ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development

Stefanie Scherzinger, Thomas Cerqueus, Eduardo Cunha de Almeida

ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development

Stefanie Scherzinger*, Thomas Cerqueus† and Eduardo Cunha de Almeida‡
* OTH Regensburg, stefanie.scherzinger@oth-regensburg.de
† Lero@UCD, School of Computer Science and Informatics, University College Dublin, thomas.cerqueus@ucd.ie
‡ UFPR, Brazil & University of Luxembourg, eduardo@inf.ufpr.br

***Abstract*—Building scalable web applications on top of NoSQL data stores is becoming common practice. Many of these data stores can easily be accessed programmatically, and do not enforce a schema. Software engineers can design the data model on the go, a flexibility that is crucial in agile software development. The typical tasks of database schema management are now handled within the application code, usually involving object mapper libraries. However, today's Integrated Development Environments (IDEs) lack the proper tool support when it comes to managing the combined evolution of the application code and of the schema. Yet simple refactorings such as renaming an attribute at the source code level can cause irretrievable data loss or runtime errors once the application is serving in production. In this demo, we present ControVol, a framework for controlled schema evolution in application development against NoSQL data stores. ControVol is integrated into the IDE and statically type checks object mapper class declarations against the schema evolution history, as recorded by the code repository. ControVol is capable of warning of common yet risky cases of mismatched data and schema. ControVol is further able to suggest quick fixes by which developers can have these issues automatically resolved.**

I. Introduction

As software evolves in building web applications, so does the schema of persisted data. Keeping the schema in sync with the application code is a known challenge, and remains an active research area. Systematic *database schema evolution control*, to use a term coined in [1], arises in relational data stores, object-oriented data stores, XML and NoSQL data stores alike [2]–[4].

However, with agile software development, the frequency of evolutionary changes is reaching a new and unprecedented rate: A recent study on the web application managing Wikipedia shows that the database schema in this software project experiences at least one publicly released schema change per month [5]. Large web companies such as Google make weekly, if not daily releases (quoting Marissa Meyer in [6]). As a consequence, parts of the developer community turn to schemaless NoSQL data stores which provide more flexibility compared to relational databases.

NoSQL document stores such as Google Cloud Datastore [7] or MongoDB [8] are popular backends for web development, since they allow for storing structured data (rather than just opaque values) and offer a basic support for queries. While these data stores do not enforce a global schema, professional software engineering actually relies on data being structured consistently, both in the objects of the application code and in the persisted entities. Developers therefore turn to object mappers for the translation of persisted entities into objects in the application space, and back. Thus, the schema of persisted data is now maintained within the application code.

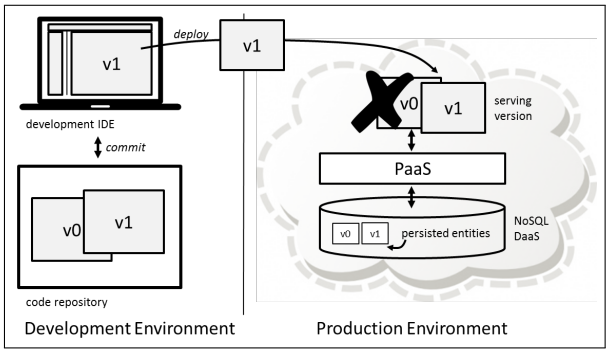


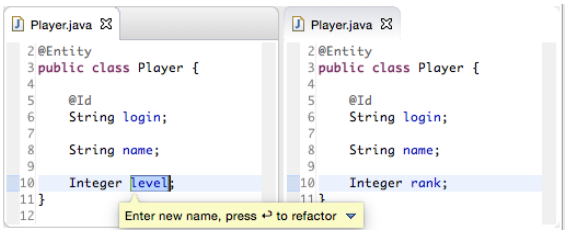
Fig. 1: Deploying version v1 of the application from development into production. Version v0 stops serving, yet its persisted *legacy* entities remain in the NoSQL data store.

Let us sketch the typical setup of a web development project, e.g., using a platform-as-a-service (PaaS) stack for hosting web applications. Figure 1 shows the development environment on the left. The software engineers have just finalized version v1 of the application using an Integrated Development Environment (IDE). The code repository manages the source code versions, and currently contains the upcoming release v1, as well as the earlier release v0. On the right, version v0 of the application has been serving in production until now. The application is backed by a NoSQL data store, offered as database-as-a-service (DaaS).

Once version v1 is ready, the engineers deploy it to production and the serving version v0 is replaced. After deployment, version v1 is now serving users and persisting its own entities into the data store, adding to the entities persisted by version v0, which have now become *legacy* entities.

The new application code, in particular its object mapper class declarations, must be able to also handle the legacy entities, a requirement that brings about a range of challenges. For instance, when the current application code loads a legacy entity that it cannot properly handle, this can trigger runtime errors or can lead to severe cases of data loss. Currently, there is no proper tool support within IDEs to detect such problems early on. Instead, developers need to rely on their discipline, foresight, and on exhaustive testing.

What is missing in the development environment is a



(a) Before renaming. (b) After renaming.

Fig. 2: Refactoring at the risk of data loss: Attribute *level* is renamed to *rank*. The IDE consistently changes all references in the source code, but this does not affect already persisted entities. When legacy players are loaded, their *level* attribute cannot be matched and is consequently not loaded. If the player is then persisted, its *level* data is lost.

framework that statically type checks object mapper class declarations and gives immediate feedback, already during the development process.

Type checking in database programming has a long history [9]. In this tradition, we present ControVol, an Eclipse plugin capable of detecting problems related to schema evolution. To the best of our knowledge, ControVol is the first tool of its kind specifically designed for NoSQL data stores.

Contributions: The main contributions of our demo are:

- We discuss the acute problem of managing schema evolution in agile web development against schemaless NoSQL data stores.
- We present the ControVol Eclipse plugin, which successfully integrates with established Java object mappers.
- ControVol statically type checks object mapper class declarations against earlier versions in the code repository. It issues valuable warnings during changes to the code, thus providing instant feedback to developers.
- ControVol also suggests and performs automatic fixes to resolve possible schema migration problems.

Organization: The rest of this paper is organized as follows. Section II describes common schema migration pitfalls in refactoring object mapper class declarations. Section III introduces the ControVol framework with its core features and underlying type checking scheme. Section IV sketches the demonstration plan. We conclude with Section V.

II. Schema Evolution Pitfalls

We now discuss common pitfalls in evolving the software regardless of the data already persisted in production. These pitfalls are rooted in typical Schema Modification Operations, as they commonly arise in web development projects [5]. We start with discussing a concrete example in greater detail, and then list further common pitfalls.

Renaming Attributes: We assume an online role playing game. Figure 2(a) shows the Java declaration of a Player class. The object mapper annotation `@Entity` declares that instances of this class can be persisted. A player's login serves as the unique key (`@Id`). Based on the annotations, object mappers take care of the marshalling and unmarshalling between objects

Fig. 3: Refactoring without data loss: Annotation `@AlsoLoad` ensures that when loading legacy entities, attribute *level* will also be loaded and renamed to *rank*.

in the application space and entities persisted in the NoSQL data store.

Figure 2 also shows that the developer is about to rename the attribute *level* to *rank*. IDEs such as Eclipse provide convenient refactoring support, and will consistently change all references to this attribute in the source code. The developer then commits the new version of the code into the code repository. With the next release of the application, this code is deployed to production. Yet, if the new application loads a persisted legacy player with the new class declaration, their *level* data will not be loaded, since there is no matching class attribute. Moreover, the *level* is irretrievably lost once the object has been persisted in its new form.

This seemingly innocent refactoring step at the level of the source code causes data loss, since NoSQL object mappers will generally not throw an exception when encountering unmatched attributes in persisted entities. As the data loss is "silent", it requires extensive and targeted testing if it is to be caught prior to launch. In addition, we also risk runtime errors if the remaining code relies on the *rank* attribute being set to a meaningful value.

A safe way to rename an attribute, shown in Figure 3, uses an object mapper library capable of *lazy* schema evolution. Mappers such as Objectify [10] and Morphia [11] provide dedicated migration annotations. In Objectify, `@AlsoLoad` declares that if an attribute *level* is present when loading a persisted player entity, its value is loaded as the *rank* attribute. This class declaration safely handles both the legacy players having a *level*, as well as the players with a *rank*. All legacy players loaded by the application will thus be migrated. The change is persisted when the object is saved to the data store.

Further Schema Migration Pitfalls: Besides renaming attributes, there are further pitfalls in refactoring object mapper class declarations:

- Changing the type of an attribute may yield errors at runtime if the types are incompatible, e.g., converting from *String* to *Integer*. Some type changes are more subtle, e.g., from *Float* to *Integer*. Objectify will not raise an exception, but the result of type conversion might not be anticipated by the developer (e.g., a truncated value).
- Reintroducing attributes that are still present in ancient legacy entities may yield unexpected values when loading these entities, since the values from legacy entities may have been written with different semantics.

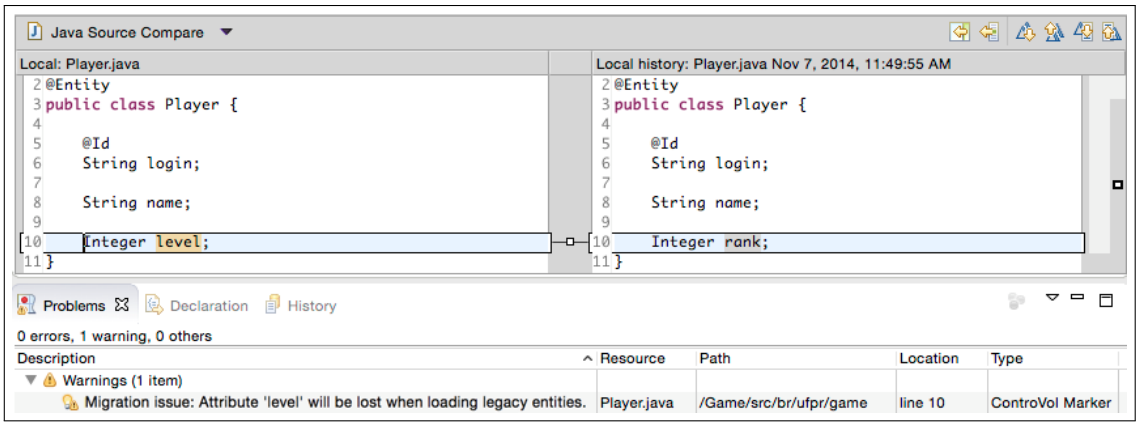


Fig. 4: The ControVol Eclipse plugin warns that in the latest version of class Player (shown to the right), attribute *level* from an earlier class declaration (shown on the left) may have been inadvertently removed (or renamed).

- Ambiguous migration annotations can yield runtime errors. For instance, Objectify throws a runtime exception when trying to load a legacy object with both a *level* and a *rank* attribute with the class declaration from Figure 3.

With frequent releases, it becomes easy to lose track of the schema evolution history. Relying on the developers' discipline alone is risky. Thus, a tool that can automatically check for these pitfalls is of great value in agile software development.

III. The ControVol Type Checking Scheme

The core of ControVol is its static type checking system which checks the compatibility of object mapper class declarations. ControVol has access to the code repository, and thus the full history of all released class declarations. With each change to an object mapper class declaration, ControVol type checks against the schema evolution history of this class. In sketching out the basic idea in the following, we use a notation inspired by the Machiavelli system [12].

Schema Migration Warnings: We reconsider the problem of renaming attributes. From the class declaration in Figure 2(a) we derive the following mapping function.

$$\begin{aligned} \text{Player}_{2(a)} : \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \text{level} : \text{Integer}]\!]\} \\ \rightarrow \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \text{level} : \text{Integer}]\!]\} \end{aligned}$$

The domain of this mapping captures the entities that can be loaded *safely* (i.e., without data loss or runtime exceptions). The codomain captures the entities persisted according to the object mapper class declaration. In our example, the mapper expects all players to have the attributes *login*, *name*, and *level*, and persists entities of the same type.

The class declaration from Figure 2(b) expects an attribute *rank* (instead of *level*):

$$\begin{aligned} \text{Player}_{2(b)} : \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \text{rank} : \text{Integer}]\!]\} \\ \rightarrow \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \text{rank} : \text{Integer}]\!]\} \end{aligned}$$

Now the legacy players that were persisted according to the class declaration of Figure 2(a) cannot be safely loaded according to the class declaration of Figure 2(b). In other words, the codomain of $\text{Player}_{2(a)}$ and the domain of $\text{Player}_{2(b)}$ do not match. While legacy entities will be loaded without runtime errors, the object mapper cannot load the *level* attribute. Objectify will instead create a new *rank* attribute, initialized to zero by default.

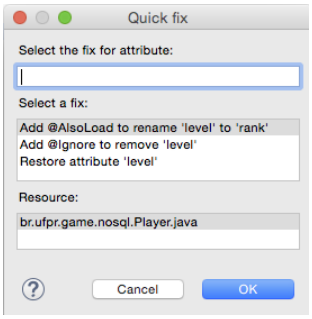


Fig. 5: ControVol suggests quick fixes to resolve warnings.

ControVol recognizes a potential pitfall[1] and reports a warning in the IDE, as shown in Figure 4. For the convenience of the developers, we show the conflicting class versions side by side (see Figure 4).

Quick Fixes: Developers using IDEs like Eclipse are accustomed to resolving warnings with so-called *quick fixes*, which can be applied automatically with only a few mouse clicks. In this tradition, ControVol proposes quick fixes to resolve the migration warning from Figure 4, as can be seen in Figure 5:

- Adding Objectify annotation `@AlsoLoad("level")` to attribute *rank* explicitly renames *level* to *rank*,
- adding annotation `@Ignore` to attribute *level* makes clear that *level* is removed intentionally, and
- restoring attribute *level* prevents losing its value. In this case, attributes *level* and *rank* co-exist.

Annotations Recognized by ControVol: ControVol type

[1] If developers use the refactoring tools within the IDE for renaming attributes, ControVol recognizes this specifically as a renaming issue. Otherwise, ControVol suspects that attribute *level* has been inadvertently removed.

checks the following Objectify annotations to attributes[2].

- `@Id` marks the identifying key of the object.
- `@Ignore` denotes that an attribute will not be loaded from the data store or saved to the data store.
- `@IgnoreSave` denotes that an attribute will be loaded from the data store, but not saved.
- `@IgnoreLoad` denotes that an attribute will not be loaded from the data store, but it will be saved to the data store.
- `@AlsoLoad` denotes that an attribute will be renamed.

For instance, let us consider how ControVol treats `@AlsoLoad` in the Java class declaration from Figure 3. The inferred mapping function performs pattern matching:

$$\begin{aligned} \text{Player}_3 : \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \mathbf{level} : \text{Integer}]\!]\} \\ \rightarrow \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \mathbf{rank} : \text{Integer}]\!]\} \\ \text{Player}_3 : \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \mathbf{rank} : \text{Integer}]\!]\} \\ \rightarrow \quad & \{[\![\text{login} : \text{String},\ \text{name} : \text{String},\ \mathbf{rank} : \text{Integer}]\!]\} \end{aligned}$$

This object mapper class declaration loads legacy and current players alike. Since the entities persisted by class declarations $\text{Player}_{2(a)}$ and $\text{Player}_{2(b)}$ can be safely loaded by Player_3, ControVol reports no warnings.

Objectify also offers life-cycle annotations for methods. A method with annotation `@OnLoad` is called when an entity is loaded. Developers can thus declare more complex migrations, such as splitting Strings, or extracting class member attributes into embedded classes (c.f. [13]). Since these methods may contain arbitrary Java code, the migration problems that can be recognized by static code analysis are inherently limited. We plan to extend ControVol so that simple yet common migrations within methods may be analyzed as well.

IV. Outline of the Demonstration

The general outline for our interactive demo is this:

1) We introduce the typical setup for NoSQL web development, as shown in Figure 1: Developers write source code in an IDE (Eclipse), manage it in a version control system (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 refactoring the application code without also changing the schema of the persisted legacy entities: changing the types of attributes, renaming or removing attributes, and ambiguous annotations.
3) We demonstrate the imminent consequences of such mistakes: data loss, data corruption, and runtime errors.
4) We let the ControVol Eclipse plugin detect these mistakes. For instance, Figure 2 shows a case of inadvertent data loss by the seemingly harmless renaming of attributes in the IDE. ControVol then issues warnings (see Figure 4).
5) We let ControVol suggest quick fixes (see Figure 5). Developers can then conveniently resolve the detected problems, as depicted in Figure 3.

[2]We list the Objectify annotations, as the Morphia annotations are similar.

V. Summary

While schema evolution has been intensively studied for relational, object-oriented, and XML databases, the problem presents itself with new acuteness in the construction of web applications on top of NoSQL data stores. With an ever-growing developer community using object mappers capable of lazy schema migration, there is a pressing need to address their concerns. Earlier academic solutions, while extremely valuable research contributions, cannot be directly applied: Either these solutions were not designed with NoSQL data stores in mind (but for relational databases [14]), or they were not developed for widely adopted web programming languages such as Java, as well as their object mappers (e.g., versus Eiffel in [15]). We regard this as an opportunity for the data management community to contribute our experience in building data management tools.

We plan to release the ControVol Eclipse plugin under an open source licence, and to extend it with a query rewriting capabilities, so that queries also work for legacy entities.

Acknowledgments

This work was partially supported by Science Foundation Ireland grant 10/CE/I1855 to Lero (www.lero.ie), SERPRO Brazil and National Research Fund Luxembourg TOOM Project: C12/IS/4011170. We thank Maximilian Böhm for his early ControVol prototype that he built as part of his Bachelor thesis project at OTH Regensburg.

References

[1] J. Andany, M. Léonard, and C. Palisser, "Management Of Schema Evolution In Databases," in *Proc. VLDB '91*, 1991.

[2] M. Hartung, J. F. Terwilliger, and E. Rahm, "Recent Advances in Schema and Ontology Evolution," in *Schema Matching and Mapping*, 2011, pp. 149–190.

[3] S. Scherzinger, M. Klettke, and U. Störl, "Managing Schema Evolution in NoSQL Data Stores," in *Proc. DBPL*, 2013.

[4] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., 2003.

[5] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema Evolution in Wikipedia - Toward a Web Information System Benchmark," in *Proc. ICEIS*, 2008, pp. 323–332.

[6] S. Lightstone, *Making it Big in Software*. Prentice Hall, 2010.

[7] Google Developers, "Google Cloud Datastore," Nov. 2014, https://developers.google.com/datastore/.

[8] "MongoDB," Nov. 2014, http://www.mongodb.org/.

[9] M. P. Atkinson and O. P. Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, vol. 19, no. 2, pp. 105–170, 1987.

[10] "Objectify," Nov. 2014, https://code.google.com/p/objectify-appengine/.

[11] "Morphia. A type-safe Java library for MongoDB," Nov. 2014, https://github.com/mongodb/morphia/.

[12] P. Buneman and A. Ohori, "Polymorphism and Type Inference in Database Programming," *ACM Transactions on Database Systems*, vol. 21, no. 1, pp. 30–76, 1996.

[13] "Objectify: Migrating Schemas," Nov. 2014, https://code.google.com/p/objectify-appengine/wiki/SchemaMigration.

[14] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the Database Schema Evolution Process," *The VLDB Journal*, vol. 22, no. 1, pp. 73–98, 2013.

[15] M. Piccioni, M. Oriol, and B. Meyer, "Class Schema Evolution for Persistent Object-Oriented Software: Model, Empirical Study, and Automated Support," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 184–196, 2013.