

Creating Migrations

James P. Gilbert

2025-09-24

Contents

Introduction	1
Assumptions	1
Creating the required file structure	1
In an R package	2
Using folder structure	2
Adding a migration	2
Adding migrator	2
Unit testing	3
Common issues	3
Supporting all database platforms	3
SQLite column types	3
Non-existent data	4

Introduction

Migrating existing data models can be a tricky process that often creates incompatibility between result viewers and existing result sets. This guide aims to show how to use the **ResultModelManager** class to create migrations for a given result model, either using a package or file structure. Please see the HADES library for more information on HADES packages.

Assumptions

This package assumes that you are familiar with R and OHDSI Hades packages in general. These examples make use of **DatabaseConnector** and **SqlRender**. The management of data integrity is left to the user, migrations should be designed and tested before deployment. Steps to maintain the data (such as backup plans) should be made prior to performing migrations in case of data corruption.

Creating the required file structure

The first step is creating a proper folder structure for migrations. The chosen path is dependent on the structure used, the most consistent and recommended way is to expose a function within an R package to allow users to upgrade a data model. However, a flat folder structure that does not require an R package to be installed is also supported.

In an R package

Data migrations should exist in an isolated folder within the `/inst/sql/` directory of a package. The recommended convention is to use `migrations` across all Hades package. As migrations are supported by multiple database platforms this folder should exist within the generic (and SqlRender OHDSI common sql) `sql_server` folder, `inst/sql/sql_server/migrations`. For any database specific migrations they should be in the appropriate sub directory. For example:

```
inst/sql/  
  sql_server/migrations/Migration_1-create.sql  
  sqlite/migrations/Migration_1-create.sql  
  redshift/migrations/Migration_1-create.sql
```

Using folder structure

A folder structure requires a slightly different set up. Here, the migrations should be split via database platform within the migration path. For example.

```
migrations/  
  sql_server/Migration_1-create.sql  
  sqlite/Migration_1-create.sql  
  redshift/Migration_1-create.sql
```

Adding a migration

All data migrations are assumed to be in OHDSI SQL and stored within a migration folder (see above for set up). Inside this folder only migrations that conform to a regular expression such as `(Migration_[0-9]+)-(.+).sql`. Explicitly, this encodes several things:

- That the file is a migration and only intended to be executed once and by a DMM instance
- The position in the sequence in which the migration will be executed (i.e. a natural number)
- The string name of the migration
- The fact that its an sql file

For example, the following file names will work:

```
Migration_2-MyMigration.sql  
Migration_2-v3.2whatever.sql  
Migration_4-TEST.sql  
Migration_4-2018922-vAAAA.sql
```

However, the following would be invalid:

```
MyMigration.sql # Does not include Migration_1  
Migration_2v3.2whatever.sql # missing -  
-TEST_Migration_1.sql # Wrong order  
Migraton_4-a.sql # Migration spelt wrong
```

Adding migrator

Each package/project should expose an instantiated DMM with the package specific considerations. For example, for the package `CohortDiagnostics` a function such as the following may be written:

```
#' @export  
getDataMigrator <- function(connectionDetails, databaseSchema, tablePrefix) {  
  ResultModelManager::DataMigrationManager$new(  
    connectionDetails = connectionDetails,
```

```

    databaseSchema = databaseSchema,
    tablePrefix = tablePrefix,
    migrationPath = "migrations",
    packageName = "CohortDiagnostics"
  )
}

```

This will return an instance of data migrator that will expose the functionality on a given data set. Naturally, the package is not strictly required for creating a migration manager (should the directory structure conform to the above outline) here you should set it according to your project's set up.

Loading the migrator is then straightforward:

```

connectionDetails <- DatabaseConnector::createConnectionDetails(MySettings)
migrator <- getDataMigrator(connectionDetails = connectionDetails, databaseSchema = "mySchema", tablePrefix = "myPrefix")

```

To check migrations are valid

```

migrator$check() # Will return false and display any erroneous files

```

To get the status of all migrations

```

migrator$getStatus() # Will return data frame of all sql migrations and if they have been executed or not

```

To run the migrations:

```

## It is strongly recommended that you create some form of backup before doing this
migrator$executeMigrations()

```

Unit testing

No specific advice is given for how to write unit tests for migrations, however, it is strongly advised that migrations are unit tested.

Common issues

The following is a list of expected issues when handling Data migrations:

Supporting all database platforms

It is likely a challenge to support all SqlRender/DatabaseConnector supported dbmses. Therefore, careful consideration with regards to supported platforms should be made. At the time of writing, for results handling, we recommend supporting the open source platforms of SqlRender and Postgresql. This decision is left to the package author.

SQLite column types

It is not possible to change a data type within an SQLite table (the `ALTER TABLE` command does not work). Consequently, you will likely have to rename the existing table, create a new table with the modified DDL and then copy the existing data across (using appropriate data transformations/casting).

For example, changing an INT column in the table `foo` to a float requires the sqlite specific transformation:

```

“{sqlite-sql} {DEFAULT @foo = foo}

```

```

ALTER TABLE @database_schema.@table_prefix@foo RENAME TO _foo_old;

```

```

CREATE TABLE @database_schema.@table_prefix@foo ( id bigint, foo float );

```

```
INSERT INTO @database_schema.@table_prefix@foo (id, foo) SELECT * FROM __foo_old; ““
```

Non-existent data

The presence of a data model does not mean data is present. As packages are developed, it is expected that new data formats will be created. The recommended pattern for this case is to allow existing data to be upgraded but to handle the use case of missing data in downstream reports/web applications.