The ViewFoundSet

A ViewFoundSet (since 8.4) is a foundset based on a QBSelect query object (since 8.4). A ViewFoundSet can be created by:

```
databaseManager.getViewFoundSet(String name, QBSelect query)
```

This will return a ViewFoundSet object that has as datasource: "view:[name]". This foundset will have a different API then the normal JSFoundSset, it has the basic stuff like getSize(), getRecord(), forEach(), loadAllRecords() but no find/search support (just create a new ViewFoundSet with the updated QBSelect), or new/delete record.

Because it is just based on a QBSelect you can query from many tables a set of data at once. So using this can be a big improvement for showing table data that normally would show related data through relations or through a valuelist or aggregates from a join table.

If you want to use a ViewFoundSet on a Form, you can create one and then register this to the system:

```
databaseManager.registerViewFoundSet(ViewFoundSet foundset)
```

This will mean that that instance will be kept in memory for forms that have that datasource (set through the solution model in 8.4, next major release will have ViewFoundSet support in the developer itself). If you don't need it anymore:

```
databaseManager.unregisterViewFoundSet(String datasource)
```

By default a ViewFoundSet will not react to databroadcast changes, you can enable that by calling:

```
viewfoundset.enableDatabroadcastFor(QBTableClause queryTable, int flags)
```

So for a specific QBSelect table like the main table or a join table, you can enable databroadcast. Depending on the flag(s) that is given certain stuff will happen or the pk's for that table should also be selected (besides the columns that you really wanted to query).

By default if only enableDatabroadcastFor(table) is called we will monitor only the columns for that table based on the pk that is also selected, So if a databroadcast comes in for that table, we look if we have that pk in our ViewFoundSet and update the columns that are changed, those columns could come from the existing RowManager/Record data we already have, then no query is done at all, but it could also result in a single query selecting the changed column by pk on that table. So this ViewFoundSet.MONITOR_COLUMNS flag should not result in a big performance hit.

These are the flags that can be currently used:

```
    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen also for column
     * changes of the given table/datasource. This is used by default if you just use enableDatabroadcastFor()
     * without flags. If you use the one with the flags you need to give this one if you just want to listen to
     * column changes that are in the result for a given datasource and pk.
     *
     * This constants needs to have the pk's selected for the given datasource (should be in the results).
     */
    ViewFoundSet.MONITOR_COLUMNS;

    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen also for column
     * changes of the given table/datasource in the join statement - like order_lines.productid that has a join
     * to orders and is displaying the productname. If a change in such a join condition (like
     * order_lines.productid in the sample above) is seen then the query will be fired again to detect changes.
     */
    ViewFoundSet.MONITOR_JOIN_CONDITIONS;

    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen also for column
     * changes of the given table/datasource that are used in the where statement - like
     * order_lines.unit_price > 100. If a change is seen on that datasource on such a column used in the where
     * a full query will be fired again to detect changes.
     */
    ViewFoundSet.MONITOR_WHERE_CONDITIONS;

    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen for inserts on the
     * given table/datasource. This will always result in a full query to detect changes whenever an insert on
```

```
     * that table happens.
     */
    ViewFoundSet.MONITOR_INSERT;

    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen for deletes on the
     * given table/datasource. This will always result in a full query to detect changes whenever an delete on
     * that table happens.
     */
    ViewFoundSet.MONITOR_DELETES;

    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen for deletes on the
     * given table/datasource which should be the primary/main table of this query. If a delete comes in for this
     * table, then we will only remove the records from the ViewFoundSet that do have this primary key in its
     * value. So no need to do a full query. So this will only work if the query shows order_lines for the
     * order_lines table, not for the products table that is joined to get the product_name. Only 1 of the 2
     * monitors for deletes should be registered for a table/datasource.
     *
     * This constants needs to have the pk's selected for the given datasource (should be in the results)
     */
    ViewFoundSet.MONITOR_DELETES_FOR_PRIMARY_TABLE;

    /**
     * Constant for the flags in {@link #enableDatabroadcastFor(QBTableClause, int)} to listen for changes in
     * columns (selected) of the given datasource in the query that can affect aggregates. This means that when
     * there are deletes, inserts or updates on columns selected from that datasource, a full re-query will
     * happen to refresh the aggregates.
     *
     * IMPORTANT: in general, this flag should be set on (possible multiple) datasources from the query that
     * have group by on their columns, and the columns don't contain the pk, or that have the actual aggregates
     * on their columns (because all those could influence the value of aggregates).
     *
     * For example (ignoring the fact that in a real-life situation these fields might not change), a view
     * foundset based on this query:
     *
     * SELECT orders.customerid, orders.orderdate, SUM(order_details.unitprice) FROM orders
     *    LEFT OUTER JOIN order_details ON orders.orderid = order_details.orderid
     *    GROUP BY orders.customerid, orders.orderdate
     *      ORDER BY orders.customerid asc, orders.orderdate desc
     *
     * will want to enable databroadcast flag MONITOR_AGGREGATES on both "orders" (because if "orderdate" or
     * "customerid" - that are used in GROUP BY - change/are corrected on a row, that row could move from one
     * group to the other, affecting the SUM(order_details.unitprice) for the groups involved) and
"order_details"
     * (because if "unitprice" changes/is corrected, the aggregate will be affected).
     *
     * But if the above query would also select the orders.odersid (and also group by that) then the orders row
     * that you select for that sum will always be unique and only {@link #MONITOR_COLUMNS} has to be used for
     * those - if needed.
     */
    ViewFoundSet.MONITOR_AGGREGATES;
```

So some like MONITOR_COLUMNS or MONITOR_DELETES_FOR_PRIMARY_TABLE will not have a big impact on performance but the others could have a larger impact because of the full queries that are done then when a change is seen to update the ViewFoundSet.

Besides monitoring for databroadcast, the ViewFoundSet can als be updateable by using one of the 2 save() functions on it. There is one requirement for this and that is that for all the columns that you change in a ViewRecord of a table, you also have to have the pk selected for that table in the QBSelect. So that Servoy can generate an update statement to the table with the changed columns for that pk.

If you change data in a ViewFoundSet then all the refreshes based on the databroadcast flags (and also loadForMoreData, so loading in then next 200 records) will be postponed until you save the changed Records. This works this way because we can't do a query to update/add more records because we don't know what the changed record exactly is, how it maps on the new data.