The Servoy Foundset

In This Chapter

The Servoy Foundset is a developer's window into Servoy's Data Binding layer. A single foundset always maps to a single database table (or view) and is responsible for reading from and writing to that table. From the user interface, a foundset controls which records are loaded and displayed, as well as how records are created, edited and deleted. From the developer's perspective, a foundset is a programmable object with specific behaviors and run-time properties that provide a high-level abstraction to facilitate low-level data operations.

For all programming reference information, see the JSFoundSet API documention in the reference guide.

Forms Bound to a Foundset

A Servoy Form is typically bound to a single database table and the form will always contain a single foundset which is bound to the same table. Much of the action in the user interface, such as a user editing data fields, directly affects the form's foundset. Conversely, actions taken on the foundset, such as programmatically editing data, is immediately reflected in the form.

While a form is always bound to a foundset, a foundset may be used by zero or more forms. Foundsets can be created and used by a programmer to accomplish many tasks.

Often, there can be several different forms which are bound to the same table. In most cases the forms will share the same foundset and thus provide a unified view. For example, imagine a form showing a list of customer records, where clicking on one of the records switches to another form showing a detailed view of only the selected record. The forms will automatically stay in sync and there is no need to coerce the forms to show the same record. Exceptions to this behavior include scenarios where forms are shown through different Relations, or have been explicitly marked to use a separate foundset.

See also the namedFoundset property of a form.

Loading Records

One of the primary jobs of a foundset is to load records from the table to which it is bound. A foundset is always based on an underlying SQL query, which may change often during the lifetime of the foundset. However the query will always take the form of selecting the Primary Key column(s) from the table and will also always include an `ORDER BY` clause, which in its simplest form will sort the results based on the Primary Key column(s).

Foundset Loading

```
SELECT customerid FROM customers ORDER BY customerid ASC
```

After retrieving the results for Primary Key data, the foundset will issue subsequent SQL queries to load the matching record data in smaller, optimized blocks. This query happens automatically in an on-demand fashion to satisfy the foundset's scrollable interface.

Example: Record loading query

```
SELECT * FROM customers WHERE customerid IN (?,?,?,?,?,?,?,?) ORDER BY customerid ASC
```

A foundset's underlying query can change dramatically throughout the client session. The following events will modify a foundset's underlying query:

- When a form to which it is bound is loaded
- When the loadRecords method is called programmatically
- When the sort definition is changed
- When it exits find mode

⚠ See also the Database Manager's getSQL and getSQLParameters methods

Loading Records Programmatically

The `loadRecords` method is used to directly modify the underlying query that loads PK data. There are several uses.

Load by a Single PK

This is the simplest approach, which loads a single recordy by its primary key value.

```
foundset.loadRecords(123);
```

Load by PK Data Set

This approach simply dictates that a foundset will load records based on specified primary key data.

```
var ids = [1,2,3,6,9];                                // an array of record PKs
var ds = databaseManager.convertToDataSet(ids); // convert the ids to a JSDataset
foundset.loadRecords(ds);                          // load records
```

⚠ Notice the array was converted first to a JSDataset object. This object, which is like a 2-dimensional array, is used to provide support for composite primary keys.

Load by Another Foundset

This approach is useful to essentially copy the query of another foundset.

```
foundset.loadRecords(anotherFoundset);
```

Load by Query

This approach allows a SQL query fragment to be used to set the foundset's underlying query. There are certain restriction on the form that a query can take. For obvious reasons, the query must return the primary key column(s) from the table to which the foundset is bound. For a full description see the reference guide.

```
var sql = 'select id from my_table where my_table.column1 in ?,?,?;
var args = [1,2,3];
foundset.loadRecords(sql, args);
```

⚠ See also the loadRecords API in the reference guide for complete usage options.

Sorting

All foundsets contain a sorting definition that determines the order in which records are loaded and displayed. Sorting is always expressed in the *ORDER BY* clause of a foundset's query and thus handled by the database.

A foundset's sorting definition is encapsulated in a String property, which can be programmatically read using the getCurrentSort method, and written using the sort method.

Parameters for this property include an ordered list of one or more data providers, each of which having a sort direction, either ascending or descending. The string takes a form such that each data provider and its sort direction are separated by white space. The direction is abbreviated either *asc* or *desc*. Multiple data providers are separated by commas.

Example: Sort String Format

```
'column1 asc, column2 desc'        // Sort on column1 ascending, then column2 desceding
```

The order of the data providers determines their relative priority when sorting, such that when two records contain the same value for a higher priority data provider, the sorting will be deferred the next lowest priority data provider.

Example: The following sort string will sort, first on last name, and second on first name.

```
foundset.sort('last_name asc, first_name asc');
```

The result is that all records are sorted by last name. But in the case where the last names are the same, then the first name is used.
|| last_name || first_name ||

Sloan	Zachary
Smith	Jane
Smith	Jon
Snead	Aaron

Available Data Provider Types

The following data provider types may be used as sort criteria:

- Any Column
- Any Related Column
- Any Related Aggregate

Example: Sort a customers foundset based on the number of orders each customer has, in this case a related aggregation.

```
SELECT customers.customerid FROM customers
INNER JOIN orders ON customers.customerid=orders.customerid
GROUP BY customers.customerid ORDER BY count(orders.orderid) ASC
```

Results in the following query:

```
SELECT customers.customerid FROM customers
INNER JOIN orders ON customers.customerid=orders.customerid
GROUP BY customers.customerid ORDER BY count(orders.orderid) ASC
```

Sorting on related columns and aggregates changes is simple and powerful. However this changes the nature of the foundset's query. One should be advised of this and ensure that the database is tuned accordingly.

Scrolling Result Set

The foundset maintains a scrollable interface for traversing record data. This interface includes a numeric index for every record that is returned by the foundset's query.

Foundset Size

The foundset also has a `size` property, which indicates the number of records that are indexed by the foundset at any given time. Because the foundset's SQL query may eventually return thousands or millions of results, the initial size of the foundset has a maximum of 200. This value can grow dynamically, in blocks of 200, as the foundset is traversed.

Selected Index

The foundset maintains a selected index, a cursor with which to step through the records. If the selected index equals or exceeds the size of the foundset, the foundset will automatically issue another query to load the next batch of primary key data. Thus the foundset loads record data and grows dynamically with the changing Selected Index property. There are two methods used to get/set the foundsets selected index. They are getSelectedIndex and setSelectedIndex respectively.

Example: In the example below, note that the foundset's size changes after the selected index has changed. The foundset's cache grows dynamically

```
// Foundset size grows dynamically as the foundset is traversed
foundset.getSize(); // returns 200
foundset.setSelectedIndex(200);
foundset.getSize(); // returns 400 because the foundset loaded the next 200 record pks
```

Iterating Over a Foundset

Often, as part of some programming operation, it is necessary to iterate over a part or all of a foundset.

There are several approaches to iterating: using the foundset iterator, changing the selected index of the foundset, accessing a record object, accessing data provider values as an array.

While the last three iterating options are more intuitive, and also vary with regards to performance and usage, the foundset iterator is the most recommended to be used since it is the only option that ensures iterating over all the records of the foundset, without missing any of them due to the multiple clients performing changes on the same foundset at the same time.

It is also possible to use JSFoundsetUpdater API to iterate over and update a foundset, though iterating is not its main goal.

Using the Foundset Iterator

Sometimes there is more than one user working on the same foundset, possibly inserting or deleting records. When iterating on a foundset, it needs to be ensured that the loop neither skips nor processes twice any record due to the foundset modifications occurred from other clients. Thus, in such cases, a secure iterator is needed to perform the iteration on the foundset.

The `forEach` method does exactly that, iterating over **all** the records of a foundset and calling the callback method given as parameter for each one of them.

Example This is an example of how to use the `forEach` method for iterating over a foundset.

```
foundset.loadAllRecords();
foundset.forEach(
                        /**
                         * @param {JSRecord} record
                         * @param recordIndex
                         * @param {JSFoundset} fs
                         */
                        function(record, recordIndex, fs) {
                                application.output("record processed: " + record + ", record index: " +
recordIndex);
                        }
                );
```

See also the JSFoundset's `forEach` method.

Changing the Selected Index

Perhaps the most intuitive approach is to programmatically change the foundset's selected index property.

Example: The example below iterates over the **entire** foundset using a for loop.

```
for(var i = 1; i <= foundset.getSize(); i++){
        foundset.setSelectedIndex(i);
        // operate on the selected record
}
```

See also the JSFoundset's setSelectedIndex method.

Accessing a Record Object

While setting the selected index of the foundset is sometimes necessary, it also contains some overhead and therefore is not always the most efficient way to iterate over a foundset. However, one can iterate in a similar manner, access a record object without changing the selected index of a foundset by using the `getRecord` method of the foundset.

Example This example iterates over the foundset, but does not affect the selected index. The performance will be better than the previous example, and will not have any side effects in the UI if the foundset is bound to a form.

```
for(var i = 1; i <= foundset.getSize(); i++){
        var rec = foundset.getRecord(i);        // does not affect the selected index
}
```

See also the JSFoundset's getRecord method.

Accessing Data Provider Values As an Array

Sometimes the purpose of iterating over a foundset is to access all the values for a particular data provider. The most efficient way to do this is to obtain an array of values for the foundset's data provider using the `getFoundSetDataProviderAsArray` method of the `databaseManager` API.

Example This example shows how to access all the values in a foundset for a single data provider. Iterating over a simple array offers better performance over normal foundset iteration.

```
var ids = databaseManager.getFoundSetDataProviderAsArray(foundset,'order_id');
for(i in ids){
    var id = ids[i];
}
```

See also the JSFoundset's getFoundSetDataProviderAsArray method.

Related Foundsets

Foundsets are often constrained or filtered by a Relation. In this situation, the foundset is said to be a Related Foundset and its default SQL query will include in its `WHERE` clause, the parameters by which to constrain the foundset.

It is important to make the distinction that a relation and a foundset are not one in the same. Rather, a relation name is used to reference a specific foundset object within a given context. The context for a related foundset is always a specific record object. But for convenience, related foundsets may be referenced within a form's scripting scope and as a property of any foundset. However in these cases, the context is always implied to be the **selected record** in the context.

For example:

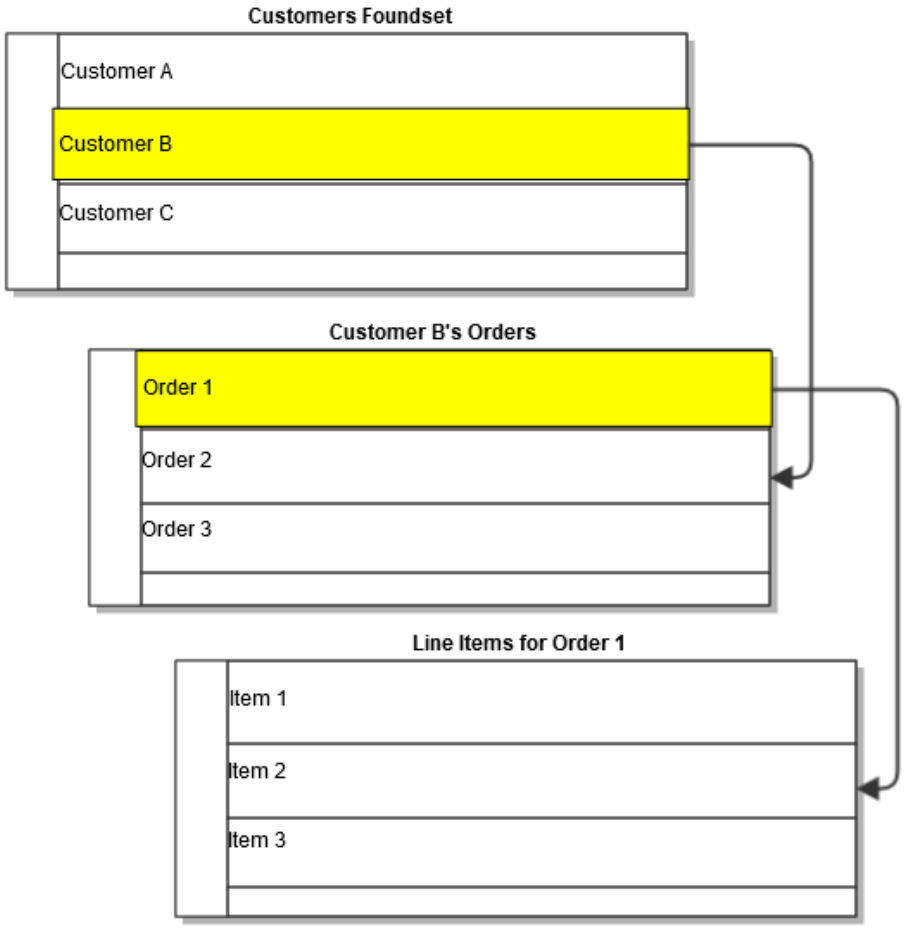
Take a predefined Relation, customers_to_orders, which models a one-to-many relationship between a customers table and an orders table. The following three lines of code, executed within the scripting scope of a form based on the customers table, all produce the same result.

```
// Returns the number of orders for the selected customer record in this form's foundset
customers_to_orders.getSize();

// ...the same as:
foundset.customers_to_orders.getSize();

// ...also the same as:
foundset.getSelectedRecord().customers_to_orders.getSize();
```

Related foundsets can be chained together using relation names. Again, the shorthand implies the context of the **selected record** for each foundset.

Example

```
// Returns the number of order details for the selected order record of the selected customer:
customers_to_orders.orders_to_order_details.getSize();

// ...is the same as:
customers_to_orders.getSelectedRecord().orders_to_order_details.getSelectedRecord().getSize();
```



Foundsets and Data Broadcasting

A foundset may be automatically updated when the client receives a Data Broadcast Event . If the data change affected the table to which the foundset is bound, the foundset will be refreshed to reflect the change.

Performing Batch Updates

Foundsets are typically updated on a record-by-record basis, either as the user operates on a foundset-bound GUI component, or through programmatic interactions. However, sometimes it is necessary to perform an update to an entire foundset. For performance reasons, it is not advised that this be done by programmatically iterating over the foundset's records. Rather, it is recommended that batch updates be performed using the JS FoundsetUpdater API.

The Foundset Updater API is ideal to use for the following situations:

Updating an Entire Foundset

This essentially has the effect of issuing a SQL UPDATE statement using the WHERE clause that constrains the foundset. This presents a significant performance advantage over updating records individually. In the example below, a related foundset is updated, meaning all orders belonging to the selected customer will be affected. **Please note:** This method will not trigger any associated Table Events or modification columns.

```
var fsUpdater = databaseManager.getFoundSetUpdater(customers_to_orders);
fsUpdater.setColumn('status',101);
fsUpdater.performUpdate();
```

Updating a Partial Foundset with Different Values for Each Record

The Foundset Updater API can also be used to update part of a foundset. Moreover, unlike the above example, this approach allows for different values for each record. In the example below, the first 4 records (starting from the selected index) are updated by specifying an array of values for each column that is affected.

```
//        update first four records
var fsUpdater = databaseManager.getFoundSetUpdater(foundset);
fsUpdater.setColumn('customer_type',[1,2,3,4]);
fsUpdater.setColumn('my_flag',new [1,0,1,0]);
fsUpdater.performUpdate();
```

⚠ When using this approach, it matters what the selected index of the foundset is. The update will start with this record.