# Database Manager

#### Return Types

JSFoundset JSColumn JSDataSet JSFoundSetUpdater JSTable JSRecord

#### Property Summery

| | |
|---|---|
| Boolean | #nullColumnValidatorEnabled<br>Enable/disable the default null validator for non null columns, makes it possible todo the checks later on when saving, when for example autosave is disabled. |

#### Method Summery

| | |
|---|---|
| Boolean | #acquireLock(foundset, record_index, [lock_name])<br>Request lock(s) for a foundset, can be a normal or related foundset. |
| Boolean | #addTableFilterParam(server_name, table_name, dataprovider, operator, value, [filter_name])<br>Adds a filter to all the foundsets based on a table. |
| Boolean | #commitTransaction()<br>Returns true if a transaction is committed; rollback if commit fails. |
| JSFoundset | #convertFoundSet(foundset, relation)<br>Creates a foundset that combines all the records of the specified one-to-many relation seen from the given parent/primary foundset. |
| JSDataSet | #convertToDataSet(array/ids_string/foundset, [array_with_dataprovider_names])<br>Converts the argument to a JSDataSet, possible use in controller. |
| Boolean | #copyMatchingColumns(src_record, dest_record, [overwrite/array_of_names_not_overwritten])<br>Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names). |
| JSDataSet | #createEmptyDataSet(row_count, columnCount/array_with_column_names)<br>Returns an empty dataset object. |
| Boolean | #getAutoSave()<br>Returns true or false if autosave is enabled or disabled. |
| JSDataSet | #getDataSetByQuery(server_name, sql_query, arguments, max_returned_rows)<br>Performs a sql query on the specified server, returns the result in a dataset. |
| String | #getDataSourceServerName(dataSource)<br>Returns the server name from the datasource, or null if not a database datasource. |
| String | #getDataSourceTableName(dataSource)<br>Returns the table name from the datasource, or null if not a database datasource. |
| String | #getDatabaseProductName(serverName)<br>Returns the database product name as supplied by the driver for a server. |
| JSRecord[] | #getEditedRecords()<br>Returns an array of edited records with outstanding (unsaved) data. |
| JSRecord[] | #getFailedRecords()<br>Returns an array of records that fail after a save. |
| JSFoundset | #getFoundSet(server_name/data_source, [table_name])<br>Returns a foundset object for a specified datasource or server and tablename. |
| Number | #getFoundSetCount(foundset)<br>Returns the total number of records in a foundset. |
| Object[] | #getFoundSetDataProviderAsArray(foundset, dataprovider)<br>Returns a foundset dataprovider (normally a column) as JavaScript array. |
| JSFoundSetUpdater | #getFoundSetUpdater(foundset)<br>Returns a JSFoundsetUpdater object that can be used to update all or a specific number of rows in the specified foundset. |
| Object | #getNextSequence(dataSource\|serverName, [tableName], columnName)<br>Gets the next sequence for a column which has a sequence defined in its column dataprovider properties. |
| String | #getSQL(foundset) |
| String | #getSQL(foundset, includeFilters)<br>Returns the internal SQL which defines the specified (related)foundset. |
| Object[] | #getSQLParameters(foundset) |
| Object[] | #getSQLParameters(foundset, includeFilters)<br>Returns the internal SQL parameters, as an array, that are used to define the specified (related)foundset. |
| String[] | #getServerNames()<br>Returns an array with all the server names used in the solution. |
| JSTable | #getTable(foundset/record/datasource/server_name, [table_name])<br>Returns the JSTable object from which more info can be obtained (like columns). |
| Number | #getTableCount(dataSource)<br>Returns the total number of records(rows) in a table. |
| Object[][] | #getTableFilterParams(server_name, [filter_name])<br>Returns a two dimensional array object containing the table filter information currently applied to the servers tables. |

| | |
|---|---|
| String[] | #getTableNames(serverName)<br>Returns an array of all table names for a specified server. |
| String[] | #getViewNames(serverName)<br>Returns an array of all view names for a specified server. |
| Boolean | #hasLocks([lock_name])<br>Returns true if the current client has any or the specified lock(s) acquired. |
| Boolean | #hasNewRecords(foundset/record, [foundset_index])<br>Returns true if the argument (foundSet / record) has at least one row that was not yet saved in the database. |
| Boolean | #hasRecordChanges(foundset/record, [foundset_index])<br>Returns true if the specified foundset, on a specific index or in any of its records, or the specified record has changes. |
| Boolean | #hasRecords(foundset/record, [qualifiedRelationString])<br>Returns true if the (related)foundset exists and has records. |
| Boolean | #hasTransaction()<br>Returns true if there is an transaction active for this client. |
| Boolean | #mergeRecords(source_record, combined_destination_record, [columnnamesarray_to_copy])<br>Merge records from the same foundset, updates entire datamodel (via foreign type on columns) with destination record pk, deletes source record. |
| void | #recalculate(foundsetOrRecord)<br>Can be used to recalculate a specified record or all rows in the specified foundset. |
| Boolean | #refreshRecordFromDatabase(foundset, index)<br>Flushes the client data cache and requeries the data for a record (based on the record index) in a foundset or all records in the foundset. |
| Boolean | #releaseAllLocks([lock_name])<br>Release all current locks the client has (optionally limited to named locks). |
| Boolean | #removeTableFilterParam(serverName, filterName)<br>Removes a previously defined table filter. |
| void | #rollbackEditedRecords()<br>Rolls back in memory edited records that are outstanding (not saved). |
| void | #rollbackTransaction()<br>Rollback a transaction started by databaseManager. |
| Boolean | #saveData([record])<br>Saves all outstanding (unsaved) data and exits the current record. |
| Boolean | #setAutoSave(autoSave)<br>Set autosave, if false then no saves will happen by the ui (not including deletes!). |
| void | #setCreateEmptyFormFoundsets()<br>Turnoff the initial form foundset record loading, set this in the solution open method. |
| void | #startTransaction()<br>Start a database transaction. |
| Boolean | #switchServer(sourceName, destinationName)<br>Switches a named server to another named server with the same datamodel (Recommended to be used in an onLoad method for a solution). |

**Property Details**

nullColumnValidatorEnabled

Enable/disable the default null validator for non null columns, makes it possible todo the checks later on when saving, when for example autosave is disabled.

**Returns**
Boolean

**Sample**

```
databaseManager.nullColumnValidatorEnabled = false;//disable

//test if enabled
if(databaseManager.nullColumnValidatorEnabled) application.output('null validation enabled')
```

**Method Details**

acquireLock

Boolean **acquireLock**(foundset, record_index, [lock_name])

Request lock(s) for a foundset, can be a normal or related foundset.
The record_index can be -1 to lock all rows, 0 to lock the current row, or a specific row of > 0
Optionally name the lock(s) so that it can be referenced it in releaseAllLocks()

returns true if the lock could be acquired.

**Parameters**
foundset – The JSFoundset to get the lock for
record_index – The record index which should be locked.
[lock_name] – The name of the lock.

**Returns**
Boolean – true if the lock could be acquired.

**Sample**

```
//locks the complete foundset
databaseManager.acquireLock(foundset,-1);

//locks the current row
databaseManager.acquireLock(foundset,0);

//locks all related orders for the current Customer
var success = databaseManager.acquireLock(Cust_to_Orders,-1);
if(!success)
{
  plugins.dialogs.showWarningDialog('Alert','Failed to get a lock','OK');
}
```

addTableFilterParam

Boolean **addTableFilterParam**(server_name, table_name, dataprovider, operator, value, [filter_name])

Adds a filter to all the foundsets based on a table.
Note: if null is provided as the tablename the filter will be applied on all tables with the dataprovider name
returns true if the tablefilter could be applied.

**Parameters**
server_name – The name of the database server connection for the specified table name.
table_name – The name of the specified table.
dataprovider – A specified dataprovider column name.
operator – One of "=, <, >, >=, <=, !=, LIKE, or IN".
value – The specified filter value.
[filter_name] – The specified name of the database table filter.

**Returns**
Boolean – true if the tablefilter could be applied.

**Sample**

```
//best way to call this in a global solution startup method
var success = databaseManager.addTableFilterParam('admin', 'messages', 'messagesid', '>', 10,
'higNumberedMessagesRule')
```

commitTransaction

Boolean **commitTransaction**()

Returns true if a transaction is committed; rollback if commit fails.

**Returns**
Boolean – if the transaction could be committed.

**Sample**

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceld or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
  databaseManager.rollbackTransaction();
}
```

convertFoundSet

JSFoundset **convertFoundSet**(foundset, relation)

Creates a foundset that combines all the records of the specified one-to-many relation seen from the given parent/primary foundset.

**Parameters**
{Object} foundset – The JSFoundset to convert.
{Object} relation – can be a one-to-many relation object or the name of a one-to-many relation

**Returns**
JSFoundset – The converted JSFoundset.

**Sample**

```
// Convert in the order form a orders foundset into a orderdetails foundset,
// that has all the orderdetails from all the orders in the foundset.
var convertedFoundSet = databaseManager.convertFoundSet(foundset,order_to_orderdetails);
// or var convertedFoundSet = databaseManager.convertFoundSet(foundset,"order_to_orderdetails");
forms.orderdetails.controller.showRecords(convertedFoundSet);
```

convertToDataSet

JSDataSet **convertToDataSet**(array/ids_string/foundset, [array_with_dataprovider_names])

Converts the argument to a JSDataSet, possible use in controller.loadRecords(dataset)

**Parameters**

array/ids_string/foundset – The data that should go into the JSDataSet.
[array_with_dataprovider_names] – Array with column names.

**Returns**

JSDataSet – JSDataSet with the data.

**Sample**

```
// converts a foundset pks to a dataset
var dataset = databaseManager.convertToDataSet(foundset);
// converts a foundset to a dataset
//var dataset = databaseManager.convertToDataSet(foundset,['product_id','product_name']);
// converts an object array to a dataset
//var dataset = databaseManager.convertToDataSet(files,['name','path']);
// converts an array to a dataset
//var dataset = databaseManager.convertToDataSet(new Array(1,2,3,4,5,6));
// converts an string list to a dataset
//var dataset = databaseManager.convertToDataSet('4,5,6');
```

copyMatchingColumns

Boolean **copyMatchingColumns**(src_record, dest_record, [overwrite/array_of_names_not_overwritten])

Copies all matching non empty columns (if overwrite boolean is given all columns except pk/ident, if array then all columns except pk and array names). returns true if no error did happen.

NOTE: This function could be used to store a copy of records in an archive table. Use the getRecord() function to get the record as an object.

**Parameters**

src_record – The source record to be copied.
dest_record – The destination record to copy to.
[overwrite/array_of_names_not_overwritten] – true (default false) if everything can be overwritten or an array of names that shouldnt be overwritten.

**Returns**

Boolean – true if no errors happend.

**Sample**

```
for( var i = 1 ; i <= foundset.getSize() ; i++ )
{
	var srcRecord = foundset.getRecord(i);
	var destRecord = otherfoundset.getRecord(i);
	if (srcRecord == null || destRecord == null) break;
	databaseManager.copyMatchingColumns(srcRecord,destRecord,true)
}
//saves any outstanding changes to the dest foundset
controller.saveData();
```

createEmptyDataSet

JSDataSet **createEmptyDataSet**(row_count, columnCount/array_with_column_names)

Returns an empty dataset object.

**Parameters**

row_count – The number of rows in the DataSet object.
columnCount/array_with_column_names – Number of columns or the column names.

**Returns**

JSDataSet – An empty JSDataSet with the initial sizes.

**Sample**

```
// gets an empty dataset with a specifed row and column count
var dataset = databaseManager.createEmptyDataSet(10,10)
// gets an empty dataset with a specifed row count and column array
var dataset2 = databaseManager.createEmptyDataSet(10,new Array ('a','b','c','d'))
```

getAutoSave

Boolean **getAutoSave**()

Returns true or false if autosave is enabled or disabled.

**Returns**

Boolean – true if autosave if enabled.

**Sample**

```
//Set autosave, if false then no saves will happen by the ui (not including deletes!). Until you call saveData
or setAutoSave(true)
//Rollbacks in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.rollbackEditedRecords()
databaseManager.setAutoSave(true)
```

getDataSetByQuery

JSDataSet **getDataSetByQuery**(server_name, sql_query, arguments, max_returned_rows)

Performs a sql query on the specified server, returns the result in a dataset.
Will throw an exception if anything did go wrong when executing the query.

**Parameters**

{String} server_name – The name of the server where the query should be executed.
{String} sql_query – The custom sql.
{Object[]} arguments – Specified arguments or null if there are no arguments.
{Number} max_returned_rows – The maximum number of rows returned by the query.

**Returns**

JSDataSet – The JSDataSet containing the results of the query.

**Sample**

```
//finds duplicate records in a specified foundset
var vQuery =" SELECT companiesid from companies where company_name IN (SELECT company_name from companies group
bycompany_name having count(company_name)>1 )";
var vDataset =databaseManager.getDataSetByQuery(databaseManager.getDataSourceServerName(controller.
getDataSource()), vQuery, null, 1000);
controller.loadRecords(vDataset);

var maxReturnedRows = 10;//useful to limit number of rows
var query = 'select c1,c2,c3 from test_table where start_date = ?';//do not use '.' or special chars in names
or aliases if you want to access data by name
var args = new Array();
args[0] = order_date //or  new Date()
var dataset = databaseManager.getDataSetByQuery(databaseManager.getDataSourceServerName(controller.
getDataSource()), query, args, maxReturnedRows);

// place in label:
// elements.myLabel.text = '<html>'+dataset.getAsHTML()+'</html>';

//example to calc a strange total
global_total = 0;
for( var i = 1 ; i <= dataset.getMaxRowIndex() ; i++ )
{
        dataset.rowIndex = i;
        global_total = global_total + dataset.c1 + dataset.getValue(i,3);
}
//example to assign to dataprovider
//employee_salary = dataset.getValue(row,column)
```

getDataSourceServerName

String **getDataSourceServerName**(dataSource)
Returns the server name from the datasource, or null if not a database datasource.

**Parameters**
{String} dataSource – The datasource string to get the server name from.

**Returns**
String – The servername of the datasource.

**Sample**

```
var servername = databaseManager.getDataSourceServerName(datasource);
```

getDataSourceTableName

String **getDataSourceTableName**(dataSource)
Returns the table name from the datasource, or null if not a database datasource.

**Parameters**
{String} dataSource – The datasource string to get the tablename from.

**Returns**
String – The tablename of the datasource.

**Sample**

```
var tablename = databaseManager.getDataSourceTableName(datasource);
```

getDatabaseProductName

String **getDatabaseProductName**(serverName)
Returns the database product name as supplied by the driver for a server.

NOTE: For more detail on named server connections, see the chapter on Database Connections, beginning with the Introduction to database connections in the Servoy Developer User's Guide.

**Parameters**
{String} serverName – The specified name of the database server connection.

**Returns**
String – A database product name.

**Sample**

```
var databaseProductName = databaseManager.getDatabaseProductName(servername)
```

getEditedRecords

JSRecord[] **getEditedRecords**()
Returns an array of edited records with outstanding (unsaved) data.

NOTE: To return a dataset of outstanding (unsaved) edited data for each record, see JSRecord.getChangedData();
NOTE2: The fields focus may be lost in user interface in order to determine the edits.

**Returns**
JSRecord[] – Array of outstanding/unsaved JSRecords.

**Sample**

```
//This method can be used to loop through all outstanding changes,
//the application.output line contains all the changed data, their tablename and primary key
var editr = databaseManager.getEditedRecords()
for (x=0;x<editr.length;x++)
{
	var ds = editr[x].getChangedData();
	var jstable = databaseManager.getTable(editr[x]);
	var tableSQLName = jstable.getSQLName();
	var pkrec = jstable.getRowIdentifierColumnNames().join(',');
	var pkvals = new Array();
	for (var j = 0; j < jstable.getRowIdentifierColumnNames().length; j++)
	{
		pkvals[j] = editr[x][jstable.getRowIdentifierColumnNames()[j]];
	}
	application.output('Table: '+tableSQLName +', PKs: '+ pkvals.join(',') +' ('+pkrec +')');
	// Get a dataset with outstanding changes on a record
	for( var i = 1 ; i <= ds.getMaxRowIndex() ; i++ )
	{
		application.output('Column: '+ ds.getValue(i,1) +', oldValue: '+ ds.getValue(i,2) +', newValue:
'+ ds.getValue(i,3));
	}
}
//in most cases you will want to set autoSave back on now
databaseManager.setAutoSave(true);
```

getFailedRecords

JSRecord[] **getFailedRecords**()

Returns an array of records that fail after a save.

**Returns**

JSRecord[] – Array of failed JSRecords

**Sample**

```
var array = databaseManager.getFailedRecords()
for( var i = 0 ; i < array.length ; i++ )
{
	var record = array[i];
	application.output(record.exception);
	if (record.exception.getErrorCode() == ServoyException.RECORD_VALIDATION_FAILED)
	{
		// exception thrown in pre-insert/update/delete event method
		var thrown = record.exception.getValue()
		application.output("Record validation failed: "+thrown)
	}
 // find out the table of the record (similar to getEditedRecords)
 var jstable = databaseManager.getTable(record);
 var tableSQLName = jstable.getSQLName();
 application.output('Table:'+tableSQLName+' in server:'+jstable.getServerName()+' failed to save.')
}
```

getFoundSet

JSFoundset **getFoundSet**(server_name/data_source, [table_name])

Returns a foundset object for a specified datasource or server and tablename.

**Parameters**

server_name/data_source – The servername or datasource to get a JSFoundset for.
[table_name] – The tablename of the first param was the servername.

**Returns**

JSFoundset – A new JSFoundset for that datasource.

**Sample**

```
var fs = databaseManager.getFoundSet(controller.getDataSource())
var ridx = fs.newRecord()
var record = fs.getRecord(ridx)
record.emp_name = 'John'
databaseManager.saveData()
```

getFoundSetCount

Number **getFoundSetCount**(foundset)

Returns the total number of records in a foundset.

NOTE: This can be an expensive operation (time-wise) if your resultset is large.

**Parameters**

{Object} foundset – The JSFoundset to get the count for.

**Returns**

Number – the foundset count

**Sample**

```
//return the total number of records in a foundset.
databaseManager.getFoundSetCount(foundset);
```

getFoundSetDataProviderAsArray

Object[] **getFoundSetDataProviderAsArray**(foundset, dataprovider)

Returns a foundset dataprovider (normally a column) as JavaScript array.

**Parameters**

{Object} foundset – The foundset

{String} dataprovider – The dataprovider for the values of the array.

**Returns**

Object[] – An Array with the column values.

**Sample**

```
// returns an array with all order_id values of the specified foundset.
var array = databaseManager.getFoundSetDataProviderAsArray(foundset,'order_id');
```

getFoundSetUpdater

JSFoundSetUpdater **getFoundSetUpdater**(foundset)

Returns a JSFoundsetUpdater object that can be used to update all or a specific number of rows in the specified foundset.

**Parameters**

{Object} foundset – The foundset to update.

**Returns**

JSFoundSetUpdater – The JSFoundsetUpdater for the specified JSFoundset.

**Sample**

```
//1) update entire foundset
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
fsUpdater.setColumn('customer_type',1)
fsUpdater.setColumn('my_flag',0)
fsUpdater.performUpdate()

//2) update part of foundset, for example the first 4 row (starts with selected row)
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
fsUpdater.setColumn('customer_type',new Array(1,2,3,4))
fsUpdater.setColumn('my_flag',new Array(1,0,1,0))
fsUpdater.performUpdate()

//3) safely loop through foundset (starts with selected row)
controller.setSelectedIndex(1)
var count = 0
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
while(fsUpdater.next())
{
        fsUpdater.setColumn('my_flag',count++)
}
```

getNextSequence

Object **getNextSequence**(dataSource|serverName, [tableName], columnName)

Gets the next sequence for a column which has a sequence defined in its column dataprovider properties.

NOTE: For more infomation on configuring the sequence for a column, see the section Auto enter options for a column from the Dataproviders chapter in the Servoy Developer User's Guide.

**Parameters**

dataSource|serverName – The datasource that points to the table which has the column with the sequence, or the name of the server where the table can be found. If the name of the server is specified, then a second optional parameter specifying the name of the table must be used. If the datasource is specified, then the name of the table is not needed as the second argument.

[tableName] – The name of the table that has the column with the sequence. Use this parameter only if you specified the name of the server as the first parameter.

{String} columnName – The name of the column that has a sequence defined in its properties.

**Returns**

Object – The next sequence for the column, null if there was no sequence for that column or if there is no column with the given name.

**Sample**

```
var seqDataSource = forms.seq_table.controller.getDataSource();
var nextValue = databaseManager.getNextSequence(seqDataSource, 'seq_table_value');
application.output(nextValue);

nextValue = databaseManager.getNextSequence(databaseManager.getDataSourceServerName(seqDataSource),
databaseManager.getDataSourceTableName(seqDataSource), 'seq_table_value')
application.output(nextValue);
```

getSQL

String **getSQL**(foundset, includeFilters)

Returns the internal SQL which defines the specified (related)foundset.
Optionally, the foundset and table filter params can be excluded in the sql (includeFilters=false).
Make sure to set the applicable filters when the sql is used in a loadRecords() call.

**Parameters**

{Object} foundset – The JSFoundset to get the sql for.

{Boolean} includeFilters – optional, include the foundset and table filters, default true.

**Returns**

String – String representing the sql of the JSFoundset.

**Sample**

```
var sql = databaseManager.getSQL(foundset)
```

getSQLParameters

Object[] **getSQLParameters**(foundset, includeFilters)

Returns the internal SQL parameters, as an array, that are used to define the specified (related)foundset.

**Parameters**

{Object} foundset – The JSFoundset to get the sql parameters for.
{Boolean} includeFilters – optional, include the parameters for the filters, default true.

**Returns**

Object[] – An Array with the sql parameter values.

**Sample**

```
var sqlParameterArray = databaseManager.getSQLParameters(foundset)
```

### getServerNames

String[] **getServerNames**()

Returns an array with all the server names used in the solution.

NOTE: For more detail on named server connections, see the chapter on Database Connections, beginning with the Introduction to database connections in the Servoy Developer User's Guide.

**Returns**

String[] – An Array of servernames.

**Sample**

```
var array = databaseManager.getServerNames()
```

### getTable

JSTable **getTable**(foundset/record/datasource/server_name, [table_name])

Returns the JSTable object from which more info can be obtained (like columns).
The parameter can be a JSFoundset,JSRecord,datasource string or server/tablename combination.

**Parameters**

foundset/record/datasource/server_name – The data where the JSTable can be get from.
[table_name] – The tablename of the first param is a servername string.

**Returns**

JSTable – the JSTable get from the input.

**Sample**

```
var jstable = databaseManager.getTable(controller.getDataSource());
//var jstable = databaseManager.getTable(foundset);
//var jstable = databaseManager.getTable(record);
//var jstable = databaseManager.getTable(datasource);
var tableSQLName = jstable.getSQLName();
var columnNamesArray = jstable.getColumnNames();
var firstColumnName = columnNamesArray[0];
var jscolumn = jstable.getColumn(firstColumnName);
var columnLength = jscolumn.getLength();
var columnType = jscolumn.getTypeAsString();
var columnSQLName = jscolumn.getSQLName();
var isPrimaryKey = jscolumn.isRowIdentifier();
```

### getTableCount

Number **getTableCount**(dataSource)

Returns the total number of records(rows) in a table.

NOTE: This can be an expensive operation (time-wise) if your resultset is large

**Parameters**

{Object} dataSource – Data where a server table can be get from. Can be a foundset, a datasource name or a JSTable.

**Returns**

Number – the total table count.

**Sample**

```
//return the total number of rows in a table.
var count = databaseManager.getTableCount(foundset);
```

### getTableFilterParams

Object[][] **getTableFilterParams**(server_name, [filter_name])

Returns a two dimensional array object containing the table filter information currently applied to the servers tables.
The "columns" of a row from this array are: tablename,dataprovider,operator,value,tablefilername

**Parameters**
server_name – The name of the database server connection.
[filter_name] – The filter name for which to get the array.
**Returns**
Object[][] – Two dimensional array.
**Sample**

```
var params = databaseManager.getTableFilterParams(databaseManager.getDataSourceServerName(controller.
getDataSource()))
for (var i = 0; params != null && i < params.length; i++)
{
        application.output('Table filter on table ' + params[i][0]+ ': '+ params[i][1]+ ' '+params[i][2]+ '
'+params[i][3] +(params[i][4] == null ? ' [no name]' : ' ['+params[i][4]+']'))
}
```

getTableNames
String[] **getTableNames**(serverName)
Returns an array of all table names for a specified server.
**Parameters**
{String} serverName – The server name to get the table names from.
**Returns**
String[] – An Array with the tables names of that server.
**Sample**

```
//return all the table names as array
var tableNamesArray =databaseManager.getTableNames('user_data');
var firstTableName = tableNamesArray[0];
```

getViewNames
String[] **getViewNames**(serverName)
Returns an array of all view names for a specified server.
**Parameters**
{String} serverName – The server name to get the view names from.
**Returns**
String[] – An Array with the view names of that server.
**Sample**

```
//return all the view names as array
var viewNamesArray =databaseManager.getViewNames('user_data');
var firstViewName = viewNamesArray[0];
```

hasLocks
Boolean **hasLocks**([lock_name])
Returns true if the current client has any or the specified lock(s) acquired.
**Parameters**
[lock_name] – The lock name to check.
**Returns**
Boolean – true if the current client has locks or the lock.
**Sample**

```
var hasLocks = databaseManager.hasLocks('mylock')
```

hasNewRecords
Boolean **hasNewRecords**(foundset/record, [foundset_index])
Returns true if the argument (foundSet / record) has at least one row that was not yet saved in the database.
**Parameters**
foundset/record – The JSFoundset or JSRecord to test.
[foundset_index] – The record index in the foundset to test (not specified means has the foundset any new records)
**Returns**
Boolean – true if the JSFoundset has new records or JSRecord is a new record.

**Sample**

```
var fs = databaseManager.getFoundSet(databaseManager.getDataSourceServerName(controller.getDataSource()),
'employees');
        databaseManager.startTransaction();
        var ridx = fs.newRecord();
        var record = fs.getRecord(ridx);
        record.emp_name = 'John';
        if (databaseManager.hasNewRecords(fs)) {
                application.output("new records");
        } else {
                application.output("no new records");
        }
        databaseManager.saveData();
        databaseManager.commitTransaction();
```

hasRecordChanges

Boolean **hasRecordChanges**(foundset/record, [foundset_index])

Returns true if the specified foundset, on a specific index or in any of its records, or the specified record has changes.

NOTE: The fields focus may be lost in user interface in order to determine the edits.

**Parameters**

foundset/record – The JSFoundset or JSRecord to test if it has changes.
[foundset_index] – The record index in the foundset to test (not specified means has the foundset any changed records)

**Returns**

Boolean – true if there are changes in the JSFoundset or JSRecord.

**Sample**

```
if (databaseManager.hasRecordChanges(foundset,2))
{
        //do save or something else
}
```

hasRecords

Boolean **hasRecords**(foundset/record, [qualifiedRelationString])

Returns true if the (related)foundset exists and has records.

**Parameters**

foundset/record – A JSFoundset to test or a JSRecord for which to test a relation
[qualifiedRelationString] – The relationname if the first param is a JSRecord.

**Returns**

Boolean – true if the foundset/relation has records.

**Sample**

```
if (elements.customer_id.hasRecords(orders_to_orderitems))
{
        //do work on relatedFoundSet
}
//if (elements.customer_id.hasRecords(foundset.getSelectedRecord(),'orders_to_orderitems.
orderitems_to_products'))
//{
//        //do work on deeper relatedFoundSet
//}
```

hasTransaction

Boolean **hasTransaction**()

Returns true if there is an transaction active for this client.

**Returns**

Boolean – true if the client has a transaction.

**Sample**

```
var hasTransaction = databaseManager.hasTransaction()
```

mergeRecords

Boolean **mergeRecords**(source_record, combined_destination_record, [columnnamesarray_to_copy])

Merge records from the same foundset, updates entire datamodel (via foreign type on columns) with destination record pk, deletes source record.
Do use a transaction!
returns true if successful.

NOTE: For more information on foreign types, see Properties options: Details in the Dataproviders chapter of the Servoy Developer User's Guid

**Parameters**
source_record – The source JSRecord to copy from.
combined_destination_record – The target/destination JSRecord to copy into.
[columnnamesarray_to_copy] – The column names Array that should be copied.

**Returns**
Boolean – true if the records could me merged.

**Sample**

```
databaseManager.mergeRecords(foundset.getRecord(1),foundset.getRecord(2));
```

### recalculate

void **recalculate**(foundsetOrRecord)

Can be used to recalculate a specified record or all rows in the specified foundset.
May be necessary when records are inserted in a program external to Servoy.

**Parameters**
{Object} foundsetOrRecord – JSFoundset or JSRecord to recalculate.

**Returns**
void

**Sample**

```
// recalculate one record from a foundset.
databaseManager.recalculate(foundset.getRecord(1));
// recalculate all records from the foundset.
// please use with care, this can be expensive!
//databaseManager.recalculate(foundset);
```

### refreshRecordFromDatabase

Boolean **refreshRecordFromDatabase**(foundset, index)

Flushes the client data cache and requeries the data for a record (based on the record index) in a foundset or all records in the foundset.
Used where a program external to Servoy has modified the database record.
Record index of -1 will refresh all records in the foundset and 0 the selected record.

**Parameters**
{Object} foundset – The JSFoundset to refresh
{Number} index – The index of the JSRecord that must be refreshed (or -1 for all).

**Returns**
Boolean – true if the refresh was done.

**Sample**

```
//refresh the second record from the foundset.
databaseManager.refreshRecordFromDatabase(foundset,2)
//flushes all records in the related foundset  (-1 is or can be an expensive operation)
databaseManager.refreshRecordFromDatabase(order_to_orderdetails,-1);
```

### releaseAllLocks

Boolean **releaseAllLocks**([lock_name])

Release all current locks the client has (optionally limited to named locks).
return true if the locks are released.

**Parameters**
[lock_name] – The lock name to release.

**Returns**
Boolean – true if all locks or the lock is released.

**Sample**

```
databaseManager.releaseAllLocks('mylock')
```

### removeTableFilterParam

Boolean **removeTableFilterParam**(serverName, filterName)

Removes a previously defined table filter.

**Parameters**
{String} serverName – The name of the database server connection.
{String} filterName – The name of the filter that should be removed.
**Returns**
Boolean – true if the filter could be removed.
**Sample**

```
var success = databaseManager.removeTableFilterParam('admin', 'higNumberedMessagesRule')
```

rollbackEditedRecords

void **rollbackEditedRecords**()

Rolls back in memory edited records that are outstanding (not saved).
Best used in combination with the function databaseManager.setAutoSave()
This does not include deletes, they do not honor the autosafe false flag so they cant be rollbacked by this call.
**Returns**
void
**Sample**

```
//Set autosave, if false then no saves will happen by the ui (not including deletes!). Until you call saveData
or setAutoSave(true)
//Rollbacks in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.rollbackEditedRecords()
databaseManager.setAutoSave(true)
```

rollbackTransaction

void **rollbackTransaction**()

Rollback a transaction started by databaseManager.startTransaction().
Will also rollback deletes that are done between start and a rollback call that are not handled by autosave false and rollbackEditedRecords()
**Returns**
void
**Sample**

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceld or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
  databaseManager.rollbackTransaction();
}
```

saveData

Boolean **saveData**([record])

Saves all outstanding (unsaved) data and exits the current record.
Optionally, by specifying a record, can save a single record instead of all the data.

NOTE: The fields focus may be lost in user interface in order to determine the edits.
**Parameters**
[record] – The JSRecord to save.
**Returns**
Boolean – true if the save was done without an error.
**Sample**

```
databaseManager.saveData();
//databaseManager.saveData(foundset.getRecord(1));//save specific record
```

setAutoSave

Boolean **setAutoSave**(autoSave)

Set autosave, if false then no saves will happen by the ui (not including deletes!).
Until you call databaseManager.saveData() or setAutoSave(true)

If you also want to be able to rollback deletes then you have to use databaseManager.startTransaction().
Because even if autosave is false deletes of records will be done.

**Parameters**
{Boolean} autoSave – Boolean to enable or disable autosave.

**Returns**
Boolean – false if the current edited record could not be saved.

**Sample**

```
//Rollbacks in mem the records that were edited and not yet saved. Best used in combination with autosave false.
databaseManager.setAutoSave(false)
//Now let users input data

//On save or cancel, when data has been entered:
if (cancel) databaseManager.rollbackEditedRecords()
databaseManager.setAutoSave(true)
```

### setCreateEmptyFormFoundsets

void **setCreateEmptyFormFoundsets**()

Turnoff the initial form foundset record loading, set this in the solution open method.
Simular to calling foundset.clear() in the form's onload event.

NOTE: When the foundset record loading is turned off, controller.find or controller.loadAllRecords must be called to display the records

**Returns**
void

**Sample**

```
//this has to be called in the solution open method
databaseManager.setCreateEmptyFormFoundsets()
```

### startTransaction

void **startTransaction**()

Start a database transaction.
If you want to avoid round trips to the server or avoid the posibility of blocking other clients
because of your pending changes, you can use databaseManager.setAutoSave(false/true) and databaseManager.rollbackEditedRecords().

startTransaction, commit/rollbackTransacton() does support rollbacking of record deletes which autoSave = false doesnt support.

**Returns**
void

**Sample**

```
// starts a database transaction
databaseManager.startTransaction()
//Now let users input data

//when data has been entered do a commit or rollback if the data entry is canceld or the the commit did fail.
if (cancel || !databaseManager.commitTransaction())
{
  databaseManager.rollbackTransaction();
}
```

### switchServer

Boolean **switchServer**(sourceName, destinationName)

Switches a named server to another named server with the same datamodel (Recommended to be used in an onLoad method for a solution).
return true if successfull.

**Parameters**
{String} sourceName – The name of the source database server connection
{String} destinationName – The name of the destination database server connection.

**Returns**
Boolean – true if the switch could be done.

**Sample**

```
//dynamically changes a server for the entire solution, destination database server must contain the same tables
/columns!
//will fail if there is a lock, transaction , if repository_server is used or if destination server is invalid
//in the solution keep using the sourceName every where to reference the server!
var success = databaseManager.switchServer('crm', 'crm1')
```