ControVol: Let yesterday's data catch up with today's application code

Thomas Cerqueus, Eduardo Cunha de Almeida, Stefanie Scherzinger

ControVol: Let yesterday's data catch up with today's application code

Thomas Cerqueus
Université de Lyon, CNRS
Lyon, France
thomas.cerqueus@insa-lyon.fr

Eduardo Cunha de Almeida
Federal University of Paraná
Curitiba, Brazil
eduardo@inf.ufpr.br

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

ABSTRACT

In building software-as-a-service applications, a flexible development environment is key to shipping early and often. Therefore, schema-flexible data stores are becoming more and more popular. They can store data with heterogeneous structure, allowing for new releases to be pushed frequently, without having to migrate legacy data first. However, the current application code must continue to work with any legacy data that has already been persisted in production. To let legacy data structurally "catch up" with the latest application code, developers commonly employ object mapper libraries with life-cycle annotations. Yet when used without caution, they can cause runtime errors and even data loss. We present ControVol, an IDE plugin that detects evolutionary changes to the application code that are incompatible with legacy data. ControVol warns developers already at development time, and even suggests automatic fixes for lazily migrating legacy data when it is loaded into the application. Thus, ControVol ensures that the structure of legacy data can catch up with the structure expected by the latest software release. A demo video on ControVol is available at `http://tinyurl.com/mh7a743`.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Integrated environments; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability, Validation*

General Terms

Design, Reliability, Verification

Keywords

NoSQL web development, software- and schema evolution

1. INTRODUCTION

WWW 2015 Companion, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
Include the http://DOI string/url which is specific for your submission.

In agile development of software-as-a-service applications, software developers commonly strive to ship early and often: A first release is made as early as possible, to catch the time-to-market window. The software is then evolved incrementally, to improve performance and to account for user feedback. This approach requires flexible development and production environments. For instance, platform- and database-as-a-service products such as Google App Engine [1] and Google Cloud Datastore are very popular software stacks, especially in the startup community:

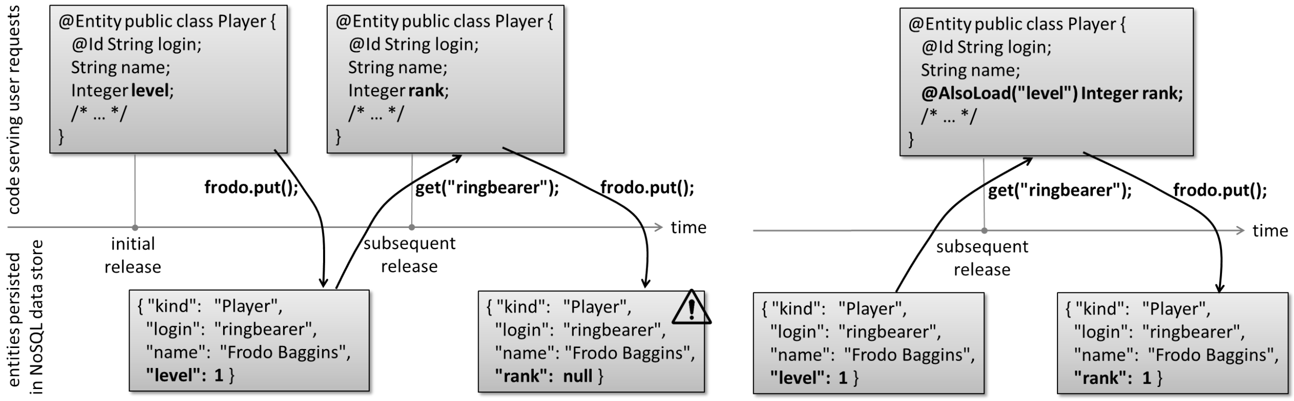
1. When applications are hosted by a platform-as-a-service provider, the developers can focus on application development, without having to worry too much about the scalability of their infrastructure.
2. Platform-as-a-service providers commonly support sophisticated release management tasks out-of-the-box. For instance, they can host different versions of an application running against the same data store, to allow for A/B-testing of new features[1].
3. If the data is persisted in a NoSQL data store, especially when provided as-a-service, developers do not need to worry about storage limits. Moreover, schema-flexible NoSQL data stores can persist structurally heterogeneous data. This makes it possible to re-release the application without prior data migration, since the data stored in production up to that point need not be migrated immediately.

Yet eventually, the legacy data needs to structurally "catch up" with the latest application code, to allow for long-term maintainability and efficiency in software development.

We illustrate this in Figure 1(a). Above the time line, we see Java class declarations for players in an online role playing game. Players are identified by their login. In the initial release, each player has a name and is playing at a certain level. The object mapper annotation `@Entity` makes player objects persistable. Object mapper libraries such as Objectify [2] simplify application development by taking care of persisting and loading objects. The annotation `@Id` marks the identifying attribute. Below the time line to the left, we see a JSON entity persisted according to the initial release. NoSQL data stores are commonly accessed programmatically, by a simple API with a `put()` command for storing, and a `get()` command for loading objects by their key.

In the subsequent release, attribute *level* is renamed to *rank*. While all incoming user requests are now being served

[1] E.g., traffic splitting in Google App Engine `https://cloud.google.com/appengine/docs/adminconsole/trafficsplitting`.



(a) Legacy-data agnostic application code can lead to data loss. (b) Legacy-data aware application code.

Figure 1: (a) Coding without regard to legacy data, the latest object mapper class declaration for players can no longer map the legacy entities without data loss. The object mapper life-cycle annotations in (b) allow legacy data to lazily "catch up" with the new application code when it is loaded into the application.

by the new application code, the NoSQL data store still contains objects persisted by the earlier version. We consider data like Frodo's JSON entity with the outdated structure a *legacy entity*. If Frodo's player is now loaded into the application, not all class member attributes can be mapped. The unmapped *rank* attribute is set to *null*. Worse yet, when the object is persisted (overwriting the legacy entity with the `put()` command), the value of *level* is irretrievably lost.

Figure 1(b) shows a revised class declaration that allows for the legacy data to "catch up" with what the latest application code expects to load from storage: Several NoSQL object mappers provide migration-specific life-cycle annotations or have announced them on their feature roadmap [3]. For instance, due to the Objectify annotation `@AlsoLoad`, Frodo's legacy entity can be loaded by the latest application code without data loss: Frodo's *level* value is also loaded and then assigned to the *rank* attribute. The next time that Frodo's player object is persisted, its *level* is stored as *rank*.

Today, the robustness of lazy data migration with the help of object mappers completely relies on the developers' discipline and foresight to properly specify the annotations: Developers work without any tool support that could reliably catch problems already at development time, and ideally, to automatically suggest the proper life-cycle annotations.

2. CONTRIBUTIONS AND OUTLINE

In our poster presentation, we lay out a generic setup for building software-as-a-service applications. In particular, we introduce Google App Engine and Google Cloud Datastore as platform- and database-as-a-service products, and use the Objectify object mapper for loading and storing objects.

1. We show how seemingly innocuous changes to the application code can lead to runtime errors or data loss when the released software encounters incompatible legacy data. In particular, we consider problems involving adding, removing, and renaming class member attributes in class declarations. We also consider problems related to changes in the attribute types.
2. We demo ControVol, an Eclipse plugin that tracks all changes to the source code and automatically checks for compatibility with the complete release history, as available in the source code repository.
3. ControVol detects the precarious changes from (1), issues warnings, and even proposes to automatically fix its findings. Thus, ControVol ensures that legacy data can catch up with the data structures expected by the latest code release, lazily migrating legacy data through object mapper life-cycle annotations.
4. Our poster explains the internal typing rules by which ControVol checks changes to object mapper class declarations for backward compatibility. We introduce these rules in [4] in greater detail.
5. An earlier version of ControVol has been demoed at ICDE'15 [5]. As a new feature since that first prototype, our poster presentation at WWW will show how ControVol detects a new class of problems caused by re-introducing class member attributes that have been removed from the source code in earlier releases, but may still be resident in legacy data. We refer to [4] for a more detailed discussion on the problem of re-introducing attributes than can be given here.

Being integrated into the Eclipse IDE, ControVol guides developers in building robust applications that are not only backwards-compatible with legacy data, but that also allow the data to catch up lazily with the latest application code.

3. REFERENCES

[1] D. Sanderson, *Programming Google App Engine*, 2nd ed. O'Reilly Media, Inc., 2012.

[2] "Objectify v5," Mar. 2015, https://code.google.com/p/objectify-appengine/.

[3] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger, "Schemaless NoSQL Data Stores – Object-NoSQL Mappers to the Rescue?" in *Proc. BTW'15*, 2015.

[4] T. Cerqueus, E. C. de Almeida, and S. Scherzinger, "Safely Managing Data Variety in Big Data Software Development," in *Proc. BIGDSE'15*, 2015.

[5] S. Scherzinger, E. C. de Almeida, and T. Cerqueus, "ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development," in *Proc. ICDE'15*, 2015.