Query builder

Query Builder

In Servoy 6.1 a query builder is introduced.
With the query builder you can compose query objects with a sql-like structure.
The query is database-independant, Servoy will convert the query object to a sql string that is applicable for the specific database.

The query object can be used to load foundsets or to create a data set.

The query builder API is available in both the java plugin api and the javascript api.
The Java api is described in the Servoy public API page, this document describes how to use the query builder api in javascript.

Javascript editor support

The javascript editor lists the possible properties and methods in a query builder and its subobjects via code completion.
To be able to list the right columns of a table, the query object sometimes needs to be typed, like for example:

```
/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
```

This tells the javascript editor the query is on the orders table, so the query.columns property will list the orders table columns.
When type info is used correctly, the javascript editor will give warnings on non-existing columns in case of typos.

The query builder API is designed with so-called fluent interfaces.
This allows more readble code by using method chaining.
It means that not every addition to the query has to be a new statement, with the right indentation, a query can be read in a natural way, for example:

```
query.where
        .add( < condition > )
        .add( < condition > )
        .add( < condition > )
.root.sort.add( < column > ).add( < column > )
```

All elements in the query have a property parent and root to get to the parent or root node of the query object to access properties higher up the node.

Query object structure

A query object can be created via databaseManager.createSelect() or retreived from a foundset via foundset.getQuery()
At toplevel the properties that can be accessed are:

- where
  The where-clause condition of the query
- having
  The having-clause condition of the query
- groupBy
  The group-by-clause condition of the query
- joins
  To add new joins or access existing joins
- columns
  To access colums form the query table
- result
  The selected columns in the query
- sort
  The order-by clause of the query
- params
  Contains query parameters
- or
  Factory property to create a new or-condition
- and
  Factory property to create a new and-condition
- not
  Factory property to create a new not-condition
- exists
  Factory property to create a new exists-condition

The following sections contain examples of how to use these.

Conditions

Conditions are defined on columns, added to the where clause.

```
// select orders for 1 contact
/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
query.where.add(query.columns.contact_id.eq(999))
foundset.loadRecords(query)
```

Conditions can also be named and removed individually.

```
query.where.add('mycond', query.columns.contact_id.eq(999))
query.where.add('flagcond', query.columns.flag.ge(2))
query.where.remove('flagcond')
```

All conditions can be removed:

```
query.where.clear()
```

To find the names of the queries, use the conditionnames property.

```
for (var c in q.where.conditionnames)
{
	if (c != 'mycond') q.where.remove(c)
}
```

You can use relations via the joins property.

```
// select orders, join order_details via relation, find all orders with a detail with quantity > 1
/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
query.where.add(query.joins.orders_to_order_details.columns.quantity.gt(1))
foundset.loadRecords(query)
```

Joins can also be created ad-hoc.

```
// select orders, join order_details inner join, find all orders with a detail with quantity <= 10
/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
/** @type QBJoin<db:/example_data/order_details> */
var join = query.joins.add('db:/example_data/order_details', JSRelation.INNER_JOIN, 'odetail')
join.on.add(join.columns.orderid.eq(query.columns.orderid))
query.where.add(join.columns.quantity.le(10))
foundset.loadRecords(query)
```

In stead of creating a query object from scratch, you can also take the foundset query and modify it

```
// find orders for contact 999 in the current foundset selection
var query = foundset.getQuery();
query.where.add(query.columns.contact_id.eq(999))
foundset.loadRecords(query)
```

Joins can be nested, even on the same table:

```
        // find all grand-children of john, using relations

        /** @type QBSelect<db:/example_data/person> */
        var query = databaseManager.createSelect('db:/example_data/person')
        query.where.add(query.joins.person_to_parent.joins.person_to_parent.columns.name.eq('john'))
        foundset.loadRecords(query)
```

Same with ad-hoc joins:

```
        // find all grand-children of john, creating ad-hoc joins

        /** @type QBSelect<db:/example_data/person> */
        var query = databaseManager.createSelect('db:/example_data/person')
        var join1 = query.joins.add('db:/example_data/person')
        join1.on.add(query.columns.parent_person_id.eq(join1.columns.person_id))
        var join2 = query.joins.add('db:/example_data/person')
        join2.on.add(join1.columns.parent_person_id.eq(join2.columns.person_id))

        query.where.add(join2.columns.name.eq('john'))
        foundset.loadRecords(query)
```

Aggreate conditions are defined on columns.

```
        // aggregate conditions, find orders with no detail
        /** @type QBSelect<db:/example_data/orders> */
        var query = databaseManager.createSelect('db:/example_data/orders')
        query.groupBy.addPk() // have to group by on pk when using having-conditions in (foundset) pk queries
        .root.having.add(query.joins.orders_to_order_details.columns.quantity.count.eq(0))
        foundset.loadRecords(query)
```

Existing aggregates can be removed via clear() and clearHaving():

```
        query.groupBy.clear().root.clearHaving()
```

This example shows more complex conditions with IN, OR, NOT and EXISTS.

```
        // select pk from orders where order_id NOT IN  [1, 2, 3] or exist (select 1 from order_details where
detail.orderid = orders.orderid)
        /** @type QBSelect<db:/example_data/order_details> */
        var subquery = databaseManager.createSelect('db:/example_data/order_details')

        /** @type QBSelect<db:/example_data/orders> */
        var query = databaseManager.createSelect('db:/example_data/orders')
        query.where.add(query
                .or
                        .add(query.columns.order_id.not.isin([1, 2, 3]))

                        .add(query.exists(
                                        subquery.where.add(subquery.columns.orderid.eq(query.columns.order_id)).
root
                        ))
                )

        foundset.loadRecords(query)
```

Sorting

The order by is defined on the sort property of the query.

```
// sort orders related by order_detail quantity descending and then by companyid ascending
/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
query.sort
        .add(query.joins.orders_to_order_details.columns.quantity.desc)
        .add(query.columns.companyid)
foundset.loadRecords(query)
```

Existing sorting can be removed with the clear() method on sort:

```
query.sort.clear()
```

Data Sets

When you get a data set with the query builder you need to specify the columns to be returned via the result property.

```
// select company_id, customerid from orders where contact_id = 100 order by company_id asc
/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
query.result.add(query.columns.company_id).add(query.columns.customerid)
.root.where.add(query.columns.contact_id.eq(100))
.root.sort.add(query.columns.company_id)

var ds = databaseManager.getDataSetByQuery(query, -1)
```

Existing columns can be removed with the clear() method on results:

```
query.result.clear()
```

Parameterized queries

Queries can contain parameterized values.
Note that foundset.loadRecords(query) will create a copy of the query, so setting a parameter after loadRecords() will not affect a previous call.

```
// parametererized queries

/** @type QBSelect<db:/example_data/orders> */
var query = databaseManager.createSelect('db:/example_data/orders')
query.where.add(query.columns.contact_id.eq(query.getParameter('mycontactid')))

// load orders where contact_id = 100
query.params['mycontactid'] = 100
foundset.loadRecords(query)

// load orders where contact_id = 200
query.params['mycontactid'] = 200
foundset.loadRecords(query)
```

Functions and arithmetic operations

A number of functions and arithmetic operations are supported in queries.
A function can be used from a column (query.columns.companyName.upper()), or in a static way via the functions property in query (query.functions.upper (query.columns.companyName)).

For example, the query

```
query.result.add(query.columns.col1).add(query.columns.col2)
        .where.add(query.functions.floor(query.columns.col1.divide(3600)).gt(query.columns.col2)))
```

will result in sql

```
SELECT col1, col2 FROM tab1 WHERE FLOOR(col1 / 3600) > col2
```

Supported arithmetic operators

name	operator
plus	+
minus	-
divide	/
multiply	*

Supported functions

name	
abs	absolute
bit_length	bit length
cast	cast to type, see constants in QUERY_COLUMN_TYPES
ceil	ceil rounding
coalesce	coalesce function
concat	concatenate columns or values
day	extract day from date
floor	floor rounding
hour	extract hour from date
len	length
locate	locate string
lower	string to lower
minute	extract minute from date
mod	modulo function
month	extract month from date
nullif	nullif function
round	rounding
second	extract second from date
sqrt	square root
substring	substring
trim	trim string
upper	string to upper
year	extract year from date