

Lesson 24: Executing SQL During App Deployment

Created by Jacques Marais, last modified by Charl Viljoen on Jun 08, 2020

As any app user I want to have the app initialised upon deployment to include the default stock items.

As a System Admin I want to have access to a data table that summarises all purchases in the system.

- Lesson Outcomes
- App Use Case Scenario
- New & Modified App Files
- Packaging SQL Script Files With App Source Code
- Script Execution
 - Order of Script Execution
 - Script Errors & Debugging
- Initial App Data Using SQL Scripts
- Materialised View for Reporting Using SQL Scripts
 - Data Model Additions
 - SQL Script
 - View & Presenter Additions
- Helium Schema Upgrade Procedure
- A Note on Temp Tables
- Lesson Source Code

Lesson Outcomes

By the end of this lesson you should know how to package SQL scripts with DSL source code in order for scripts to be executed during the schema creation/upgrade process of app deployment. You should understand the order of execution and possible usages of the feature.

App Use Case Scenario

In [Lesson 8](#), we introduced a feature that automatically creates stock items when none are present in the system. This was done to avoid having to create these default stock items each time the app is newly deployed. The invocation for this logic was included in the invite function for the "[System Admin](#)" role. This means that when a System Admin is invited to the app using the Helium core web app or the HeliumDev client, the default stock items are created if they do not already exist.

A better way to achieve the same result is to have the default records created during deployment instead of during user invitation. This can be achieved by packaging SQL for this purpose with the DSL source code.

In addition to this use case we can also make use of the same Helium feature to provide a mechanism by which to efficiently report on all purchases in the system. For this we will include a SQL script that drops an existing table, and creates a materialized view in its place. Queries can then be performed directly on the newly created view by using the [native DSL SQL features](#) or [selectors](#).

New & Modified App Files

`./model/objects/FarmerPurchaseSummary.mez`

`./web-app/views/farmer_purchase_summary/FarmerPurchaseSummary.vxml`

`./web-app/presenters/farmer_purchase_summary/FarmerPurchaseSummary.mez`

```
./sql-scripts/create_default_stock.sql  
./sql-scripts/create_farmer_purchase_summary_view.sql  
./services/UserInvite.mez
```

Packaging SQL Script Files With App Source Code

SQL scripts are packaged with DSL app source code by including them in a folder named `sql-scripts`. This folder needs to be in the app root directory. This folder is processed recursively for any files with a `.sql` extension. This means that you can have files in nested folders with no limit on the depth. As long as the correct file extension of `.sql` is used, these files will be included in your app source code that gets sent to the Helium server.

Script Execution


Order of Script Execution

Scripts are executed in alphabetical order of the path to that file. In other words, when execution order is determined, the path to that file in your project and the file name is taken into account. It is highly recommended that a strict naming convention and folder structure is followed when adding SQL script files to your project as out of order execution might result in SQL errors which will cause the app deployment to fail. One example of such a naming scheme is to prepend the file name with the date on which it was added for example:

```
2017-09-12 create_default_records.sql
```

```
2017-09-18 update_default_records.sql
```

Note that Helium will attempt to execute the scripts each time the app is deployed. SQL scripts should therefore make provision for this to avoid SQL errors or the execution of SQL that might lead to inconsistent app data. This is discussed along with examples in following sections.

 Be sure to follow a strict naming convention for SQL scripts that will ensure the correct order of execution.

Script Errors & Debugging

If a SQL script that is packaged with app source fails to execute, the error along with file name will be reported in the HeliumDev client. The packaged SQL scripts are seen as part of the app model and is executed as part of the schema upgrade for existing apps and schema creation for new apps. For this reason a script that results in an error will cause the entire deployment to fail and be rolled back. Once the script has been fixed the deployment can proceed. Also note that no SQL select queries should be packaged with app source code. Helium executes SQL scripts without expecting any result from the query. For this reason a select query will result in a failed schema upgrade and thus a failed deployment.

Initial App Data Using SQL Scripts

In order to create default app data for the stock object we simply include a SQL script file in our `sql-scripts` folder with insert statements for the default data. Keep in mind that the script will be run regardless of whether there is already data for default stock items or not. The SQL script itself should therefore make provision for this. See the two statements below for examples on how this can be achieved:

One way of achieving this is to use of a conditional insert that makes use of NOT EXISTS.

```
1  
2 INSERT INTO stock (_id_, _tstamp_, name, stocktype, del
```

```
3 SELECT '80dc5655-9600-440b-86c0-614ccaef11fe', current_
4 WHERE NOT EXISTS (
5     SELECT _id_ FROM stock WHERE _id_ = '80dc5655-9600-
    );
```

Another, more preferable, way of achieving the same result is to make use of the so called upsert functionality provided in PostgreSQL 9.5. Using this, the desired behaviour on insert conflicts can be specified. In this case the statements are as follows:

```
1 INSERT INTO stock (_id_, _tstamp_, name, stocktype, de
```

Materialised View for Reporting Using SQL Scripts

Data Model Additions

In order to make use of a materialized view for reporting purposes, we first create a matching persistent object in our app. After deployment, this will be dropped and a materialised view will be created in its place. The result is an object that can be queried in the same manner as a normal object using selectors but where the data is backed by our newly created materialised view.

```
1 persistent object FarmerPurchaseSummary{
2
3     // When was the purchase made
4     datetime purchasedOn;
5
6     // Quantity and cost of purchase
7     int quantity;
8     decimal unitPrice;
9     decimal goodsCost;
10    decimal discount;
11    int finalCost;
12
13
14    // Helium provided payment status and id
15    datetime paymentStatusUpdatedOn;
16    PAYMENT_STATUS paymentStatus;
17    uuid paymentId;
18
19    // Stock details
20    string stockName;
21    string stockType;
22
23    // Farmer details
24    string farmerNames;
25    string farmerMobileNumber;
26
27    // Shop details
28    string shopCode;
29    string shopName;
```

```
29 }
}
```

SQL Script

The following script is used to drop the existing table or materialised view and recreate it. Note the use of the `__he_delete_table_or_view__` function. This function is provided by Helium as part of the app schema in order to assist with use cases such as this. This is required for the following reason:

When the script is first run, the existing table needs to be dropped. Using "drop table if exists" will achieve this. After the table is dropped, the materialised view is created. For subsequent deployments, however, the script will again attempt to drop the table even though the table does not exist, while a view with the same name does exist. This will result in a SQL error and the schema upgrade and thus deployment will fail.

```
1 DO $$ BEGIN
2     PERFORM __he_delete_table_or_view__('farmerpurchasesummary');
3 END $$;
4 CREATE MATERIALIZED VIEW farmerpurchasesummary AS
5 SELECT
6     farmerpurchase._id_,
7     current_date,
8     purchasedon,
9     quantity,
10    unitprice,
11    goodscoast,
12    discount,
13    finalcost,
14    paymentstatusupdatedon,
15    paymentstatus,
16    paymentid,
17    stock.name AS stockname,
18    stock.stocktype AS stocktype,
19    farmer.firstname || ' ' || farmer.lastname AS farmername,
20    farmer.mobilenumber AS farmermobilenumber,
21    shop.shopcode AS shopcode,
22    shop.name AS shopname
23 FROM farmerpurchase
24 LEFT JOIN stock on farmerpurchase.stock_fk = stock._id_
25 LEFT JOIN farmer on farmerpurchase.farmer_fk = farmer._id_
26 LEFT JOIN shop on farmerpurchase.shop_fk = shop._id_;
```

View & Presenter Additions

In addition to the above, we also add a new view with a data table and a backing presenter to display the data from our view. See the lessons source code for the complete example.

Helium Schema Upgrade Procedure

When a deployment is done for a Helium application aside from the source code that is upgraded, Helium will automatically apply changes made in the model definition of the DSL app to the actual postgres database. The database changes will be applied the first time the application is accessed after a release.

Database changes that **are** migrated in schema:

- Introduce a new persistent object (The new table is created).
- Introduce a new attribute for an existing persistent object (Existing table is altered with the new column introduced).
- Introduce a new constraint, such as a unique or foreign key constraint through Validators (See [Lesson 3: User Input, Persistence, Validation, and Tables \(continued\)](#)). (The constraint is added)

Database changes that **are NOT** migrated in schema:

- Remove persistent object or attribute on an object (No tables or columns are dropped)
- Remove a validator on an attribute of an object. (The database constraint is not dropped)

In cases where a database migration is needed and Helium will not apply all of the changes needed (Such removing objects, attributes or db constraints) the developer will need to migrate the app schema by hand using SQL.

A Note on Temp Tables

It's important to note that SQL making use of [temporary tables](#) are, in most cases, not supported in Helium and in general the use of temporary tables in DSL apps is discouraged. Please see additional notes on this [here](#).

Lesson Source Code

[Lesson 24.zip](#)

No labels