Flyway Migration Workflow

This document explains how to manage database schema changes using JPA entities and Flyway migrations.

Important Principle

JPA entities do NOT automatically update the database schema. We use Flyway for all schema changes to maintain control and versioning.

Configuration

In `application.yml`:

```
spring:
  jpa:
    hibernate:
      ddl-auto: validate  # IMPORTANT: Only validates, never auto-
creates/updates schema
```

This ensures Hibernate will only validate that entities match the schema, but won't modify it.

Workflow: From JPA Entity Changes to Database Migration

Step 1: Modify Your JPA Entity

Example: Let's say you want to add a `description` field to `ShellScript`:

```
@Entity
@Table(name = "shell_script")
data class ShellScript(
    // ... existing fields ...

    @Column(nullable = true)
    val description: String? = null,  // NEW FIELD
)
```

Step 2: Generate Migration SQL (Manual Process)

Since we're using `ddl-auto: validate`, Hibernate won't auto-generate the schema. Instead, you need to:

Option A: Write SQL Manually (Recommended)

1. Analyze what changed in your entity

2. Write the corresponding SQL migration
3. Test it thoroughly

Option B: Use Hibernate Schema Generation Temporarily

If you want Hibernate to help generate the SQL (for complex changes):

1. Temporarily change `application.yml`:

```
spring:
  jpa:
    hibernate:
      ddl-auto: update  # TEMPORARY - for SQL generation only
    properties:
      hibernate:
        hbm2ddl:
          auto: update
```

2. Run the application - Hibernate will update the schema
3. Use a DB diff tool to see what changed, or check Hibernate logs
4. **IMPORTANT**: Revert `ddl-auto` back to `validate`
5. Write proper Flyway migration based on what you learned

Option C: Use a Diff Tool

Tools like:

- **Liquibase Diff** (can generate changesets from JPA entities)
- **SchemaCrawler**
- **DB comparison tools in IntelliJ IDEA**

Step 3: Create Flyway Migration File

Create a new migration file in `src/main/resources/db/migration/`:

Naming convention: `V{version}__{description}.sql`

Example: `V2__Add_description_to_shell_script.sql`

```
-- Add description column to shell_script table
ALTER TABLE shell_script ADD COLUMN description TEXT;
```

Version numbering:

- `V1__` - Initial schema (already created)
- `V2__` - First change

- `V3__` - Second change
- etc.

Step 4: Test the Migration

1. Stop the Spring Boot application if running
2. Start the application - Flyway will automatically apply the migration
3. Check the logs for successful migration:

```
Flyway: Successfully applied 1 migration to schema `main` (execution
time 00:00.123s)
```

Step 5: Verify

1. Check that the schema matches your entities (Hibernate validation should pass)
2. Test your application functionality
3. Commit both the entity changes AND the migration file to git

Example Migration Files

Adding a Column

File: `V2__Add_description_to_shell_script.sql`

```
ALTER TABLE shell_script ADD COLUMN description TEXT;
```

Adding a New Table

File: `V3__Create_user_preferences_table.sql`

```
CREATE TABLE user_preferences (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    preference_key TEXT NOT NULL,
    preference_value TEXT NOT NULL,
    created_at REAL NOT NULL DEFAULT (CAST((julianday('now') - 2440587.5)
* 86400000.0 AS REAL))
);

CREATE INDEX user_preferences_user_id_idx ON user_preferences(user_id);
```

Modifying a Column (SQLite)

Note: SQLite has limited ALTER TABLE support. To modify columns, you often need to recreate the table:

File: `V4__Change_script_command_to_longtext.sql`

```
-- SQLite doesn't support ALTER COLUMN, so we recreate the table
PRAGMA foreign_keys=OFF;

CREATE TABLE new_shell_script (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    command TEXT NOT NULL,  -- Changed from TEXT to TEXT (no change needed
in SQLite)
    ordering INTEGER NOT NULL,
    description TEXT,
    created_at REAL NOT NULL DEFAULT (CAST((julianday('now') - 2440587.5)
* 86400000.0 AS REAL)),
    created_at_hk TEXT NOT NULL DEFAULT (strftime('%Y-%m-%d %H:%M:%S',
datetime('now', '+8 hours')))
);

INSERT INTO new_shell_script SELECT * FROM shell_script;
DROP TABLE shell_script;
ALTER TABLE new_shell_script RENAME TO shell_script;

CREATE INDEX shell_script_id_idx ON shell_script(id);

PRAGMA foreign_keys=ON;
```

Best Practices

1. **Never skip versions** - Migrations must be sequential
2. **Never modify existing migration files** - Once applied, they're immutable
3. **Always test migrations** on a copy of production data
4. **Keep migrations small** - One logical change per migration
5. **Write reversible migrations** when possible (for rollback)
6. **Document complex migrations** with comments in SQL
7. **Test both up and down migrations** if you support rollback

Troubleshooting

"Migration checksum mismatch"

This means a migration file was modified after being applied. Solutions:

- Revert the file to its original state
- Or use Flyway repair (not recommended for production)

"Validation failed: Migration ... detected"

A new migration file appeared, but Flyway sees a gap in versions. Ensure sequential numbering.

Entity doesn't match schema

Run the application - if `ddl-auto: validate` fails, it means:

1. Your migration is missing some change, OR
2. Your entity definition doesn't match the migration

Flyway Commands (via Gradle)

```
# Apply all pending migrations
./gradlew flywayMigrate

# Validate applied migrations
./gradlew flywayValidate

# Show migration status
./gradlew flywayInfo

# Repair metadata table (use with caution)
./gradlew flywayRepair

# Clean database (DANGER: deletes all data)
./gradlew flywayClean
```

Recommended Development Flow

1. Make entity changes
2. Write migration SQL
3. Test locally
4. Commit entity + migration together
5. Deploy (Flyway auto-runs on startup)
6. Verify in production

Additional Resources

- Flyway Documentation
- SQLite ALTER TABLE limitations
- Hibernate DDL-auto modes