Java Library for Database Schema Migration Management in Postgres DBMS

Michael Belkin
Faculty of Computer Science
Higher School of Economics
Moscow, Russia
myabelkin@edu.hse.ru

***Abstract*—Almost any backend service must somehow persist data it operates. A database is a necessary element of any service infrastructure. Databases are well-organized data, which are administrated by database management systems. The structure of data is formally described by database schema, and it is often necessary to modify it. For this reason, database schema migration management tools are used. The paper proposes a new lightweight migration management tool for Java language projects.**

Keywords— Schema migration; Database; SQL; DBMS; Postgres; Java; Java library; Agile software development; CI/CD; Evolutionary database design; Database refactoring

I. Introduction

Modern project management methodologies in software engineering, such as Agile, are mostly based on iterative development with small, but regular codebase changes. It is obvious that project codebase and its database schema are interconnected. For example, due to business logic update, you need to add a new attribute to one of your entities. To make your application persist this new attribute, you have to modify your database schema, otherwise the program will not fully persist the entity and most likely the code execution will be terminated with the fatal error. Therefore, code modifications cannot be published prior to a database schema migration.

The developer has to manage both code and schema changes. VCS (version control systems) are used for codebase changes administration. The database schema changes are called "schema migrations" or just "migrations". Migrations cannot be managed just with VCS as far as VCS does not have access to service database and its state.

There is a different set of tools for managing migrations. These tools are either libraries, which are included in your application and migrate database schema on program start up, or standalone programs, which can access the database and execute migrations via CLI (Command Line Interface) manually or automatically in CI/CD (Continuous Integration & Continuous Delivery) pipelines.

The purpose of this work is to develop a new lightweight database migration tool as a library for programs written in Java programming language and databases managed by Postgres DBMS. Similar tools work as follows: SQL [1] expressions are stored in the project codebase and managed by VCS, so every developer can access them and make some changes. These expressions are applied just before a new application version is deployed. The developer can specify the order of migrations and set requirements in each case individually. E.g., some changes can be applied only once or upon every program restart.

This approach supports consistency between the codebase and the database, which make database design and development more transparent and easier to manage. Moreover, there are usually more than one database instance in use on the project. Every developer uses several local databases for development and debugging. There is also a testing database in the testing environment. The benefit of using database schema migration tools is the ability to have all those database schemas mentioned above in the same state. In this way developers can work locally with the current production schema and QAs (quality assurance specialists) are able to test the same application as the one running in production, and they both can use mock data safely and freely without any concerns for production data corruption.

The remainder of the paper is organized as follows. Related work is reviewed in Section 2. In Section 3, the methodology is introduced. Section 4 discusses the expected results. Section 5 concludes the paper.

II. Literature Review

The article of Martin Fowler and Pramod Sadalage [2] on evolutionary database design states that agile [3] project requirements cannot be fixed beforehand, so the detailed design phase of classical software development approaches such as "Waterfall" have no sense. This fact causes the need for regular code and database refactoring [4] and thinking about software design as on-going process. The authors assume that is a basis for evolutionary design approach. This approach requires fast and simple development process, while classical methods of managing a database are difficult and time-consuming. So, the authors came up with helping scripts and tooling for database migration management. These instruments were basically the predecessors of modern migration management frameworks such as Liquibase [5], Flyway [6] and others [7], [8].

Martin Fowler and Pramod Sadalage also stated the base principles of implementing evolutionary database development. All database artifacts such as migration changesets must be version controlled and all database changes must be migrations. DBA (database administrators) and developers must never run DML (data modification language) or DDL (data definition language) expressions manually. All refactoring rules mentioned in [9], [10], [11], [12], [13], [14], [15] must be applicable to every database change. Those changes are backward compatible with codebase and are not damaging, therefore new version deployment is error prone, and the service has close to zero seconds of downtime [16] during schema updates and continuous delivery process. There is a useful side-effect of following those rules. Every team member (developer, QA, DBA etc.) can easily get his own database instance and work with database separately without interfering with others, while those instances are synchronized with current production database schema.

Fowler also mentions [17] that if one integrates his changes with the project mainline every day, then the pain of integration almost vanishes. He proposes to keep database migrations along with corresponding codebase changes as small as possible. He states that the larger a single database change is, the more errors could occur, while small migrations are easier to handle.

As mentioned in [18] the modern information systems tend to change its database schemas regularly due to diversity of stakeholders, rapidly changing environment, and constant flow of new requirements in both enterprise and scientific projects. Authors provide statistics on number of schema modifications in popular open-source information systems. The statistic within the sample shows that average number of schema migrations is over 16 per year, while maximum is more than 42 and even the most stable schemas are migrated at least twice a year. Authors believe that automatization of schema modification process can decrease the risk of data corruption and prevent system downtimes.

There are some requirements mentioned in [19] for a "perfect world" solution of schema migration problem. The authors state that the tool's interface must provide the ability to define atomic schema changes in the incremental and structural way. Also, there must be an interface for overriding any implicit tool configuration, to help a user be in full control over a program behavior. This perfect system must check for information preservation and data redundancy and highlight possibly dangerous or ineffective schema changes and even suggest how to fix them. The system must automatically generate scripts to reverse any applied schema change, so the database schema can be migrated back to a previous version as quick as possible to reduce a system downtime if a fatal system error occurs in production due to program incompatibility with a new schema.

III. Methodology

The proposed program is a tool, which helps to manage database schema migration. The tool is a Java library, which can be used in any Java or JVM [20] project. The library is to be included in Java classpath [21], either manually via its JAR-file [22] or automatically using build automation tools like Gradle [23] or Maven [24]. The library can access and scan all files in project resources folder. That directory must contain a user-defined configuration file and migrations files in special format the library can parse.

Configuration file is a file where user specifies behavior of a tool and some common settings of migrations. There can be only one configuration file in a project. This file must contain database connection credentials such as connection URL [25], a database name, a username, and a password. The configuration file also defines the order of putting migration files into operation.

Migration file is a file where a developer defines migration sets in a form of SQL expressions applicable to Postgres DBMS. Each migration file can keep one or many migrations. The migrations application sequence is defined by their placement order in a migration file. Developer can configure some behavior aspects for each migration. For instance, whether a migration should be reapplied on every migration process, or it should be run just once; or should the migration be transactional [26] or not.

The library stores migration metadata in a changelog table in a project database. This table contains information on every migration – its configuration from migration file, the hash [27] of migration body, timestamp of the last migration execution and other configuration data. Body hash allows library to determine whether a body has been changed since previous execution and depending on the configuration raise an exception [28] or reapply the migration.

Another important aspect of migration management is locking [29]. As far as there are usually multiple instances of application servers, a migration tool can be executed simultaneously on different host machines, but on the same database, so that the migration management tool must ensure there will not be any corrupted data in multithreaded environment. To resolve this problem the library stores a lock in a separate table in a database. The lock is just a single number – one or zero. If lock equals zero, it is guaranteed there are no competitive migration management processes, so the current process must acquire the lock by setting its value to one. Therefore, other processes, if any, will not interrupt the migration. Those processes must wait until the lock is zero again and acquire it themselves.

IV. Expected Results

The anticipated result of the proposed work is a production-ready schema migration management tool in a form of Java library, which can be connected to any Java language project with the help of Maven or Gradle build automation tools or manually using a library JAR-file and jointly used with Postgres database management system. The library must be lightweight in terms of knowledge required to implement migration management in small Java project.

On every program startup, library must connect to project database, detect provided migration files in project sources, then choose changes according to current database state, program environment and migration configuration and apply those changes.

V. Conclusion

Database schema migration management is a usual task for software engineers who deal with databases. Current trends in software development force programmers to support agility both for their code and databases. Programmers require a convenient tool which helps to migrate production database schema and keep databases from testing and development environment up to date with a production database. Software engineers lack a tool, which can ensure both code-data consistency and data safety during dangerous migration process. The proposed work aims to develop this kind of tool.

Word count: 1634 w.

Michael Belkin [illegible]

References

[1] "ISO/IEC 9075-1:2016", *ISO*. [Online]. Available: https://www.iso.org/standard/63555.html. [Accessed: 24- Feb- 2022].

[2] P. Sadalage and M. Fowler, "Evolutionary Database Design", *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/evodb.html. [Accessed: 24- Feb- 2022].

[3] "What is Agile? | Atlassian", *Atlassian*. [Online]. Available: https://www.atlassian.com/agile. [Accessed: 24- Feb- 2022].

[4] M. Fowler, "Refactoring Home Page", *Refactoring.com*. [Online]. Available: https://refactoring.com. [Accessed: 24- Feb- 2022].

[5] "Liquibase Documentation", *Docs.liquibase.com*. [Online]. Available: https://docs.liquibase.com/. [Accessed: 24- Feb- 2022].

[6] "Flyway Migrations", *Flyway*. [Online]. Available: https://flywaydb.org. [Accessed: 24- Feb- 2022].

[7] C. Begin, "mybatis – MyBatis 3 | Introduction", *Mybatis.org*. [Online]. Available: https://mybatis.org/mybatis-3/. [Accessed: 24- Feb- 2022].

[8] "dbdeploy.com", *Db deploy*. [Online]. Available: http://dbdeploy.com. [Accessed: 24- Feb- 2022].

[9] D. Sato, "Parallel Change", *martinfowler.com*. [Online]. Available: https://martinfowler.com/bliki/ParallelChange.html. [Accessed: 24- Feb- 2022].

[10] S. Ambler and P. Sadalage, *Refactoring databases*. Upper Saddle River, N.J.: Addison Wesley, 2006.

[11] M. Fowler, "Incremental Migration", *martinfowler.com*. [Online]. Available: https://martinfowler.com/bliki/IncrementalMigration.html. [Accessed: 24- Feb- 2022].

[12] P. Sadalage, "Refactoring Databases - Introduce New Column", *Databaserefactoring.com*. [Online]. Available: https://databaserefactoring.com/IntroduceNewColumn.html. [Accessed: 24- Feb- 2022].

[13] P. Sadalage, "Refactoring Databases - Make Column Non Nullable", *Databaserefactoring.com*. [Online]. Available: https://databaserefactoring.com/MakeColumnNonNullable.html. [Accessed: 24- Feb- 2022].

[14] P. Sadalage, "Refactoring Databases - Split Table", *Databaserefactoring.com*. [Online]. Available: https://databaserefactoring.com/SplitTable.html. [Accessed: 24- Feb- 2022].

[15] P. Sadalage, "Refactoring Databases - Rename Table", *Databaserefactoring.com*, 2022. [Online]. Available: https://databaserefactoring.com/RenameTable.html. [Accessed: 24- Feb- 2022].

[16] J. Lewis and M. Fowler, "Microservices", *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 24- Feb- 2022].

[17] M. Fowler, "Frequency Reduces Difficulty", *martinfowler.com*. [Online]. Available: https://martinfowler.com/bliki/FrequencyReducesDifficulty.html. [Accessed: 24- Feb- 2022].

[18] Carlo Curino, Hyun Jin Moon, Alin Deutsch, Carlo Zaniolo (2013). Automating the database schema evolution process., 22(1), 73–98. doi:10.1007/s00778-012-0302-x

[19] Curino, Carlo A.; Moon, Hyun J.; Zaniolo, Carlo (2008). Graceful database schema evolution. Proceedings of the VLDB Endowment, 1(1), 761–772.doi:10.14778/1453856.1453939

[20] "Red Hat JVM/JDK Summary - Red Hat Customer Portal", *Red Hat Customer Portal*. [Online]. Available: https://access.redhat.com/articles/2950741. [Accessed: 24- Feb- 2022].

[21] "PATH and CLASSPATH", *docs.oracle.com*. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/environment/paths.html. [Accessed: 24- Feb- 2022].

[22] "JAR File Specification", *docs.oracle.com*. [Online]. Available: https://docs.oracle.com/en/java/javase/13/docs/specs/jar/jar.html. [Accessed: 24- Feb- 2022].

[23] "Gradle Build Tool", *Gradle*. [Online]. Available: https://gradle.org. [Accessed: 24- Feb- 2022].

[24] B. Porter, J. Zyl and O. Lamy, "Maven – Welcome to Apache Maven", *Maven.apache.org*. [Online]. Available: https://maven.apache.org. [Accessed: 24- Feb- 2022].

[25] "Connecting to the Database", *jdbc.postgresql.org*. [Online]. Available: https://jdbc.postgresql.org/documentation/80/connect.html. [Accessed: 24- Feb- 2022].

[26] "Transactions", *PostgreSQL Documentation*, 2022. [Online]. Available: https://www.postgresql.org/docs/8.3/tutorial-transactions.html. [Accessed: 24- Feb- 2022].

[27] Knuth, D. 1973, The Art of Computer Programming, Vol. 3, Sorting and Searching, p.527. Addison-Wesley, Reading, MA., United States

[28] "Exceptions (Java Platform)", *docs.oracle.com*, 2022. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html. [Accessed: 24- Feb- 2022].

[29] Peyton Jones, Simon (2007). “Beautiful Concurrency”. In Wilson, Greg; Oram, Andy (eds.). *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly.