Lesson 22: Executing SQL Select Queries From the DSL

Created by Jacques Marais, last modified on Jul 22, 2019

> *As a Shop Owner I would like to see a data table with the number of purchases per week for a specified period and shop.*

Lesson Outcomes

By the end of this lesson you should:

- Be able to declare multi-line string literals using the multi-line string syntax
- Be able to retrieve data from the database by executing a PostgreSQL select query from within a DSL app and map the data to a non-persistent object collection

App Use Case Scenario

The data model for our app creates a record for each purchase that takes place.

In order to display aggregated data related to purchases we have the following options:

- Create an additional persistent object which keeps track of the number of purchases per week, per shop as new purchases happen. This adds complexity to the app. In addition to complicating the data model and logic when purchases are made, possible race conditions also need to be taken into account when submitting purchases from mobile clients. This implies that aggregation might need to happened server side using scheduled functions. Although this is not the case for our tutorial app, it is a use case that is often encountered in the real world.
- Aggregate the data as it's needed using DSL selectors. Aggregating the data using DSL selectors will result in manual aggregation using loops. This will be computationally expensive and depending on the size of the data being aggregated, might be prohibitively slow. It is also an error prone method of data aggregation.
- Make use of a native PostgreSQL select query in the DSL. This alternative will be explored in the remainder of this lesson. We will make use of a SQL query and map the results to a collection of non-persistent objects. Using this method we have the advantages of flexibility and performance as provided by SQL while maintaining relatively simple and legible DSL source code.

New & Modified App Files

`./web-app/presenters/purchase_frequency/PurchaseFrequency.mez`

`./web-app/views/purchase_frequency/PurchaseFrequency.vxml`

`./model/objects/PurchaseFrequencyResult.mez`

`./web-app/lang/en.lang`

Model Additions

Our query results will be stored in an in-memory collection of non-persistent objects. This object we will be using is shown below:

```
object PurchaseFrequencyResult {
    date weekStart;
    date weekEnd;
    int count;
}
```

View & Presenter Additions

For this app feature, we will add the The **`PurchaseFrequency.vxml`** view file and **`PurchaseFrequency.mez`** presenter file containing a `PurchaseFrequency` unit.

The `PurchaseFrequency` view contains input fields for the start date, end date and shop that purchases are to be filtered upon. This is followed by a submit button to submit the filters and a data table to display the query results.

In the presenter, we have an `init` function to initialise the start date, end date, and shop variables. We also have a `submitFilter` function that performs manual validation before reloading the same view by navigating to it.

Our query and the execution thereof is performed in the collection source function `getPurchaseFrequency`. This is discussed further in the following sections of this lesson.

The source code for our presenter and view and a screenshot of the view is shown below:

```
<view label="view_heading.purchase_frequency" unit="Pu

    <menuitem label="menu_item.purchase_frequency">
        <userRole>Shop Owner</userRole>
    </menuitem>

    <select label="select.shop">
        <binding variable="selectedShop" />
        <collectionSource variable="availableShops">
            <displayAttribute name="name" />
        </collectionSource>
    </select>

    <datefield label="datefield.start_date">
        <binding variable="startDate"/>
    </datefield>

    <datefield label="datefield.end_date">
        <binding variable="endDate"/>
```

```
    </datefield>

    <submit label="submit.submit" action="submitFilter

    <table>
        <collectionSource function="getPurchaseFrequen
        <column heading="column_heading.week_start">
            <attributeName>weekStart</attributeName>
        </column>
        <column heading="column_heading.week_end">
            <attributeName>weekEnd</attributeName>
        </column>
        <column heading="column_heading.purchases">
            <attributeName>count</attributeName>
        </column>
    </table>
</view>
```

```

unit PurchaseFrequency;

datetime startDate;
datetime endDate;

Shop selectedShop;
Shop[] availableShops;

void init() {
    // Initialise the start and end date filters
    endDate = Mez:now();
    startDate = Date:addDays(endDate, -30);

    // Initialise the available shops collection and s
    availableShops = Shop:relationshipIn(owners, ShopC
    if (availableShops.length() > 0) {
        selectedShop = availableShops.get(0);
    }
}

// Apply the date and shop filters
DSL_VIEWS submitFilter() {

    if(startDate == null) {
        Mez:alertError("alert_error.no_start_date");
        return null;
    }

    if(endDate == null) {
        Mez:alertError("alert_error.no_end_date");
```

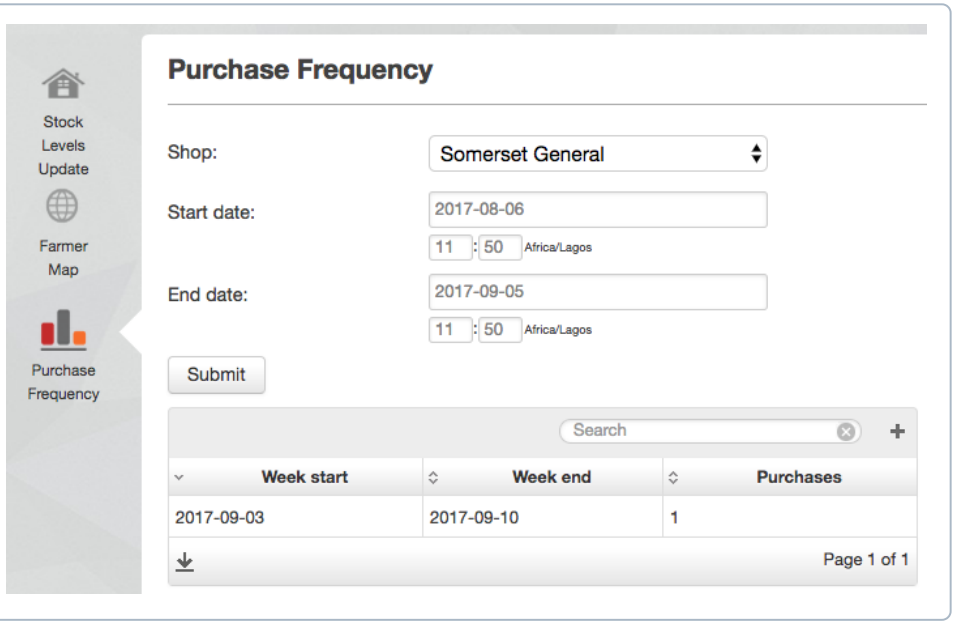
```
        return null;
    }

    if(selectedShop == null) {
        Mez:alertError("alert_error.no_shop_selected")
        return null;
    }

    return DSL_VIEWS.PurchaseFrequency;
}

// Execute SQL for report data and return as collectic
PurchaseFrequencyResult[] getPurchaseFrequency() {
    string query = /%
        WITH intervals AS (
            SELECT weekstarts.weekstart AS weekstart,
            FROM (
                SELECT
                weeks.i - cast(extract(dow from weeks.
                FROM (
                    SELECT i::date from generate_serie
                ) AS weeks
            ) AS weekstarts
        )
        SELECT weekstart, weekend, count(*)::int
        FROM farmerpurchase
        JOIN intervals
        ON farmerpurchase.purchasedon > (weekstart - 1
        GROUP BY weekstart, weekend;
    %/;

    PurchaseFrequencyResult[] result = sql:query(query
    return result;
}
```



Mutli-line Strings Literals

Note in the source code for the `PurchaseFrequency` presenter, the query is defined using a special syntax. **`/%`** and **`%/`** is used to denote the start and end of a multi-line string literal.

Query Execution and Result Mapping

The `getPurchaseFrequency` function executes our query and returns a resulting collection for the data table. The function of the query is to generate week intervals between the start and end date specified by the user, and then count purchases for each of these intervals. The resulting aggregate therefore represents purchase frequency.

To execute our query, we make use of the `sql:query` built-in function. This function takes the query as its first argument. The query can be specified as a string literal, variable, attribute or function that returns string. Note that the query that is passed to this function must be a select query. Any other type of query will result in an exception. The arguments that follow are parameters to our query. These are represented in the query itself by **`?`** and values are substituted at runtime. Once again the values can be specified using string literals, variables, attributes or functions.

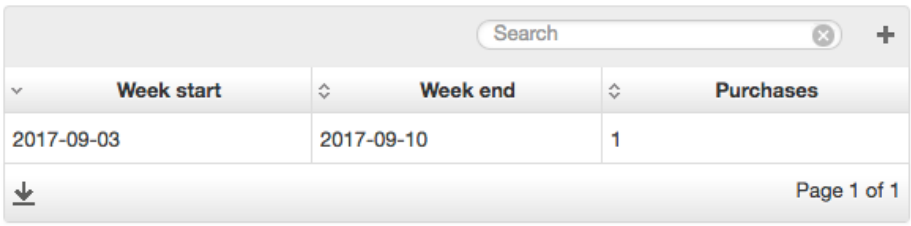
The `sql:query` function returns a collection of objects where the object attributes have to correspond to the resulting columns of the query with regards to both the name and type. Consider, as an example, the query result as run on the database directly, the results of the same query as executed from the DSL and also the object used for the query results:

```
weekstart  |  weekend   | count
-----------+------------+-------
 2017-09-03 | 2017-09-10 |     1
(1 row)
```

> Attribute names and types need to correspond to the columns for the data being returned from your select query.

> Query results should only be mapped to collections of non-persistent objects. Using persistent objects will result in undefined behaviour.

> Be aware that SQL that is executed using `sql:query` is not executed when the `sql:query` statement is encountered. Rather, it is executed when the result set is first referenced. This means that if the SQL contains errors which result in runtime errors, the error will not be generated as part of `sql:query` but rather as part of the code that accesses the result set of the statement execution.




```
 object PurchaseFrequencyResult {
    date weekStart;
    date weekEnd;
    int count;
}
```

Note from the above example that our attribute names are in camel case whereas our query columns are not. The mapping between columns and attributes is not case sensitive.

Also note that while it is possible to map simple query results to persistent object collections, it is not recommended. This is due to the following reasons:

- Populating relationships are not supported.
- Runtime errors will occur when accidentally saving objects that were retrieved using a select query.
- Using non-persistent objects with descriptive names more clearly defines the purpose of the objects and improves code legibility.

In cases where there is no direct data type in the DSL to represent a data type retrieved from a query, casting can be used. In our example this is shown by the result of the PostgreSQL count function result being cast to int:

```
count(*)::int
```

Explicit casting in queries might be needed to match the resulting row data type with a DSL object attribute.

This is done because of the fact that the PostgreSQL function returns a value of type long. The closest to this type in the DSL is int, and therefore the above casting is performed in the query.

Another example of explicit casting to take note of is in the case where a mapping to a DSL **`decimal`** type needs to occur. The underlying Java implementation of the **`decimal`** type in the DSL is double. For this reason a value needs to be cast to double precision in the SQL query before it can be mapped to a **`decimal`** attribute in the DSL. This casting can be achieved as follows:

```
select 9.98765665543356::double precision
```

```
select cast(9.98765665543356 as double precision)
```

Number of Records Limit

Reaching the configured or default limit on the number of records

The number of records returned using the `sql:query` function is limited by default. By default the limit is 1000 records. If an app requires a higher limit or the limit to be removed altogether, it has to be requested by means of a support request by following the process defined here. This app configuration will then be updated which will override the system default.

If the number of records retrieved by the `sql:query` function exceeds the current limit, an exception will be thrown. The details of the error can, as per usual, be inspected using the Helium Logging Service.

returned by the `sql:query` function will result in an exception that can seen using the Helium Logging Service.

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.

Avoid SQL using temporary tables in DSL apps.

Lesson Source Code

Lesson 22.zip

No labels