International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Automating Database Management Using Schema Verification and Version Control for Continuous Integration and Deployment: A Plugin Based Approach

Gautham Ram Rajendiran
gautham.rajendiran@icloud.com

Abstract
Automated Schema Deployment and Version Control is a critical aspect of Data Management pipelines in modern data applications . Automating this process can save considerable time, reduce errors, and streamline database migrations across environments. This paper presents the advantages of a centralized plugin-based schema version control and outlines the advantages of having such a setup. It also explores the implementation of such a plugin by using Liquibase and ANT build system.

Keywords: Continuous Integration And Deployment, Data Engineering, Schema Version Control, Database Management

Introduction

Database schema management is one of the most critical components of modern application development [1]. As applications grow and multiple teams contribute to the same codebase which makes managing schema changes manually to be increasingly challenging. Without a reliable and centralized mechanism, these schema changes can cause inconsistencies. Such inconsistencies can lead to production errors, and create confusion across environments. For example, if the database is replicated across multiple geographical regions, then manual checks are needed to make sure all regions use the same schema. This can introduce potential for errors in schema names, mismatch in column data types [2] and potentially different ETL [3] processes for every region. Maintaining regions specific ETL processes is a waste of time and effort. Automating schema control and verification ensures a more efficient development process and reduces the risk of failures when deploying applications.

Centralized Schema Control Mechanisms

A centralized schema control mechanism refers to an automated system that manages all database schema changes from a single source of truth. It allows for consistent updates across different environments (development, staging, production) and offers several advantages:

Consistency Across Environments: A centralized schema control mechanism ensures that schema changes are applied uniformly across environments, thereby minimizing the chance of errors caused by varying schema versions across regions and environments.

Reduced Manual Intervention: Developers do not need to manually apply schema changes to each environment. This saves time taken to deploy schema changes and also reduces the potential for human

error.

Automated Verification: Centralized control allows for automated checks to be run against the schema changes before they are applied to the production database. This ensures that migrations are thoroughly tested in a safe environment which reduces the risk of surprises due to failed migrations in production.

Faster Development Cycles: By automating schema updates, development teams can focus on writing code rather than worrying about manual database maintenance tasks which speeds up the overall development process.

Improved Collaboration: Centralized schema management allows multiple developers to contribute schema changes simultaneously, knowing that these changes will be consistently applied and verified. This makes collaboration smoother and avoids conflicts. For example, if two developers try to modify the same column name to be something different, then code version control software like git [4] will force code owners to consolidate their changes before merging. The automated schema control plugin will verify the changes done as a result of conflict resolution [4] against the migrations and will only allow merge to production after thorough sanity checks against the entire set of migrations for that database.

The Role of Plugins in Database Management

Many teams rely on plugins that integrate with build tools to automate many deployment processes like cleaning up the code base [5], running tests and other sanity checks. The idea behind this paper is to use such a plugin based approach to automate the process of schema version control and verification of data applications. As such, the ideal plugin will own the following responsibilities:

- Applying schema changes (migrations) from one central source.
- Verifying that these changes are valid by running them in controlled environments (e.g., in-memory databases).
- Failing the build process if a migration causes an issue.

Plugins are particularly useful in continuous integration (CI) environments, where schema changes are tested automatically every time code is pushed, ensuring a stable and reliable deployment pipeline.

Liquibase and ANT as Implementation Tools

While centralized schema control mechanisms offer broad advantages, their implementation can vary depending on the tools and technologies in use. This paper presents a solution for implementing centralized schema control in Java applications using Liquibase [6] and ANT [7]. Together, these tools form an automated database management system that can run and verify schema changes across environments.

Liquibase Overview

Liquibase is a database migration tool that simplifies schema change management by tracking, applying, and rolling back schema changes in a consistent, version-controlled manner. It stores these changes in a "changelog" that can be written in formats like XML, JSON or SQL.

Outlined below are some of the features provided by Liquibase:

- **Multi-Database Support**: Liquibase is compatible with a wide variety of databases. This makes it versatile and can be used to version control any kind of database management system or data warehouse.

- **Schema Versioning**: Every schema change is version-controlled, ensuring that all environments are synchronized with the correct schema version.
- **Flexible Change Formats**: Developers can define schema changes using SQL, database-agnostic XML, YAML, or JSON.
- **Rollback Capabilities**: Liquibase allows for easy rollback of changes, reducing the risk of errors in production.
- **Change Verification**: It provides built-in mechanisms for checking which changes have been applied and allows for automated verification of schema changes.

ANT Build Tool Overview

ANT is a Java-based build tool commonly used to automate repetitive tasks in software development, such as compiling code, running tests, and deploying applications. By integrating Liquibase with ANT, developers can automate database migrations as part of the build process. This integration allows the team to run schema migrations during every build and automatically verify the changes using an in-memory database like Hyperbase.

The integration provides several key benefits:

- **Task Automation**: Developers can automate schema changes and verification with a simple command in the ANT build file.
- **In-Memory Verification**: The integration enables migrations to run in an in-memory database, allowing for automatic verification before deploying changes to the actual production database.
- **Seamless CI Integration**: The ANT-Liquibase integration fits seamlessly into CI pipelines, ensuring schema consistency across all environments.

The ANT Plugin for Automating Database Migrations

This paper proposes the development of an ANT plugin that automates the management of database schema changes using Liquibase. The plugin simplifies the migration process by providing a single, centralized mechanism to manage schema changes and ensure their correctness.

Key Features of the Plugin

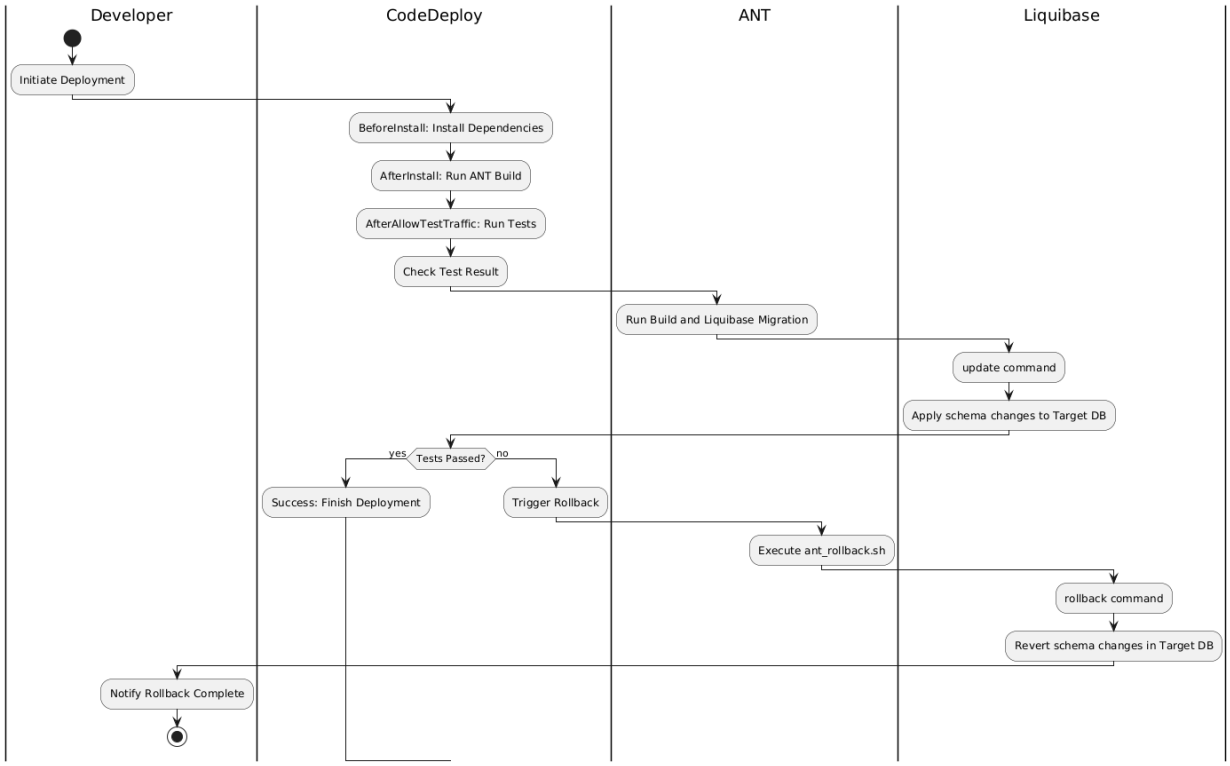
- **Centralized Schema Control**: The plugin ensures that schema migrations are applied from a central directory containing all migration scripts.
- **Verification with In-Memory Databases**: To guarantee that migrations are valid before they are applied to production, the plugin automatically spins up an in-memory database, such as Hyperbase or H2, and runs the migrations in this isolated environment. If any migration fails, the build process will halt, preventing bad changes from reaching production.
- **SQL-Based Migrations**: Since migration scripts can contain SQL queries (e.g., CREATE, UPDATE, DELETE statements), the plugin can be used for both transactional databases and data warehousing environments.
- **CI-Friendly**: The plugin integrates smoothly with CI/CD pipelines, enabling schema changes to be automatically applied and verified as part of the build process.

How It Works

1. **Define Migrations**: Developers create schema migration files in a central directory. These files may contain SQL commands or Liquibase-specific changelogs. Every database has its own root or folder for security reasons.
2. **Run Migrations in Development**: When the plugin is triggered (typically as part of the build process), it will apply the migrations to an in-memory database (for verification).
3. **Verify Migrations**: If the migrations succeed in the in-memory database, the build process continues. If there are errors, the build fails, providing early feedback to the developer about potential issues with the schema change.
4. **Apply to Production**: After verifying that the migrations work, the same plugin is used to apply the changes to the production database during deployment.

Code Implementation: A CI Pipeline using AWS Code Deploy, Liquibase and ANT build

In the context of a Java application, using AWS CodeDeploy in combination with Liquibase and ANT provides a robust mechanism to automate deployments and rollbacks. The approach ensures that schema changes are verified before being applied and that, if something goes wrong during the deployment, Liquibase can automatically revert the database changes to a previous state.



Implementation Details

AWS CodeDeploy Initiates Deployment:

AWS CodeDeploy starts by triggering the ANT build file that applies the database migrations using Liquibase as part of the build.

Run Tests and Validate:

After the schema changes are applied, automated tests are run to ensure that both the application and the database are functioning as expected.

Apply Database Migrations:
The update command in Liquibase is run through the ANT script to apply the necessary schema changes to the target database.
Rollback on Failure:
If any test fails or if there is an error during the deployment process, AWS CodeDeploy triggers a rollback. The rollback is handled by Liquibase using the rollback SQL defined in the changeSets.
When using Liquibase for migrations, it is possible to define rollback SQL directly in the changeSet. This rollback SQL ensures that Liquibase knows how to revert the database changes in case of failure.
Example of Liquibase ChangeSet with Rollback SQL:

```
<changeSet id="2020-09-23-1" author="developer">
        <createTable tableName="orders">
                <column name="order_id" type="int" autoIncrement="true">
                        <constraints primaryKey="true"/>
                </column>
                <column name="customer_id" type="int"/>
                        <column name="order_date" type="timestamp"/>
        </createTable>
        <!-- Define rollback SQL -->
        <rollback>
                <dropTable tableName="orders"/>
        </rollback>
</changeSet>
```

In this example, if the deployment fails or a rollback is triggered, Liquibase will drop the orders table to revert the schema to its previous state.

ANT Task for executing schema verification with Liquibase
The following class implements the ANT task for verifying migrations with Liquibase build, it follows the builder pattern to create the Liquibase runner process.

```
package com.amazon.liquibaseBuild.task;
import com.amazon.liquibaseBuild.process.db.DatabaseRunner;
import com.amazon.liquibaseBuild.process.liquibase.LiquibaseRunner;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Project;
import org.apache.tools.ant.Task;
import java.util.Optional;
public class ValidateSchemaTask extends Task {
   private String changeLogRoot;
   private String changeLogPath;
   public void setChangeLogPath(String changeLogPath) {
      this.changeLogPath = changeLogPath;
   }
   public void setChangeLogRoot(String changeLogRoot) {this.changeLogRoot = changeLogRoot; }
```

```
    public void execute() throws BuildException {
        this.log("Obtained changelog path " + this.changeLogPath, Project.MSG_INFO);
        DatabaseRunner dbRunner = new DatabaseRunner();
        try {

            dbRunner.run();

            Optional<LiquibaseRunner> liqRunner = LiquibaseRunner.builder()
                .withChangelogPath(this.changeLogPath)
                .withChangelogRoot(this.changeLogRoot)
                .withConnection(dbRunner.getConnection())
                .build();

            if (liqRunner.isPresent()) {
                log("Liquibase client started successfully, validating migrations", Project.MSG_INFO);
                liqRunner.get().run();
            } else {
                log("Unable to start liquibase client", Project.MSG_ERR);
                throw new Exception("Unable to start liquibase client");
            }

            log("Migrations validated, no errors found.", Project.MSG_INFO);

            dbRunner.close();
        } catch (Exception e) {
            log(e.getMessage(), Project.MSG_ERR);
            throw new BuildException("Build Exception: ", e.getMessage());
        } finally {
            try {
                dbRunner.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

}
```

This is the corresponding LiquibaseRunner implementation

```
package com.amazon.liquibaseBuild.process.liquibase;
```

```
import liquibase.Contexts;
import liquibase.LabelExpression;
import liquibase.Liquibase;
import liquibase.database.Database;
import liquibase.database.DatabaseFactory;
import liquibase.database.jvm.JdbcConnection;
import liquibase.exception.DatabaseException;
import liquibase.logging.LogLevel;
import liquibase.logging.Logger;
import liquibase.resource.FileSystemResourceAccessor;

import java.io.Closeable;
import java.io.IOException;
import java.sql.Connection;
import java.util.Objects;
import java.util.Optional;

public class LiquibaseRunner implements Closeable {

    private final String changelogPath;
    private final String changelogRoot;
    private final Database database;
    private Liquibase liquibase;

    private LiquibaseRunner(
        String changelogRoot,
        String changelogPath,
        Database database
    ) {
        this.changelogPath = changelogPath;
        this.changelogRoot = changelogRoot;
        this.database = database;
    }

    public static LiquibaseRunner.Builder builder() {
        return new Builder();
    }

    public void run() throws Exception {
        liquibase = new Liquibase(
            this.changelogPath,
            new FileSystemResourceAccessor(this.changelogRoot),
            this.database
```

```
    );
    Logger log = liquibase.getLog();
    log.setLogLevel(LogLevel.SEVERE);
    liquibase.updateTestingRollback(new Contexts(), new LabelExpression());
}

@Override
public void close() throws IOException {
    try {
        liquibase.getDatabase().close();
    } catch (DatabaseException e) {
        throw new IOException("Unable to close database connection", e);
    }
}

public static class Builder {

    private String changelogPath;
    private String changelogRoot;
    private Connection connection;

    public Builder() {
    }

    public Optional<LiquibaseRunner> build() {
        Objects.requireNonNull(this.changelogPath);
        Objects.requireNonNull(this.connection);

        Database db;
        try {
            db = getDatabase();
        } catch (Exception e) {
            return Optional.empty();
        }
        return Optional.of(new LiquibaseRunner(
            getChangelogRoot(),
            getChangelogPath(),
            db
        ));
    }

    private String getChangelogPath() {
        return this.changelogPath;
```

```
    }

    private String getChangelogRoot() {
      return this.changelogRoot;
    }

    private Database getDatabase() throws Exception {
      return DatabaseFactory.getInstance().findCorrectDatabaseImplementation(
        new JdbcConnection(this.connection)
      );
    }

    public Builder withChangelogPath(String changelogPath) {
      this.changelogPath = changelogPath;
      return this;
    }

    public Builder withChangelogRoot(String changelogRoot) {
      this.changelogRoot = changelogRoot;
      return this;
    }

    public Builder withConnection(Connection connection) {
      this.connection = connection;
      return this;
    }

  }
}
```

Creating database specific changelogs

The following Liquibase changelog that was created during implementation contains all changes for one database called “ics” as shown below. Multiple such changelogs can be created for every database owned by the application.

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
     schema/dbchangelog-4.6.xsd">
  <include file="liquibase/migrations/ics/1970-01-01_add_inventory_tables.xml"/>
  <include file="liquibase/migrations/ics/2020-06-23_add_lotnumber_to_pcsku_records.xml"/>
  <include file="liquibase/migrations/ics/2020-08-01_add_outboundsku_to_inventory_items.xml"/>
```

```
  <include file="liquibase/migrations/ics/2020-08-10_add_inventory_container_attributes_for_mcd.xml"/>
  <include file="liquibase/migrations/ics/2020-08-10_add_inventory_item_attributes_for_mcd.xml"/>
  <include file="liquibase/migrations/ics/2020-08-10_add_inventory_item_history_attributes_for_mcd.xml"/>
  <include file="liquibase/migrations/ics/2020-08-10_add_inventory_reservation_attributes_for_mcd.xml"/>
  <include file="liquibase/migrations/ics/2020-09-02_add_inventory_location_records.xml"/>
  <include file="liquibase/migrations/ics/2020-09-02_add_sort_area_sort_location.xml"/>
  <include file="liquibase/migrations/ics/2020-02-29_add_original_container_id.xml"/>
  <include file="liquibase/migrations/ics/2020-06-17_add_physical_condition_to_inventory_items.xml"/>
  <include file="liquibase/migrations/ics/2020-06-17_add_inventory_items_importability.xml"/>
</databaseChangeLog>
```

ANT Build Script to Apply Migrations

The ANT build script, build.xml, will be responsible for applying the Liquibase migrations as part of the deployment. The "deploy" target executes migrations against the target database, and the "build-ics-schema" task runs the schema verification ANT task defined in previous sections. A CI pipeline like Git Flow [8] or AWS Code Deploy [9] can be configured to run the build task, which validates the migrations prior to applying the migrations in production.

```
<project name="DatabaseMigration" default="deploy" basedir=".">
    <property name="liquibase.driver" value="com.mysql.cj.jdbc.Driver"/>
    <property name="liquibase.url" value="jdbc:mysql://localhost:3306/your_db"/>
    <property name="liquibase.username" value="root"/>
    <property name="liquibase.password" value="password"/>
    <property name="liquibase.changeLogFile" value="src/main/resources/db/changelog/db.changelog-master.xml"/>
    <target name="deploy">
        <!-- Apply Liquibase migrations -->
        <taskdef name="liquibase" classname="liquibase.integration.ant.LiquibaseTask" classpathref="classpath"/>
            <liquibase
            driver="${liquibase.driver}"
            url="${liquibase.url}"
            username="${liquibase.username}"
            password="${liquibase.password}"
            changeLogFile="${liquibase.changeLogFile}"
            logLevel="info"
            update="true"/>
    </target>

    <target name="build-ics-schema">
```

```
        <liquibaseBuild changeLogRoot="${package_root}/.."
changeLogPath="liquibase/ics_beta_changelog.xml" />
    </target>
</project>
```

Result

This system of schema version control and migration was implemented for a data warehouse in an eCommerce application. The application spanned across 4 regions (North America, Europe, China and Japan), hence the number of changes needed to create schema changes was cut by 4 times. This also resolved existing inconsistencies in index creation, column names and data types. A challenge to implementing this solution was to migrate existing inconsistent schemas to the new version controlled schema. Mechanisms such as schema evolution [10] were used to handle such situations, future research can focus on incorporating robust schema evolution mechanisms as a part of the plugin or a separate plugin entirely, which brings in a foolproof change management system in the context of database schemas.

Conclusion

This proposed solution integrates Liquibase's rollback feature with AWS CodeDeploy and ANT to automate both application and database deployments. By managing schema changes with Liquibase and triggering rollbacks via ANT, one can ensure that deployments are safe, consistent, and easily reversible. AWS CodeDeploy handles the orchestration of the entire process, providing a robust and flexible deployment pipeline with minimal downtime and risk. This setup is well-suited for teams working on large-scale data platforms that require consistent database schema management and version control.

References

1. Zhao, Xiaohui, and Chengfei Liu. "Version Management for Business Process Schema Evolution." Information Systems, vol. 38, no. 8, Nov. 2013, pp. 1046–69, doi:10.1016/j.is.2013.03.006.
2. A. Baqasah and E. Pardede, "Managing schema evolution in hybrid XML-relational database systems," 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, Perth, WA, Australia, 2010, pp. 455-460, doi: 10.1109/WAINA.2010.127.
3. Q. Hanlin, J. Xianzhen and Z. Xianrong, "Research on Extract, Transform and Load(ETL) in Land and Resources Star Schema Data Warehouse," 2012 Fifth International Symposium on Computational Intelligence and Design, Hangzhou, China, 2012, pp. 120-123, doi: 10.1109/ISCID.2012.38.
4. D. Spinellis, "Git," IEEE Software, vol. 29, no. 3, pp. 100-101, May-June 2012, doi: 10.1109/MS.2012.61.
5. "Checkstyle – Checkstyle 10.18.1." Available: https://checkstyle.sourceforge.io/.
6. "Liquibase." Available: https://www.liquibase.com/.
7. "Apache Ant." Available: https://ant.apache.org/.
8. "Comparing workflows: GitFlow workflow," Atlassian Git Tutorials. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.
9. "AWS CodeDeploy User Guide." Available: https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html.

10. Curino, Carlo, et al. "Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System." Proceedings of the VLDB Endowment, vol. 4, no. 2, Nov. 2010, pp. 117–28, doi:10.14778/1921071.1921078.