# ControVol Flex: Flexible Schema Evolution for NoSQL Application Development

Florian Haubold[1], Johannes Schildgen[2], Stefanie Scherzinger[3], Stefan Deßloch[4]

**Abstract:** We demonstrate *ControVol Flex*, an Eclipse plugin for controlled schema evolution in Java applications backed by NoSQL document stores. The sweet spot of our tool are applications that are deployed continuously against the same production data store: Each new release may bring about schema changes that conflict with legacy data already stored in production. The type system internal to the predecessor tool *ControVol* is able to detect common *schema conflicts*, and enables developers to resolve them with the help of object-mapper annotations. Our new tool *ControVol Flex* lets developers choose their schema-migration strategy, whether all legacy data is to be migrated *eagerly* by means of *NotaQL* transformation scripts, or *lazily*, as declared by object-mapper annotations. Our tool is even capable of carrying out both strategies in combination, eagerly migrating data in the background, while lazily migrating data that is meanwhile accessed by the application. From the viewpoint of the application, it remains transparent how legacy data is migrated: Every read access yields an entity that matches the structure that the current application code expects. Our live demo shows how *ControVol Flex* gracefully solves a broad range of common schema-evolution tasks.

**Keywords:** Schema evolution, NoSQL, NotaQL

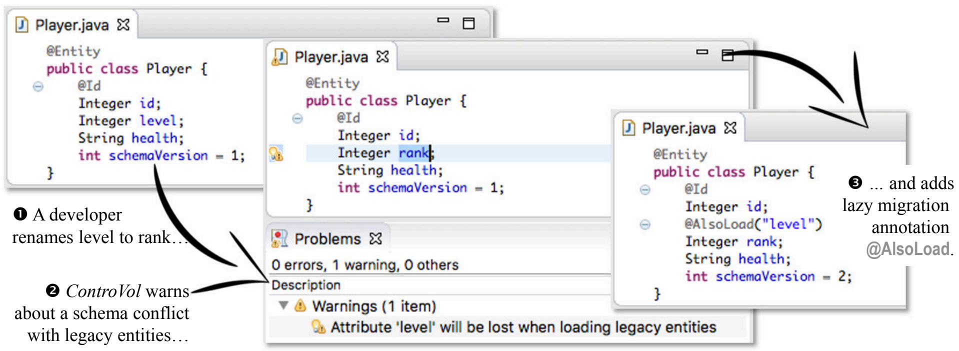
## 1 Purging Migration Debt in Schema-Flexible NoSQL Data Stores

Schema-flexible NoSQL data stores are popular with agile development teams, especially when software is deployed continuously: Even for small, incremental changes of the code, a new release is deployed to production. Each new version of the application declares its own data model or schema, usually encoded within object mapper class declarations.

NoSQL data stores like MongoDB [Mon16] can store *legacy* entities, i.e., entities that adhere to the schema imposed by earlier application releases, as well as entities written by the latest release. Object-NoSQL mappers like Morphia [Mor16] are capable of *lazily* migrating legacy entities to the latest schema, whenever they are accessed by the application. Figure 1(a) describes such a scenario for a gaming application: In the first release, each player written to the data store has a unique *id*, and further information on his or her *level* and *health* status. With the second release of the application, the schema of players changes: Attribute *level* is renamed to *rank*. When a legacy player is now loaded, its *level* value is automatically loaded as *rank*, due to the Morphia annotation *@AlsoLoad*.

Simple changes such as adding, removing, or renaming an attribute can be performed quite gracefully with this approach. However, the third release of the application brings about a

[1] Technische Universität Kaiserslautern, f_haubold12@cs.uni-kl.de
[2] Technische Universität Kaiserslautern, schildgen@cs.uni-kl.de
[3] OTH Regensburg, stefanie.scherzinger@oth-regensburg.de
[4] Technische Universität Kaiserslautern,dessloch@cs.uni-kl.de



Player.java

```
@Entity
public class Player {
    @Id
    Integer id;
    Integer level;
    String health;
    int schemaVersion = 1;
}
```

❶ A developer renames level to rank…

Player.java

```
@Entity
public class Player {
    @Id
    Integer id;
    Integer rank;
    String health;
    int schemaVersion = 1;
}
```

❷ *ControVol* warns about a schema conflict with legacy entities…

Problems

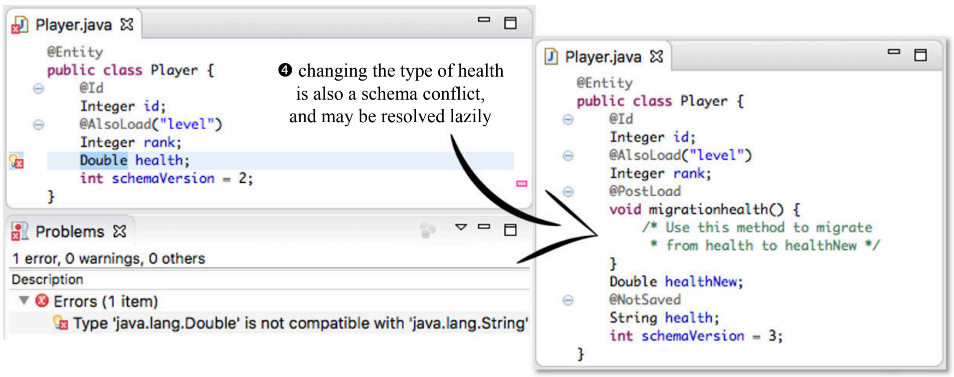
```
0 errors, 1 warning, 0 others
Description
▼ Warnings (1 item)
    Attribute 'level' will be lost when loading legacy entities
```

Player.java

```
@Entity
public class Player {
    @Id
    Integer id;
    @AlsoLoad("level")
    Integer rank;
    String health;
    int schemaVersion = 2;
}
```

❸ … and adds lazy migration annotation @AlsoLoad.

(a) Lazily renaming *level* to *rank* using the Morphia annotation @*AlsoLoad*.



Player.java

```
@Entity
public class Player {
    @Id
    Integer id;
    @AlsoLoad("level")
    Integer rank;
    Double health;
    int schemaVersion = 2;
}
```

❹ changing the type of health is also a schema conflict, and may be resolved lazily

Problems

```
1 error, 0 warnings, 0 others
Description
▼ Errors (1 item)
    Type 'java.lang.Double' is not compatible with 'java.lang.String'
```

Player.java

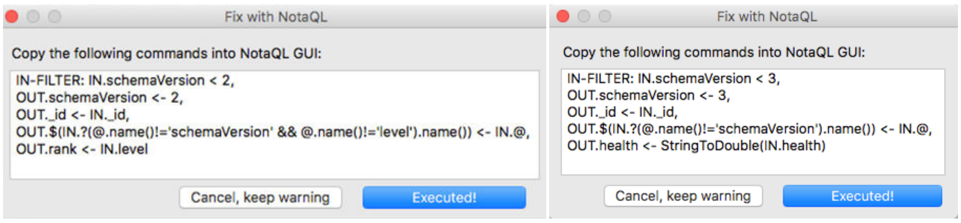
```
@Entity
public class Player {
    @Id
    Integer id;
    @AlsoLoad("level")
    Integer rank;
    @PostLoad
    void migrationhealth() {
        /* Use this method to migrate
         * from health to healthNew */
    }
    Double healthNew;
    @NotSaved
    String health;
    int schemaVersion = 3;
}
```

(b) Lazily retyping *health* from String to Double, writing custom code.

Fig. 1: Building up migration debt while lazily evolving the declaration of class Player.

more complex change, as shown in Figure 1(b): The players' health is no longer recorded as a String, but is stored as a floating point value. Lazily retyping values can be done: Whenever a player entity is loaded, the method annotated @*PostLoad* is invoked. Now, developers need to write the code to translate the legacy *health* value (no longer stored due to annotation @*NotSaved*), to a Double (stored as *healthNew*). Already in this simple scenario, we see how quickly we build up technical debt in the form of *migration debt*: Player classes now carry two *health* attributes, to distinguish legacy values from up-to-date values. This can be confusing to newcomers in the project. Moreover, immersing migration code in class declarations violates the software engineering principle of separation of concerns. Additionally, all queries issued by the application code (rather than accessing a single entity by its key) need to consider all structural variations of legacy entities. Overall, application development is slowed down due to the need to account for the structural heterogeneity of legacy entities. At some point in time, *eager* migration of all legacy entities is called for.

Today, developers lack the tool support for systematically managing schema evolution in settings such as these. That is, we need to provide a development environment that


Fix with NotaQL
Copy the following commands into NotaQL GUI:
Cancel, keep warning
Executed!

```
IN-FILTER: IN.schemaVersion < 2,
OUT.schemaVersion <- 2,
OUT._id <- IN._id,
OUT.$(IN.?(@.name()!='schemaVersion' && @.name()!='level').name()) <- IN.@,
OUT.rank <- IN.level
```

(a) Migrating to schema version 2.

Fix with NotaQL
Copy the following commands into NotaQL GUI:
Cancel, keep warning
Executed!

```
IN-FILTER: IN.schemaVersion < 3,
OUT.schemaVersion <- 3,
OUT._id <- IN._id,
OUT.$(IN.?(@.name()!='schemaVersion').name()) <- IN.@,
OUT.health <- StringToDouble(IN.health)
```

(b) Migrating to schema version 3.

Fig. 2: *NotaQL* scripts to eagerly migrate legacy entities, as produced by *ControVol Flex*.

1. keeps track of the various schema versions that occur in the production data store,
2. warns developers about possible schema conflicts when they make changes to class declarations that are incompatible with legacy entities,
3. automatically fixes detected schema conflicts lazily, and further
4. provides easy means so that developers may migrate legacy data eagerly as well.
5. Finally, a tool that even allows to carry out eager and lazy data migration concurrently, which is vital for the continuous deployment of zero-downtime applications.

In earlier work, we have presented *ControVol*, an Eclipse plugin that meets desiderata (1) through (3) [CCS15; SCC15]. In this demo, we introduce its successor *ControVol Flex*, the first tool that meets all five desiderata: *ControVol Flex* generates *NotaQL* [SD15; SLD16] scripts for eager data migration, upon the push of a button. The only requirement that *ControVol Flex* imposes is that all object mapper class declarations carry a dedicated attribute *schemaVersion* (c.f. Figure 1), maintained by *ControVol Flex*. This is a reasonable requirement: Empirical analysis of open source projects shows that maintaining timestamps or versions in persisted entities is common practice in the developer community [RSB16].

Regarding our example, the script in Figure 2(a) transforms all legacy entities written before version 2 of the application code (c.f. line 1) by an update in place: The *NotaQL* commands are read from right-to-left, where the right side matches parts of the input entity (IN), and the left side declares the change to the output entity (OUT). The identifying property *id* (mapped to the MongoDB-internal identifier *_id* by Morphia) is preserved (line 3), as are all properties other than the *level* and *schemaVersion* (lines 4). In fact, the value of a *level*-property is renamed to *rank* (line 5). In line 2, the dedicated property *schemaVersion* is upgraded to 2. Analogously, the *NotaQL* script in Figure 2(b) recasts *health* attributes. By applying both scripts, all legacy entities are eagerly upgraded to schema version 3, and thus the structure expected by the current application code.

A major advantage of *NotaQL* is that this transformation language is independent of a particular data store and even data model: This provides a convenient level of abstraction compared to system-specific APIs or aggregation pipelines. Further, developers may edit the generated *NotaQL* scripts, to unleash the full power of this transformation language in eager migration: *NotaQL* supports complex changes such as nesting and unnesting of hierarchical data, as well as arrays and aggregation operations. As such, it is a powerful tool at the hands of developers for conveniently purging migration debt from NoSQL backends.

## 2 Demonstration Outline

Our demo scenario describes the agile software-development process of an online role-playing game. The general outline for our interactive demo is this:

1. We introduce a generic development setup using the Eclipse IDE, the Java programming language, the NoSQL data store MongoDB, and the Morphia object mapper.
2. We demonstrate how schema conflicts can occur due to continuous deployment. We provoke serious problems, such as data loss by renaming attributes, type errors by changing attribute types, and missing default values by adding new attributes. *ControVol Flex* detects these conflicts and proposes appropriate quickfixes in Eclipse.
3. We then show how *ControVol Flex* helps to migrate the NoSQL schema lazily by adding Morphia annotations to our code. We also show how *ControVol Flex* generates *NotaQL* scripts to eagerly migrate legacy entities. We point out how user-friendly our plugin is by generating one script for all or even just selected schema conflicts.
4. We demo the two *hybrid modi operandi* of *ControVol Flex*: (1) First kicking off eager migration in the background, while migrating legacy entities lazily, if the application requests access and eager migration has not reached them yet. Alternatively, (2) starting out with lazy migration, and then cleaning up the remaining legacy entities to bring the data instance into a consistent state. We show that application development remains unimpaired by the mode chosen.
5. Furthermore, we demonstrate the automatic version-numbering mechanism for the different stages of our schema evolution process.

**Acknowledgements:** The authors are grateful to the extended team who has built the predecessor *ControVol*: Eduardo Cunha de Almeida and Pedro Holanda from UFPR Brazil, Thomas Cerqueus from University of Lyon, and Dennis Schmidt from OTH Regensburg.

## References

[CCS15] Cerqueus, T.; Cunha de Almeida, E.; Scherzinger, S.: Safely Managing Data Variety in Big Data Software Development. In: Proc. BIGDSE'15. 2015.

[Mon16] MongoDB, http://www.mongodb.org/, 2016.

[Mor16] Morphia, https://github.com/mongodb/morphia/, 2016.

[RSB16] Ringlstetter, A.; Scherzinger, S.; Bissyandé, T. F.: Data Model Evolution using Object-NoSQL Mappers: Folklore or State-of-the-Art? In: Proc. BIGDSE. 2016.

[SCC15] Scherzinger, S.; Cunha de Almeida, E.; Cerqueus, T.: ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development. In: Proc. ICDE'15, demo paper. 2015.

[SD15] Schildgen, J.; Deßloch, S.: NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores. In: Proc. BICOD'15. 2015.

[SLD16] Schildgen, J.; Lottermann, T.; Deßloch, S.: Cross-system NoSQL data transformations with NotaQL. In: Proc. BeyondMR'16. 2016.