Comprehensive Flyway Rollback Framework for Spring Boot

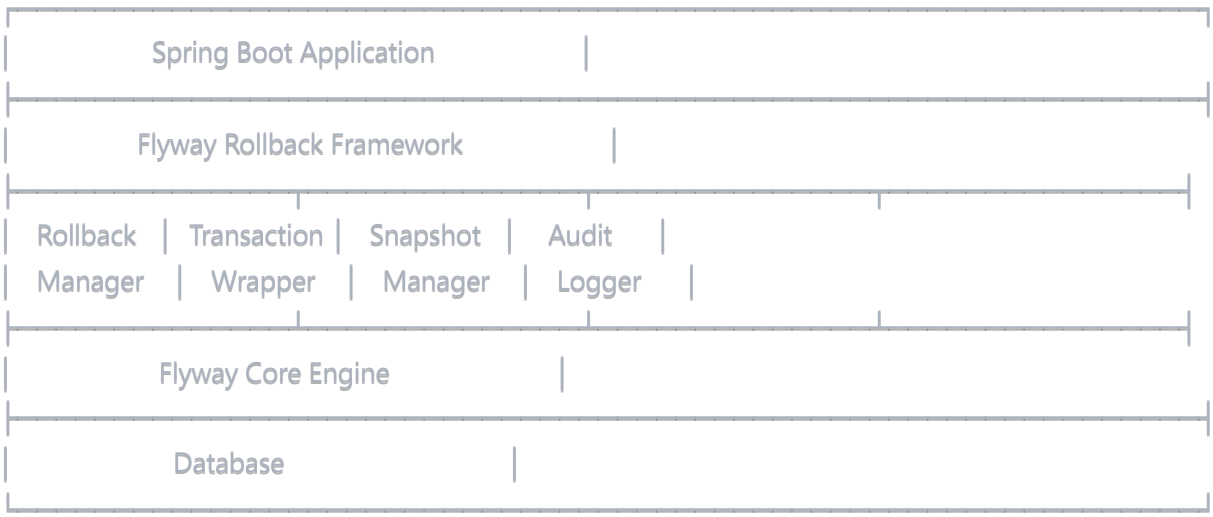
Executive Summary

Flyway's native rollback capabilities are limited, especially in the Community Edition. This framework provides a production-ready solution that extends Flyway with comprehensive rollback support for all SQL operation types (DDL, DML, DQL, DCL, TCL) while maintaining data integrity in production environments.

Key Challenges with Flyway Rollback

1. **Undo Migrations** are only available in Flyway Pro/Enterprise editions
2. **No automatic rollback** for failed migrations in databases without DDL transaction support (MySQL, MariaDB)
3. **Data loss risks** when rolling back DML operations
4. **No built-in support** for complex rollback scenarios involving data dependencies

Architecture Overview



Implementation Components

1. Core Framework Structure

java

```
// Maven Dependencies
<dependencies>
    <!-- Spring Boot -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Flyway -->
    <dependency>
        <groupId>org.flywaydb</groupId>
        <artifactId>flyway-core</artifactId>
    </dependency>

    <!-- Database-specific Flyway support -->
    <dependency>
        <groupId>org.flywaydb</groupId>
        <artifactId>flyway-mysql</artifactId>
    </dependency>

    <!-- Additional dependencies for rollback framework -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
    </dependency>

    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
    </dependency>
</dependencies>
```

2. Rollback Manager Core

java

```java
@Component
@Slf4j
public class FlywayRollbackManager {

    @Autowired
    private DataSource dataSource;

    @Autowired
    private SnapshotManager snapshotManager;

    @Autowired
    private RollbackScriptGenerator scriptGenerator;

    @Autowired
    private AuditLogger auditLogger;

    @Value("${flyway.rollback.enabled:true}")
    private boolean rollbackEnabled;

    @Value("${flyway.rollback.snapshot.enabled:true}")
    private boolean snapshotEnabled;

    public RollbackResult rollbackToVersion(String targetVersion) {
        log.info("Initiating rollback to version: {}", targetVersion);

        try {
            // 1. Validate target version
            validateTargetVersion(targetVersion);

            // 2. Create safety snapshot
            String snapshotId = null;
            if (snapshotEnabled) {
                snapshotId = snapshotManager.createSnapshot();
            }

            // 3. Get rollback plan
            RollbackPlan plan = createRollbackPlan(targetVersion);

            // 4. Execute rollback
            executeRollback(plan);

            // 5. Verify rollback
            verifyRollback(targetVersion);

            // 6. Audit log
            auditLogger.logRollback(targetVersion, plan, "SUCCESS");
```

```
            return RollbackResult.success(targetVersion, snapshotId);

        } catch (Exception e) {
            log.error("Rollback failed", e);
            auditLogger.logRollback(targetVersion, null, "FAILED: " + e.getMessage());
            throw new RollbackException("Rollback to version " + targetVersion + " failed", e);
        }
    }

    private RollbackPlan createRollbackPlan(String targetVersion) {
        List<AppliedMigration> migrationsToRollback = getMigrationsToRollback(targetVersion);
        RollbackPlan plan = new RollbackPlan();

        for (AppliedMigration migration : migrationsToRollback) {
            RollbackScript script = scriptGenerator.generateRollbackScript(migration);
            plan.addScript(script);
        }

        return plan;
    }
}
```

3. Rollback Script Generator

java

```java
@Component
@Slf4j
public class RollbackScriptGenerator {

    @Autowired
    private DDLRollbackHandler ddlHandler;

    @Autowired
    private DMLRollbackHandler dmlHandler;

    @Autowired
    private DCLRollbackHandler dclHandler;

    public RollbackScript generateRollbackScript(AppliedMigration migration) {
        String originalScript = migration.getScript();
        SqlType sqlType = determineSqlType(originalScript);

        RollbackScript rollbackScript = new RollbackScript();
        rollbackScript.setVersion(migration.getVersion());
        rollbackScript.setDescription("Rollback of: " + migration.getDescription());

        switch (sqlType) {
            case DDL:
                rollbackScript.setSql(ddlHandler.generateRollback(originalScript));
                break;
            case DML:
                rollbackScript.setSql(dmlHandler.generateRollback(originalScript, migration));
                break;
            case DCL:
                rollbackScript.setSql(dclHandler.generateRollback(originalScript));
                break;
            default:
                throw new UnsupportedOperationException("Rollback not supported for: " + sqlType);
        }

        return rollbackScript;
    }
}
```

4. DDL Rollback Handler

java

```
@Component
@Slf4j
public class DDLRollbackHandler {

    @Autowired
    private DatabaseMetadataService metadataService;

    public String generateRollback(String ddlScript) {
        DDLParser parser = new DDLParser();
        DDLStatement statement = parser.parse(ddlScript);

        StringBuilder rollback = new StringBuilder();

        switch (statement.getType()) {
            case CREATE_TABLE:
                rollback.append(generateDropTable(statement));
                break;

            case ALTER_TABLE:
                rollback.append(generateAlterTableRollback(statement));
                break;

            case DROP_TABLE:
                rollback.append(generateCreateTableFromMetadata(statement));
                break;

            case CREATE_INDEX:
                rollback.append(generateDropIndex(statement));
                break;

            case ADD_CONSTRAINT:
                rollback.append(generateDropConstraint(statement));
                break;
        }

        return rollback.toString();
    }

    private String generateAlterTableRollback(DDLStatement statement) {
        String tableName = statement.getTableName();
        TableMetadata currentMetadata = metadataService.getTableMetadata(tableName);

        StringBuilder rollback = new StringBuilder();

        for (DDLOperation operation : statement.getOperations()) {
            switch (operation.getType()) {
```

```
            case ADD_COLUMN:
                rollback.append(String.format("ALTER TABLE %s DROP COLUMN %s;\n",
                    tableName, operation.getColumnName()));
                break;

            case DROP_COLUMN:
                ColumnMetadata droppedColumn = metadataService.getDroppedColumnMetadata(
                    tableName, operation.getColumnName());
                rollback.append(String.format("ALTER TABLE %s ADD COLUMN %s %s;\n",
                    tableName, droppedColumn.getName(), droppedColumn.getDefinition()));
                break;

            case MODIFY_COLUMN:
                ColumnMetadata originalColumn = metadataService.getOriginalColumnMetadata(
                    tableName, operation.getColumnName());
                rollback.append(String.format("ALTER TABLE %s MODIFY COLUMN %s %s;\n",
                    tableName, originalColumn.getName(), originalColumn.getDefinition()));
                break;
        }
    }

    return rollback.toString();
  }
}
```

5. DML Rollback Handler with Data Preservation

java

```
@Component
@Slf4j
public class DMLRollbackHandler {

    @Autowired
    private DataSnapshotService dataSnapshotService;

    @Autowired
    private DataSourceProperties dataSourceProperties;

    public String generateRollback(String dmlScript, AppliedMigration migration) {
        DMLParser parser = new DMLParser();
        DMLStatement statement = parser.parse(dmlScript);

        StringBuilder rollback = new StringBuilder();
        rollback.append("-- DML Rollback for version ").append(migration.getVersion()).append("\n");
        rollback.append("-- Generated at: ").append(LocalDateTime.now()).append("\n\n");

        switch (statement.getType()) {
            case INSERT:
                rollback.append(generateDeleteForInsert(statement, migration));
                break;

            case UPDATE:
                rollback.append(generateReverseUpdate(statement, migration));
                break;

            case DELETE:
                rollback.append(generateInsertForDelete(statement, migration));
                break;
        }

        return rollback.toString();
    }

    private String generateReverseUpdate(DMLStatement statement, AppliedMigration migration) {
        String tableName = statement.getTableName();

        // Retrieve snapshot data before the update
        DataSnapshot snapshot = dataSnapshotService.getSnapshot(
            tableName, migration.getInstalledOn());

        if (snapshot == null) {
            log.warn("No snapshot found for table {} at migration {}",
                tableName, migration.getVersion());
            return generateFallbackReverseUpdate(statement);
```

```
    }

    StringBuilder rollback = new StringBuilder();
    rollback.append("-- Reverse UPDATE based on snapshot data\n");

    for (SnapshotRow row : snapshot.getRows()) {
        rollback.append("UPDATE ").append(tableName).append(" SET ");

        List<String> setClauses = new ArrayList<>();
        for (Map.Entry<String, Object> column : row.getColumns().entrySet()) {
            setClauses.add(String.format("%s = %s",
                column.getKey(),
                formatValue(column.getValue())));
        }

        rollback.append(String.join(", ", setClauses));
        rollback.append(" WHERE ");
        rollback.append(buildWhereClause(row.getPrimaryKey()));
        rollback.append(";\n");
    }

    return rollback.toString();
}

private String generateInsertForDelete(DMLStatement statement, AppliedMigration migration) {
    String tableName = statement.getTableName();

    // Retrieve deleted data from audit log
    List<DeletedRow> deletedRows = dataSnapshotService.getDeletedRows(
        tableName, migration.getInstalledOn());

    StringBuilder rollback = new StringBuilder();
    rollback.append("-- Restore deleted data\n");

    for (DeletedRow row : deletedRows) {
        rollback.append("INSERT INTO ").append(tableName).append(" (");
        rollback.append(String.join(", ", row.getColumns().keySet()));
        rollback.append(") VALUES (");
        rollback.append(row.getColumns().values().stream()
            .map(this::formatValue)
            .collect(Collectors.joining(", ")));
        rollback.append(");\n");
    }

    return rollback.toString();
```

```
    }
}
```

6. Transaction-Safe Rollback Executor

java

```
@Component
@Slf4j
public class TransactionSafeRollbackExecutor {

    @Autowired
    private DataSource dataSource;

    @Autowired
    private TransactionTemplate transactionTemplate;

    @Value("${flyway.rollback.batch.size:1000}")
    private int batchSize;

    public void executeRollback(RollbackPlan plan) {
        log.info("Executing rollback plan with {} scripts", plan.getScripts().size());

        for (RollbackScript script : plan.getScripts()) {
            executeScript(script);
        }
    }

    private void executeScript(RollbackScript script) {
        if (isDDL(script)) {
            executeDDLScript(script);
        } else {
            executeDMLScript(script);
        }
    }

    private void executeDDLScript(RollbackScript script) {
        try (Connection connection = dataSource.getConnection()) {
            boolean originalAutoCommit = connection.getAutoCommit();

            try {
                // Some databases support DDL transactions
                if (supportsDDLTransactions(connection)) {
                    connection.setAutoCommit(false);
                    try (Statement stmt = connection.createStatement()) {
                        stmt.execute(script.getSql());
                        connection.commit();
                    } catch (SQLException e) {
                        connection.rollback();
                        throw e;
                    }
                } else {
                    // For MySQL/MariaDB - execute without transaction
```

```
                connection.setAutoCommit(true);
                try (Statement stmt = connection.createStatement()) {
                    stmt.execute(script.getSql());
                }
            }
        } finally {
            connection.setAutoCommit(originalAutoCommit);
        }

        log.info("Successfully executed DDL rollback for version: {}", script.getVersion());

    } catch (SQLException e) {
        throw new RollbackException("Failed to execute DDL rollback", e);
    }
}

private void executeDMLScript(RollbackScript script) {
    transactionTemplate.execute(status -> {
        try (Connection connection = dataSource.getConnection()) {
            connection.setAutoCommit(false);

            String[] statements = script.getSql().split(";");

            for (int i = 0; i < statements.length; i += batchSize) {
                try (Statement stmt = connection.createStatement()) {
                    for (int j = i; j < Math.min(i + batchSize, statements.length); j++) {
                        String sql = statements[j].trim();
                        if (!sql.isEmpty()) {
                            stmt.addBatch(sql);
                        }
                    }
                    stmt.executeBatch();
                }
            }

            log.info("Successfully executed DML rollback for version: {}", script.getVersion());
            return null;

        } catch (SQLException e) {
            status.setRollbackOnly();
            throw new RollbackException("Failed to execute DML rollback", e);
        }
    });
}
}
```

7. Snapshot Manager for Production Data Safety

java

```java
@Component
@Slf4j
public class SnapshotManager {

    @Autowired
    private DataSource dataSource;

    @Value("${flyway.rollback.snapshot.retention.days:7}")
    private int snapshotRetentionDays;

    @Value("${flyway.rollback.snapshot.storage.path:/var/flyway/snapshots}")
    private String snapshotStoragePath;

    public String createSnapshot() {
        String snapshotId = generateSnapshotId();
        log.info("Creating snapshot: {}", snapshotId);

        try {
            List<String> tables = getAllTables();
            SnapshotMetadata metadata = new SnapshotMetadata(snapshotId, tables);

            for (String table : tables) {
                createTableSnapshot(snapshotId, table);
            }

            saveSnapshotMetadata(metadata);

            log.info("Snapshot {} created successfully", snapshotId);
            return snapshotId;

        } catch (Exception e) {
            log.error("Failed to create snapshot", e);
            throw new SnapshotException("Snapshot creation failed", e);
        }
    }

    private void createTableSnapshot(String snapshotId, String tableName) {
        String snapshotTableName = String.format("snapshot_%s_%s", snapshotId, tableName);

        try (Connection connection = dataSource.getConnection()) {
            // Create snapshot table
            String createTableSql = String.format(
                "CREATE TABLE %s AS SELECT * FROM %s",
                snapshotTableName, tableName);

            try (Statement stmt = connection.createStatement()) {
```

```
            stmt.execute(createTableSql);
        }

        // Add metadata
        String metadataSql = String.format(
            "ALTER TABLE %s COMMENT = 'Snapshot of %s created at %s'",
            snapshotTableName, tableName, LocalDateTime.now());

        try (Statement stmt = connection.createStatement()) {
            stmt.execute(metadataSql);
        }

    } catch (SQLException e) {
        throw new SnapshotException("Failed to create table snapshot", e);
    }
}

public void restoreFromSnapshot(String snapshotId) {
    log.info("Restoring from snapshot: {}", snapshotId);

    SnapshotMetadata metadata = loadSnapshotMetadata(snapshotId);

    for (String table : metadata.getTables()) {
        restoreTableFromSnapshot(snapshotId, table);
    }

    log.info("Snapshot {} restored successfully", snapshotId);
}

@Scheduled(cron = "0 0 2 * * ?") // Run at 2 AM daily
public void cleanupOldSnapshots() {
    LocalDateTime cutoffDate = LocalDateTime.now().minusDays(snapshotRetentionDays);
    log.info("Cleaning up snapshots older than {}", cutoffDate);

    List<SnapshotMetadata> snapshots = getAllSnapshots();

    for (SnapshotMetadata snapshot : snapshots) {
        if (snapshot.getCreatedAt().isBefore(cutoffDate)) {
            deleteSnapshot(snapshot.getId());
        }
    }
}
}
```

8. Spring Boot Configuration

yaml

```yaml
# application.yml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: password

  flyway:
    enabled: true
    locations: classpath:db/migration
    baseline-on-migrate: true
    validate-on-migrate: true

flyway:
  rollback:
    enabled: true
    snapshot:
      enabled: true
      retention:
        days: 7
      storage:
        path: /var/flyway/snapshots
    audit:
      enabled: true
      table: flyway_rollback_audit
    batch:
      size: 1000
    safety:
      require-approval: true
      dry-run: true
```

9. Migration Naming Convention for Rollback Support

```
db/migration/
├── V1.0__initial_schema.sql
├── U1.0__rollback_initial_schema.sql
├── V1.1__add_user_table.sql
├── U1.1__rollback_add_user_table.sql
├── V1.2__update_user_data.sql
├── U1.2__rollback_update_user_data.sql
└── snapshots/
    ├── V1.0__snapshot_metadata.json
    ├── V1.1__snapshot_metadata.json
    └── V1.2__snapshot_metadata.json
```

10. REST API for Rollback Management

java

```java
@RestController
@RequestMapping("/api/flyway")
@Slf4j
public class FlywayRollbackController {

    @Autowired
    private FlywayRollbackManager rollbackManager;

    @Autowired
    private RollbackApprovalService approvalService;

    @PostMapping("/rollback")
    public ResponseEntity<RollbackResult> rollback(@RequestBody RollbackRequest request) {
        log.info("Rollback request received: {}", request);

        // Validate request
        validateRollbackRequest(request);

        // Check approval if required
        if (isApprovalRequired()) {
            ApprovalStatus approval = approvalService.checkApproval(request);
            if (!approval.isApproved()) {
                return ResponseEntity.status(HttpStatus.FORBIDDEN)
                    .body(RollbackResult.needsApproval(approval.getApprovalId()));
            }
        }

        // Execute rollback
        RollbackResult result = rollbackManager.rollbackToVersion(request.getTargetVersion());

        return ResponseEntity.ok(result);
    }

    @GetMapping("/rollback/plan/{targetVersion}")
    public ResponseEntity<RollbackPlan> getRollbackPlan(@PathVariable String targetVersion) {
        RollbackPlan plan = rollbackManager.createRollbackPlan(targetVersion);
        return ResponseEntity.ok(plan);
    }

    @PostMapping("/rollback/dry-run")
    public ResponseEntity<DryRunResult> dryRun(@RequestBody RollbackRequest request) {
        DryRunResult result = rollbackManager.dryRunRollback(request.getTargetVersion());
        return ResponseEntity.ok(result);
    }

    @GetMapping("/snapshots")
```

```
    public ResponseEntity<List<SnapshotInfo>> listSnapshots() {
        List<SnapshotInfo> snapshots = snapshotManager.listSnapshots();
        return ResponseEntity.ok(snapshots);
    }
}
```

Production Deployment Strategy

1. Pre-Rollback Checklist

java

```
@Component
public class RollbackSafetyChecker {

    public SafetyCheckResult performSafetyCheck(String targetVersion) {
        SafetyCheckResult result = new SafetyCheckResult();

        // Check 1: Active transactions
        if (hasActiveTransactions()) {
            result.addWarning("Active transactions detected");
        }

        // Check 2: Replication lag
        if (getReplicationLag() > 60) {
            result.addError("Replication lag too high");
        }

        // Check 3: Data dependencies
        List<DataDependency> dependencies = checkDataDependencies(targetVersion);
        if (!dependencies.isEmpty()) {
            result.addWarning("Data dependencies found", dependencies);
        }

        // Check 4: Foreign key constraints
        List<ForeignKeyViolation> violations = checkForeignKeyViolations(targetVersion);
        if (!violations.isEmpty()) {
            result.addError("Foreign key violations detected", violations);
        }

        return result;
    }
}
```

2. Zero-Downtime Rollback Process

1. **Blue-Green Deployment Pattern**

1. Create snapshot of current state
2. Deploy rollback to green environment
3. Validate green environment
4. Switch traffic to green
5. Monitor for issues
6. Keep blue environment as fallback

2. **Canary Rollback**

1. Apply rollback to small percentage of instances
2. Monitor metrics and errors
3. Gradually increase rollback percentage
4. Complete rollback or abort based on metrics

3. Monitoring and Alerting

java

```java
@Component
@Slf4j
public class RollbackMonitor {

    @Autowired
    private MeterRegistry meterRegistry;

    @EventListener
    public void handleRollbackEvent(RollbackEvent event) {
        // Record metrics
        meterRegistry.counter("flyway.rollback.attempts",
            "version", event.getTargetVersion(),
            "status", event.getStatus().toString()).increment();

        // Alert on failures
        if (event.getStatus() == RollbackStatus.FAILED) {
            alertingService.sendAlert(new RollbackFailureAlert(event));
        }

        // Log detailed information
        log.info("Rollback event: {}", event);
    }
}
```

Best Practices

1. **Always Test Rollbacks**
   - Test in staging environment first

- Use production-like data volumes
- Verify data integrity after rollback

2. **Maintain Backward Compatibility**
   - Keep old columns for grace period
   - Use feature flags for application code
   - Support multiple schema versions
3. **Document Rollback Procedures**
   - Create runbooks for each migration
   - Document data dependencies
   - Include recovery time objectives (RTO)
4. **Implement Circuit Breakers**
   - Fail fast on repeated failures
   - Automatic fallback to snapshot restore
   - Health checks after rollback
5. **Audit Everything**
   - Log all rollback attempts
   - Track who approved rollbacks
   - Monitor rollback duration and impact

Conclusion

This comprehensive framework provides production-ready rollback capabilities for Flyway in Spring Boot applications. It handles all SQL operation types, maintains data integrity, and provides the safety mechanisms necessary for production deployments. The key is combining automated rollback generation with manual oversight and thorough testing to ensure safe database migrations in production environments.