

Finding and Fixing Type Mismatches in the Evolution of Object-NoSQL Mappings

Stefanie Scherzinger
OTH Regensburg, Germany
stefanie.scherzinger-
@oth-regensburg.de

Eduardo Cunha
de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

Thomas Cerqueus
University of Lyon, France
thomas.cerqueus
@insa-lyon.fr

Leandro Batista
de Almeida
UTFPR, Brazil
leandro@dainf.ct.utfpr.edu.br

Pedro Holanda
UFPR, Brazil
ptholanda@inf.ufpr.br

ABSTRACT

NoSQL data stores are popular backends for managing big data that is evolving over time: Due to their schema-flexibility, a new release of the application does not require a full migration of data already persisted in production. Instead, using object-NoSQL mappers, developers can specify lazy data migrations that are executed on-the-fly, when a legacy entity is loaded into the application. This paper features ControVol, an IDE plugin that tracks evolutionary changes in object-NoSQL mappings, such as adding, renaming, or removing an attribute, which may conflict with entities already persisted in production. If not resolved prior to launch of the new application, harmful evolutionary changes can cause runtime exceptions or data loss. In this demo, we focus on a novel feature of ControVol, detecting changes to attribute types that are not backwards-compatible with legacy entities. When such changes occur, ControVol issues warnings, and upon the request of developers, assists in safely carrying out type changes.

1. INTRODUCTION

Over the past years, application development for big data management with NoSQL data stores has matured: Developers no longer need to code against proprietary APIs. Instead, object-NoSQL mappers introduce a desirable level of abstraction [15]. Like their counterparts for relational databases, object-NoSQL mappers such as Objectify [9], Morphia [8], or Hibernate OGM [5] marshal and unmarshal between stored entities and objects in the application space. Annotated Java classes, the *object-NoSQL mappings*, declare how objects are to be persisted as entities.

Going beyond this core business, Objectify and Morphia enhance a key feature of NoSQL data stores in agile software development: the schema-flexibility of NoSQL backends [7].

Let us consider a standard development environment with an editor (e.g., an IDE like Eclipse) and a code repository. The production environment contains a schema-flexible NoSQL data store, possibly offered as database-as-a-service (DaaS). A platform-as-a-service infrastructure (PaaS) takes care of the load balancing at runtime. As the data store does not enforce a schema, the entities stored by different releases of the application may differ in their structure. Nevertheless, the NoSQL data store is capable of evaluating queries over the structurally heterogeneous entities.

We now switch to a real-life example. Figure 1 visualizes schema evolution in the object-NoSQL mappings of the open source project ExtraLeague [3]. ExtraLeague implements a small website for managing company-internal soccer championships. This project is written in Java, uses Google App Engine (as PaaS) and Google Cloud Datastore (as DaaS) [10], as well as the object-NoSQL mapper Objectify. At the point of writing this paper, 9 contributors have collectively made about 700 commits to this project hosted on GitHub. The chart reads as follows. The x-axis shows the number of commits to the project, which may be interpreted as the progress in time. The y-axis shows the number of object-NoSQL mappings (i.e., the Java classes that de-

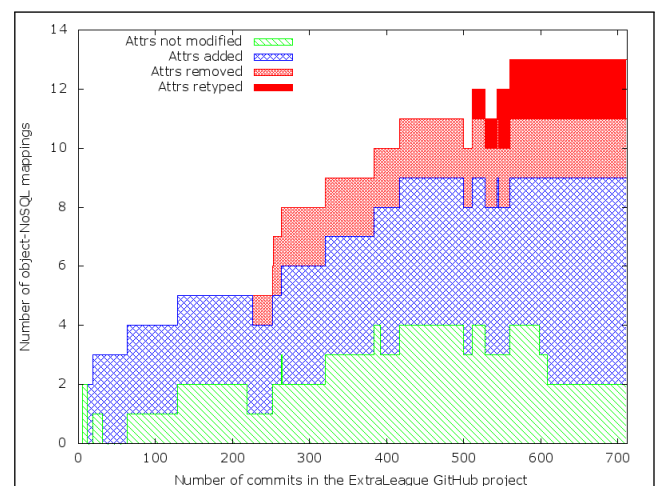


Figure 1: Schema evolution in ExtraLeague [3].

```
{
  "kind": "Player",
  "id": 1,
  "name": "Gollum",
  "health": "poor"
}
```

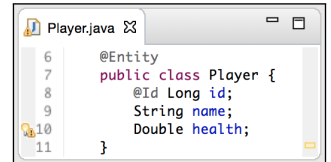
(a) Legacy entity with String-typed health

```
{
  "kind": "Player",
  "id": 2,
  "name": "Bilbo",
  "health": 5
}
```

(b) Legacy entity with Integer-typed health

```
{
  "kind": "Player",
  "id": 3,
  "name": "Frodo",
  "health": 1.2
}
```

(c) Up-to-date entity with Double-typed health



```
Player.java
6 @Entity
7 public class Player {
8     @Id Long id;
9     String name;
10    Double health;
11 }
```

(d) Class Player declaring the health type as Double

Figure 2: (a) – (c) Legacy entities (in JSON format) and (d) the current object-NoSQL mapping.

clare the current data model). Over time, the project grows to 13 mappings. Some object-NoSQL mappings are removed from the project, resulting in interim dips in the chart. At any point in time, the chart states how many classes remain unchanged (“Attrs not modified”). If a mapping contains at least one retyped attribute, it is flagged as such (“Attrs retyped”). Otherwise, if it contains at least one removed attribute, it is also flagged accordingly (“Attrs removed”). Finally, when attributes have been added, but none of the other cases apply, the mapping is flagged as such (“Attrs added”). Thus, we classify the object-NoSQL mappings by their most disruptive schema change.

In total, the object-NoSQL mappings have changed in 47 commits. When the object-NoSQL mappings change with a new release, this effectively amounts to a schema change. Yet rather than migrating all legacy entities prior to a release, legacy entities may be migrated *lazily*, when they are loaded into the application, one at a time. This is supported by several object-NoSQL mappers, such as Objectify.

Lazily evolving an entity by adding an attribute is generally a safe operation: When a legacy entity is loaded into the application, the attribute is added as the entity is mapped to a Java object. When the object is saved again, the entity is persisted with the new attribute.

However, when attributes are removed, renamed, or retyped, the object-NoSQL mappings may no longer be backwards-compatible with legacy entities. Accidental removal or faulty renamings of attributes lead to runtime problems, a topic we have addressed in an earlier paper on ControVol [2].

In this paper, we focus on attribute retypings, which also occur in Figure 1. Retypings can lead to

1. data loss (due to implicit type conversions),
2. runtime exceptions (due to type incompatibilities), and
3. confusing query results (due to the lack of standardization of NoSQL query languages).

These runtime issues may only be sporadic: The production data store may contain only a small number of structural outliers that nobody in the development team is aware of. Yet this makes trouble shooting even more difficult. Therefore, systematic tool support is of the essence.

Contributions: In this demonstration, we present new ControVol features for (1) finding type mismatches in the evolution of object-NoSQL mappings, (2) addressing the subtleties that retyping can have on query evaluation in some NoSQL data stores, and (3) suggesting quick fixes so that developers may pro-actively address these problems. Our earlier demos of ControVol [1, 11] showed how ControVol finds and fixes issues caused by the removal and renaming of attributes in object-NoSQL mappings. This earlier ver-

sion would also warn when attributes were retyped, but then forced the developer to restore the original type. This effectively made it impossible to change attribute types. ControVol, as presented in this paper, actually enables developers to change types in a controlled manner.

Videos and further information on ControVol are available at <https://sites.google.com/site/controvolplugin/>.

2. TYPE MISMATCHES BY EXAMPLE

In discussing mismatched types, we focus on Java primitive types¹: Boolean, Byte, Short, Integer, Long, Float, Double, and String. ControVol may be extended to handle structured types as well.

We next consider the scenarios featured in this demo. Figure 2 shows player entities stored by different releases of an online role playing game. It also shows the object-NoSQL mapping of the current release. The object mapper annotation `@Entity` declares that player objects may be stored. Annotation `@Id` marks the unique ID among all players.

2.1 Runtime Errors due to Type Changes

Our upcoming example illustrates how mismatched types can cause runtime errors. As runtime errors are particularly undesirable in web applications, the underlying problem should be addressed *prior* to launch. We then describe how ControVol finds and fixes these issues.

EXAMPLE 1. The legacy entity for player Gollum in Figure 2(a) records Gollum’s health as a String. Let us assume Gollum’s player has been stored several releases back, when health was classified as “poor”, “fair”, or “excellent”. Much has changed since then: The object-NoSQL mapping in Figure 2(d) expects to load a Double value. Thus, loading Gollum’s entity will cause a runtime exception due to an unsuccessful type cast. □

ControVol monitors code changes from within Eclipse to detect this issue at development time. To do so, ControVol accesses the code repository and compares different versions of object-NoSQL mappings. For instance, if ControVol detects the earlier declaration shown in Figure 3, then it warns about the type mismatch with the declaration in Figure 2(d). Note the warning symbol in line 10 of the screenshot, injected by ControVol.

¹We use the term *primitive type* casually, to refer to classes of the `java.lang` package that wrap Java primitive types (int, long, float, etc.). Void, Character, and Object are currently not considered by ControVol, as Objectify does not support storing values of these types.

```

Player.java
6  @Entity
7  public class Player {
8      @Id Long id;
9      String name;
10     String health; //poor /fair /excellent
11 }

```

Figure 3: Legacy object-NoSQL mapping by which Gollum’s entity from Figure 2(a) was persisted.

```

Player.java
8  @Entity
9  public class Player {
10     @Id Long id;
11     String name;
12
13     @IgnoreSave String health;
14     Double healthDouble;
15
16     @OnLoad
17     private void migrateHealth() {
18         if (healthDouble == null) {
19             // TODO Auto-generated method stub
20         }
21     }
22 }

```

Figure 5: ControVol-generated code stub.

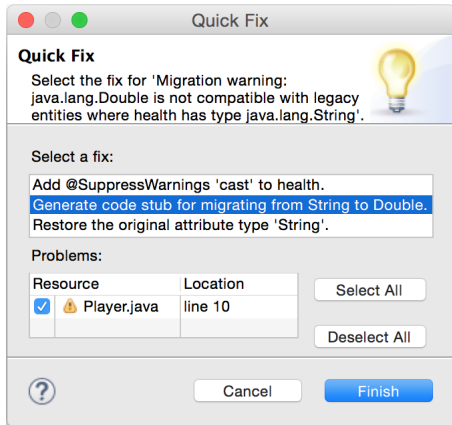


Figure 4: Quick fixes proposed by ControVol.

ControVol also proposes quick fixes to address this issue, as shown in Figure 4. Developers can choose to (1) suppress this warning, (2) generate a code stub for translating the String to a Double, or (3) to change the type back to String.

We discuss the second quick fix in greater detail. If selected, ControVol rewrites the class as shown in Figure 5:²

- The original health attribute of type String has been restored. Annotation `@IgnoreSave` ensures that the value is loaded from legacy entities, but not saved anymore.
- The new health attribute of type Double has been renamed, so as not to conflict with the legacy attribute.
- A method stub for `migrateHealth` has been generated. Due to the Objectify annotation `@OnLoad`, this method is invoked whenever a player entity is loaded.

The developers may now translate the value of legacy attribute `health` to a Double value within `migrateHealth`.³

2.2 Mixed Value Types in Query Evaluation

We next give an example of the effect that attributes with mixed value types can cause during query evaluation.

EXAMPLE 2. The entity for legacy player Bilbo in Figure 2(b) can be loaded by the mapping in Figure 2(d) with-

²In Figure 5, we use the Objectify syntax `@OnLoad`. As the annotations are not standardized for object-NoSQL mappers, we require different annotations when using a different object-NoSQL mapper. ControVol is currently being extended to support Morphia as well.

³Lazy migration using object-NoSQL mappers can be very convenient for small, incremental schema changes. Yet when object-NoSQL mappings need to be compatible with entities from *several* releases back, their declarations become cluttered with migration code. We refer to [13] for a proposal on how multi-step lazy migration may be realized outside of the application layer.

out a runtime exception, since the Integer value is implicitly cast to Double (yet loss of precision is possible). However, as long as Bilbo’s entity is not migrated, this type mix in the data store can produce seemingly confusing query results. For instance, evaluating the query

```
SELECT id FROM Player ORDER BY health ASC LIMIT 10
```

on our entities from Figure 2 in Google Cloud Datastore returns Bilbo’s ID *before* Frodo’s. Yet Bilbo’s health is 5, while Frodo’s health is 1.2 This seems counter-intuitive given ascending sort. Worse, MongoDB returns Frodo’s ID before Bilbo’s for the same query. □

This puzzling behavior is due to the lack of standardization in NoSQL query languages. In Google Cloud Datastore, all queries are evaluated over indexes. The indexes contain hierarchically sorted entries, with primary order on the value type and secondary on a type-specific ordering [4]. Let us consider the index capturing the entities from Figure 2, as well as a fourth player named Peregrin with *id* 4 and a Double health value of 9.9.

In displaying the index as shown in Table 1, we employ the visual notation from [10]: Google Cloud Datastore evaluates the query from Example 2 using an index containing the keys of player entities (consisting of the kind “Player” and the player *id*), and the values of their health properties. These are sorted in ascending order. The index entries are sorted, with Integer values before Strings, and further before floating point values.

Key	health ↑
Player/2	5
Player/1	"poor"
Player/3	1.2
Player/4	9.9

Table 1: Datastore.

Key	health ↑
Player/3	1.2
Player/2	5
Player/4	9.9
Player/1	"poor"

Table 2: MongoDB.

Now, retrieving all player IDs in ascending order of their health is conducted by a single scan over this index. Scanning the index from top to bottom retrieves Bilbo’s ID, then Gollum’s, Frodo’s, and finally Peregrin’s. To NoSQL novices, this may seem surprising, and understandably so: When the entities are loaded as Java objects into the application, Frodo’s health value has type Double (due to the implicit type conversion on loading). This makes it particularly confusing why Bilbo with a health value of 5.0 should

be sorted before Frodo with a health of 1.2. From the viewpoint of the developer, this is a puzzle that can only be solved by inspecting the raw contents of the data store.

Worse yet, this effect is product-specific: Table 2 captures the order in which MongoDB returns the query results. Here, the sort operator returns all numeric values before strings, and hence, a different result.

Thus, even when developers aim at platform independence by using object-NoSQL mappers, the implementation details of NoSQL data stores shine through. Since such effects can be easy to miss, we have set up ControVol to warn if a change to an object-NoSQL mapping might introduce mixed value types. This gives developers a chance to react, e.g., to identify these legacy entities and to *eagerly* migrate them to Double values. This can be done by writing custom code or using declarative, special-purpose schema evolution languages, c.f. [12, 14]. Having made sure that incompatible legacy entities no longer exist, and having allowed the data store indexes sufficient time to be rebuilt (which happens asynchronously in Google Cloud Datastore), the warning issued by ControVol may be suppressed.

2.3 Demo Outline

The general outline for our interactive demo is this:

1. We introduce the typical setup for NoSQL web development: Developers write code in the Eclipse IDE, manage its versions in Git, and regularly deploy the application to a PaaS framework (Google App Engine). The application is backed by a NoSQL data store (Google Cloud Datastore).
2. We show common pitfalls in evolving the application code: Foremost, we focus on issues introduced by type changes in object-NoSQL mappings. We also demonstrate the earlier features of ControVol, such as finding problems caused by renaming or removing attributes.
3. We provoke various consequences of mismatched types: data loss, data corruption, runtime errors, and counter-intuitive query results, as discussed in Section 2. ControVol then finds these issues and proposes quick fixes, which it carries out semi-automatically.
4. We further run ControVol on open source projects publicly hosted on GitHub, and thus, on real-life code.

We discuss the impact of conflicting type declarations with our audience, as well as the tradeoffs between quick fixes.

3. SUMMARY

When it comes to assessing the potential impact of ControVol, we point out two current trends: First, schema-flexible NoSQL data stores are gaining popularity [7], especially in agile web development: When projects undergo frequent releases and cannot afford downtime due to a release, NoSQL data stores can be suitable backends. Second, object-NoSQL mappers with support for lazy migration are gaining popularity. Among the currently popular libraries, there are Objectify for Google Cloud Datastore, and Morphia for MongoDB. The future roadmap for Hibernate OGM explicitly mentions similar plans for lazy data migration [6]. Thus, there seems to be a trend for vendors of object-NoSQL mappers to provide annotation-based support for data migration. Looking at both trends, we see a growing market for tools like ControVol, especially considering the gaping void when it comes to a tooling eco-system for NoSQL-backed application development.

Acknowledgments

ControVol was partly funded by CNPq grant 441944/2014-0. We thank Tegawendé F. Bissyandé from *University of Luxembourg* for suggesting to us to crawl GitHub projects. We further thank Uta Störl from *Darmstadt University of Applied Sciences* for sharing her insights on queries over mixed value types in MongoDB. Last but not least, we thank the anonymous reviewers for the careful reading of our paper and their helpful suggestions.

4. REFERENCES

- [1] T. Cerqueus, E. Cunha de Almeida, and S. Scherzinger. ControVol: Let yesterday's data catch up with today's application code. In *Proc. WWW'15, poster*, 2015.
- [2] T. Cerqueus, E. Cunha de Almeida, and S. Scherzinger. Safely Managing Data Variety in Big Data Software Development. In *Proc. BIGDSE'15*, 2015.
- [3] ExtraLeague, Sept. 2015. Source code available at <https://github.com/squix78/extraleague>, latest release at <http://ncaleague-test.appspot.com>.
- [4] Google Cloud Platform. Datastore Indexes, Jan. 2016. https://cloud.google.com/appengine/docs/java/datastore/indexes/#Java_Properties_with_mixed_value_types.
- [5] Hibernate OGM, Jan. 2016. <http://hibernate.org/ogm/>.
- [6] Hibernate OGM Roadmap for Hibernate OGM 5.0, Jan. 2016. <http://hibernate.org/ogm/roadmap/>.
- [7] Z. H. Liu and D. Gawlick. Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL. In *CIDR'15*, 2015.
- [8] Morphia, Jan. 2016. <https://github.com/mongodb/morphia/>.
- [9] Objectify, Jan. 2016. <https://github.com/objectify/objectify>.
- [10] D. Sanderson. *Programming Google App Engine with Java*. O'Reilly Media, Inc., 2015.
- [11] S. Scherzinger, E. Cunha de Almeida, and T. Cerqueus. ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development. In *Proc. ICDE'15, demo paper*, 2015.
- [12] S. Scherzinger, M. Klettke, and U. Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. DBPL'13*, 2013.
- [13] S. Scherzinger, U. Störl, and M. Klettke. A Datalog-based Protocol for Lazy Data Migration in Agile NoSQL Application Development. In *Proc. DBPL'15*, 2015.
- [14] J. Schildgen and S. DeBloch. NotaQL is not a Query Language! It's for Data Transformation on Wide-Column Stores. In *Proc. BICOD'15*, 2015.
- [15] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger. Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue? In *Proc. BTW'15*, 2015.