als zweiten Parameter erwartet die *scalar*-Methode eine Funktion, die den Wert der Eigenschaft aus einer Instanz ausliest. Im Beispiel wird mit *_id* die Kurzschreibweise verwendet: *(c: Customer) => c.id* ist die ausführliche Variante. Diese gibt den Wert der *id*-Eigenschaft einer beliebigen *Customer*-Instanz zurück. Falls eine Eigenschaft einer Klasse veränderlich ist, wie hier die Eigenschaft *id* der Klasse *Customer*, kann mit dem dritten Parameter der *scalar*-Methode eine Funktion angegeben werden, die die *id*-Eigenschaft einer *Customer*-Instanz auf den übergebenen Wert setzt. Diese Funktion erwartet als ersten Parameter (*x*) die *Customer*-Instanz, deren *id*-Eigenschaft auf den Wert des zweiten Funktionsparameters (*v*) gesetzt werden soll.

Das Mapping der übrigen *Customer*-Eigenschaften mit elementaren Datentypen erfolgt nach demselben Muster. Damit der MongoDB-Scala-Treiber weiß, welche Eigenschaften der *Customer*-Klasse gemappt werden sollen, muss die Eigenschaft *** des Traits *ObjectIn* (*ObjectShape* erbt von diesem) mit einer Liste überschrieben werden, die diese Eigenschaften auflistet (Zeile 8). Zu guter Letzt muss dem MongoDB-Scala-Treiber noch mitgeteilt werden, wie Instanzen der Klasse *Customer* erzeugt werden. Zu diesem Zweck wird die *factory* Methode – ebenfalls aus dem *ObjectIn* Trait – über-

Zeile 15 zeigt die Eleganz des MongoDB-Scala-Treibers.

Listing 7

```

1. "increment the number of orders if the customer place an order" in {
2.   customers << customerOttoNormal
3.   customers.update(Customer.id eq_? "1234",
4.                     Customer.numberOfOrders inc 1)
5.   val foundCustomers = Customer where {Customer.numberOfOrders
6.     is_> 4} take 1 in customers
7.   foundCustomers.firstOption must beSome[Customer].which
8.     {_.numberOfOrders == 5}
9. }
```

1/1 mobile 360

geschrieben, die als Fabrikmethode für neu zu erzeugende *Customer*-Instanzen fungiert.

Eingebettete Objekte

Der MongoDB-Scala-Treiber unterstützt das Mapping von komplexen eingebetteten Objekten bzw. Aggregaten. Ein Beispiel ist die Eigenschaft *bankData* aus dem Listing 6. Zeile 7 definiert diese Eigenschaft für das *Customer*-Mapping. Die Schreibweise unterscheidet sich von der *scalar*-Fabrikmethode, da hier ein (Scala-)Objekt der Klasse *EmbeddedField* erzeugt wird. Die Parameter des Konstruktoraufrufs von *EmbeddedField* entsprechen fast denen der *scalar*-Fabrikmethode. Wie *BankData*-Instanzen auf ein (BSON-)Dokument abgebildet werden sollen, definiert der *BankDataIn* Trait. Die *BankData* Scala-Klasse selbst ist unveränderlich (immutable), was sich auch im Mapping der Klasse *BankData* widerspiegelt. So wird für keine Eigenschaft ein dritter Parameter bei der *Field*.*scalar*-Methode angegeben und die *factory*-Methode ist wesentlich komplexer als jene des *Customer*-Objekts. Im Grunde genommen werden in der *factory*-Methode jedoch lediglich die Werte für die Konstruktorparameter aus dem *DBObject* ausgelesen, die kompakte Schreibweise ermöglicht dabei das Scala Extractor Pattern bzw. eine for comprehension [9].

More Syntax-Sugar

Um mit dem Java-Beispiel gleich zu ziehen, fehlt noch ein letztes Example aus der Webshopspezifikation, das in Listing 7 zu sehen ist.

Zunächst wird erneut der Kunde Otto Normal in die leere MongoDB-Collection *customer* eingefügt. Anschließend wird in Zeile 3 eine (atomare) Inkrement-Operation ausgelöst. Der erste Parameter dieses Aufrufs definiert eine Query, die hier dafür sorgt, dass nur Kunden mit einer ID „1234“ von der Inkrement-Operation erfasst werden. Die eigentliche Inkrement-Operation wird durch den zweiten Parameter definiert und ist genau wie die vorausgehende Query-Definition völlig typischer. In Zeile 5 werden alle Kunden aus der MongoDB-Collection herausgefiltert, die mehr als vier Bestellungen getätigt haben. Die zuvor durchgeführte Inkrement-Operation sorgt dafür, dass der Kunde *Otto Normal* genau ein Teil dieser Ergebnismenge ist. Diese Behauptung wird durch den Aufruf in Zeile 6 verifiziert, wobei das Specs-BDD-Framework erneut seine Ausdruckskraft ausspielen kann.

Fazit

Dem Einsatz von MongoDB in einem Java-Projekt steht technisch gesehen nichts im Weg. Der MongoDB-Java-Treiber bietet die Funktionalität, um in Java mit einer

MongoDB arbeiten zu können. Darüber hinaus bieten zahlreiche Communityprojekte elegante Mechanismen, um Objektmodelle auf (BSON-)Dokumente abzubilden. In diesem Artikel wurde mit dem MongoDB-Scala-Treiber eine Möglichkeit vorgestellt, wie dies mit der JVM-Sprache Scala funktionieren kann. Die MongoDB-Community bietet aber auch für andere, auf der JVM-lauffähige Sprachen – wie Clojure, JRuby oder Groovy – interessante Projekte. Wem der MongoDB-Scala-Treiber als zu komplex erscheint oder wer Scala in seinem aktuellen Projekt nicht einsetzen

darf, findet auch einige Java-Projekte, die das Arbeiten mit MongoDB effizienter gestalten. Das Projekt Morphia [10] bietet hier z. B. ein Java-Annotations-basiertes Mapping von Klassen auf (BSON-)Dokumente an. Auch in Softwareprojekten, in denen schon eine relationale Datenbank zum Einsatz kommt, lohnt sich der Blick auf MongoDB zur Bewältigung einiger Anforderungen. Mit GridFS bietet das MongoDB-Projekt beispielsweise eine hochskalierbare Lösung für das Speichern von (großen) Binärdaten an. Alle Beispiele aus diesem Artikel finden Sie bei Github unter [12].

Die MongoDB-Community bietet auch für Clojure und Groovy interessante Projekte.

Maximilian Weber ist als Softwareentwickler bei der FlowFact AG in Köln tätig. In einem agilen Team arbeitet er dort an der Entwicklung des eCRM, einem CRM-System für die Immobilienbranche. Seine Themenschwerpunkte sind Architektur und testgetriebene Entwicklung.

Links & Literatur

- [1] <http://www.mongodb.org/display/DOCS/Java+Language+Center>
- [2] <http://www.mongodb.org/display/DOCS/Security+and+Authentication>
- [3] <http://www.martinfowler.com/bliki/FluentInterface.html>
- [4] <http://github.com/alaz/mongo-scala-driver>
- [5] <http://code.google.com/p/specs/>
- [6] <http://behaviour-driven.org/>
- [7] <http://code.google.com/p/specs/w/list>
- [8] <http://programming-scala.labs.oreilly.com/ch08.html#Implicits>
- [9] <http://codemonkeyism.com/scala-goodness-extractors/>
- [10] <http://code.google.com/p/morphia/>
- [12] <http://github.com/maxweber>