Working with Data

In This Chapter

Database Connectivity

At its core, Servoy is a comprehensive platform to develop database-driven applications. As such, Servoy allows developers to connect to any standard Relational Database Management System (RDBMS). To achieve this, Servoy employs Java Database Connectivity (JDBC) technology. JDBC is a connectivity standard that allows Java-based applications to interact transparently with any database vendor that provides a compliant driver.

Servoy's Query Engine

The Servoy platform uses Structured Query Language (SQL), a standard communications protocol to issue requests to databases. The traditional development of database applications typically requires advanced SQL knowledge to adequately and efficiently retrieve data. Moreover, database vendors often adapt their own "flavors" of SQL, making database portability an issue. If an application uses non-standard SQL expressions, it cannot be easily deployed against databases other the the one for which it was developed.

The Servoy platform obviates the need for developers to write their own SQL in almost all scenarios. Instead Servoy dynamically generates all the SQL required to read and write data for all but the most complex scenarios. Servoy's generated queries are guaranteed to be optimized and database-neutral. At the same time, the Servoy platform is open and allows developers with the knowledge and preference for writing SQL to do so as well.

Named Connections

Developers will specify a *Server Name*, which maps to a specific database connection configuration (i.e. user, password, URL, etc). At run time, Servoy will automatically create a pool of connections for a given Server Name, which will be reused for the duration of the application server up time. Developers are always insulated from the complexities of connecting to the database.

The details of a connection configuration are stored as part of the development or deployment environment and are never part of the code base. This allows the database connection to be changed for a given context without making modifications to an application's code base.

For example, it is common to have separate databases for development/testing and production. Therefore, a Server Name could resolve to a test database in both Servoy Developer and an instance of Servoy Server used for staging. The same Server Name would resolve to a production database for an instance of Server Server used in production. Another example is for multiple, on-premise, deployments where local instances of Servoy Server rely on local database connections.

Switching Connections

While database connections can be changed transparently between different development and deployment contexts, they can also be changed within the same context. Servoy's API provides a means for developers to change, at run time, from one Server Name to another. For example, different application user groups may be required to use different connections to the same database. In another example, different customers may have their data stored in their own separate database. In both these cases, a specific Server Name is identified and used for an individual client session.

Connection Pooling

Servoy uses database connection pooling technology which provides significant benefits in terms of application performance, concurrency and scalability. Database connections are often expensive to create because of the overhead of establishing a network connection and initializing a database connection session in the back end database. Moreover, the ongoing management of all of a database's connection sessions can impose a major limiting factor on the scalability of an application. Valuable database resources such as locks, memory, cursors, transaction logs, statement handles and temporary tables all tend to increase based on the number of concurrent connection sessions. This limitation is overcome using Connection Pooling, whereby a limited number of connections is shared by a larger number of clients. When a client makes a new request for data, the application server briefly borrows a connection from the pool to issue the query, then returns it to the pool. In this manner an application can scale well beyond the limits of the database without compromising performance. Servoy's connection pools are configurable so that connectivity can be optimized to suit applications of various sizes.

Dynamic Data Binding

The Servoy platform provides a Graphical User Interface (GUI), as well as an Application Program Interface (API) which dynamically bind to database resources. This means the Servoy Application Server will dynamically generate and issue the SQL required to Read, Insert, Update and Delete database records in response to both the actions of the user and the behest of the developer.

In the simplest example, a user navigates to a form (which is bound to a specific database table) showing customer data. When the form shows, the Servoy Application Server issues a SQL *select* statement on the customer database table, retrieves the results of the query and displays them in the form.

When a user types a value into a text field (which is bound to a specific column of the database table) and clicks out, the Servoy Application Server issues a SQL *update* command to the database to modify the *selected record*. The resulting change is also broadcast to all other connected clients.

In another example a user may click a button, initiating a script written by a developer using Servoy's data API, a high-level abstraction to perform low-level database operations. The developer need only write code to interact with the API. The Servoy Application Server will translate the instructions into the raw SQL needed to perform the action.

The fundamental unit of data binding in both the GUI and the API is the Foundset object.

Client Cache

A Servoy client instance keeps track of which database records are in use. This is called the Client Cache and it optimizes performance by reducing the number of queries made to the database. Records are tracked by primary key. The first time the contents of a record are accessed, the Application Server must issue a query to the database on behalf of the client. The values for all of the columns of the record object are held in memory and therefore, subsequent access of the record will not produce anymore queries to the database. The user experience is greatly enhanced as one can browse quickly between forms and cached records.
A record may fall out of the cache gracefully to conserve memory and is automatically reloaded the next time it is accessed. This happens at the discretion of the client's caching engine, which is highly optimized. Relieved of the burden of managing memory, the developer can focus on more important things.

Data Broadcasting

What happens to the cache when the contents of a record are changed by another Servoy client session?

Fortunately, the Servoy Application Server issues a Data Broadcast Event to all clients whenever a record is inserted, updated or deleted by another Servoy Client. This notification prompts each client to automatically update its cache providing the end users a shared, real-time view of the data.

In a simple example, two remote users are looking at the same *customer* record. The first user modifies the customer's name and commits the change. The second user immediately sees the change updated in his/her client session.

This functionality is provided by default for all Servoy client types. There is nothing that a developer needs to do to enable it. However, the developer may augment the default functionality by implementing the Solution's onDataBroadcast event handler and invoking specific business logic.

Updating the Client Cache

What happens when data is changed outside of any Servoy client sessions?

If a data change originates from another application, client caches may become "stale" for the affected records, meaning that the cached values are not in sync with the last values stored to the database. This is most likely to happen if an **existing** record has already been cached prior to being **updated** by another application. It may also happen if records are added or deleted. However, the duration of the problem will be shorter for inserts and deletes than for updates. This is because while foundsets may reload primary keys periodically as the user navigates the client session, the contents of a record can remain cached indefinitely.

Fortunately, Servoy's APIs provides several opportunities to programmatically update client caches. The best approach depends on the situation.

Flush All Client Caches

This approach will re-cache **all** records in **all** client caches for a specified database table. All foundsets based on the specified table will reissue their queries and all cached records will be refreshed from the database. This approach is the most comprehensive and therefore, most expensive in terms of performance. This approach is ideal to use when:

- External changes were made using the Raw SQL Plugin to update a specific database table. (This plugin bypasses the Data Binding layer and will not be reflected in the client cache.
- External changes are known to have been made on a specified table because a another application or service was invoked from Servoy client session.
- External changes may have been made to a specific table by another application, but it is not known for sure. In this situation it may be ideal to periodically update the client caches using a Headless Client , which is a server-side client session that can perform automated, scheduled operations.

> For more information, see the flushAllClientsCache method in the programming reference guide.

Notify Data Change

This approach essentially sends a Data Broadcast Event to all clients. Clients are informed of changes to specific records, just as if those records were modified from within the Servoy environment. This approach is more granular than updating the entire cache for a specific table and should also yield better performance. This approach is ideal to use when:

- External changes were made using the Raw SQL Plugin to update a specific database records and the primary keys are known.
- Another application or service was invoked from Servoy client session affecting change to specific records who's primary keys are known.

> This approach can also be used to allow Servoy's cache to be updated proactively from another application via a simple web service call. Any method can be exposed as a RESTful web service. Therefore, it is simple to create service that allows another application to inform Servoy of data changes. For more information see the documentation on Servoy's RESTful Web Services Plugin .

> For more information, see the notifyDataChange method in the programming reference guide.

Refresh Record From Database

This approach refreshes the cache for a single record or an entire foundset in the calling client only. Therefore, unlike the previous two approaches, it does not affect the cache of all clients. This approach is ideal to use when:

- External changes may have been made which affect a record or foundset.
- It is not desirable to update the cache for other clients.

For more information see the refreshRecordsFromDatabase method in the programming reference guide.

Data Transactions in Servoy

Data manipulations in Servoy happen inside an *in-memory* transaction. When a record is created or modified, either by user action or by developer, nothing is committed to the database immediately. The Servoy Client tracks all newly-created and modified records, including which columns have changed, their former and latter values. As records are added or modified, the amount of information stored in the In-Memory transaction accrue until they are committed or *rolled back*. The duration of this In-Memory transaction can be short or long depending on the client's configurable *Auto Save* setting.

Auto Save: ON

By default, every Servoy client is started with the Auto Save setting initialized to on/true. This means that the In-Memory transaction is typically very short as changes are committed automatically as the user navigates the client session. Specific actions like clicking in a form's area, navigating to a different form, clicking a button, etc. all trigger a save event. Auto Save is ideal for situations where the user is intended to be able to make edits freely.

Auto Save: OFF

The developer may optionally set the Auto Save setting to off/false. This means that the length of the In-Memory transaction is controlled by the developer. As changes accrue, they are never committed until the developer programmatically invokes a save event. It is ideal to disable Auto Save for scenarios where the user is intended to perform edits in a controlled situation where a group of edits may be saved or rolled back all together.

The Auto Save setting can be programmatically changed throughout the duration of the client session to accommodate different modes for different editing scenarios.

See also the Database Manager's setAutoSave method in the programming reference guide.

The Anatomy of the In-Memory Transaction

Servoy provides a robust data API, giving the developer full access to the In-Memory transaction, which consists of a listing of all record objects that were added or modified. For each of these record objects, there is a listing of every column whose value was changed. For every modified column, there is a reference to the value before and after the edit. The transaction API also allows developers to distinguish between records that are newly-created and do not yet exist in the database, versus records that already exist in the database, but have outstanding edits.

See also the Database Manager's getEditedRecords and the JSRecord's getChangedData methods.

Saving Data Changes

A developer can programmatically issue a save event, causing the contents of the In-Memory transaction to be automatically translated into instructions to insert/update database tables. A developer can optionally invoke a save event for a specific record only, leaving the rest of the transaction unaffected.

If for some reason one or more records were unable to be saved (i.e. due to a back-end database violation, etc.), the transaction will also keep track of *Fail ed Records* and their associated errors.

See also the Database Manager's saveData and getFailedRecords methods, as well as the exception property of the JSRecord

Rolling Back Data Changes

A developer can programmatically issue a command to *rollback* the contents of the entire In-Memory transaction, causing newly created records to be removed and modified records to be reverted to their state prior to the start of the In-Memory transaction. The developer can optionally choose to rollback changes for a specific record, leaving the rest of the transaction unaffected.

See also the Database Manager's rollbackEditedRecords method, as well as the rollbackChanges method of the JSRecord.

What about Deleting Records?

It is important to note that record deletes are not part of the In-Memory transaction. When a record is deleted, the instructions are sent to the database immediately and the delete cannot be rolled back.

See also the deleteRecord method of the JSFoundset.

The Servoy Foundset

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.

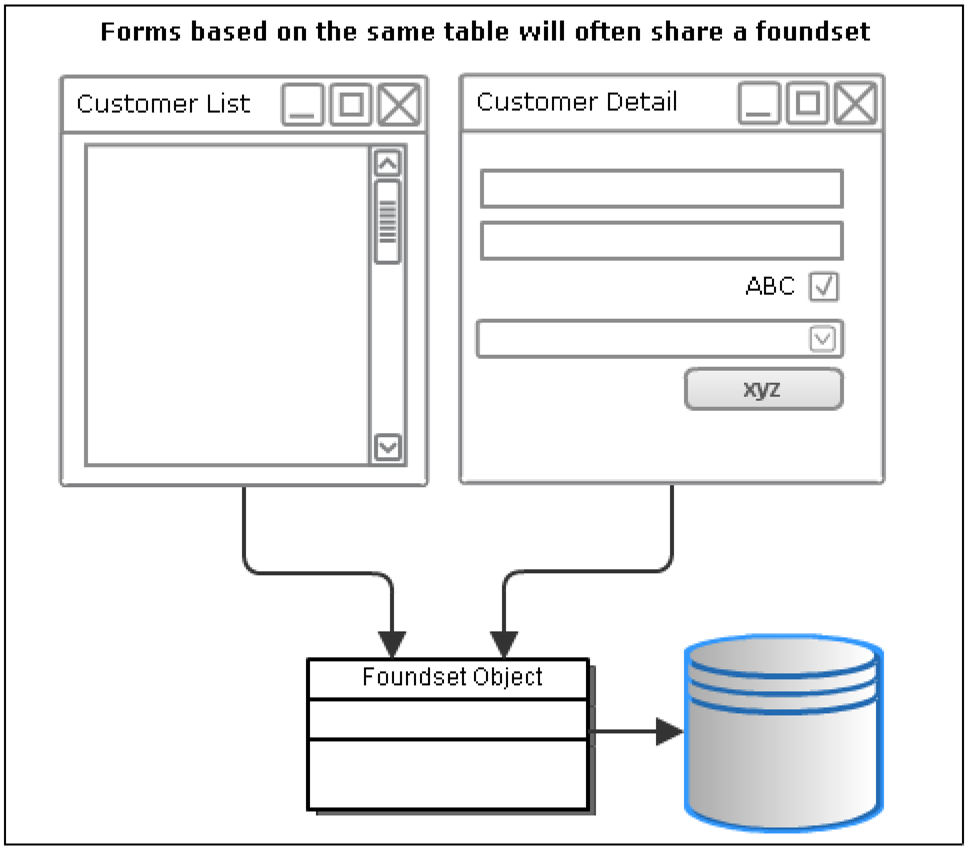
Forms Bound to a Foundset

A Servoy Form is typically bound to a single database table and the form will always contain a single Foundset object 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 object 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.

```
foundset.sort('customers_to_orders.order_count 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 part or all of a foundset. There are several approaches to iterating, each having their appropriate usage. General a Javascript *for* or *while* statement is used to control the flow of execution.

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 of 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:

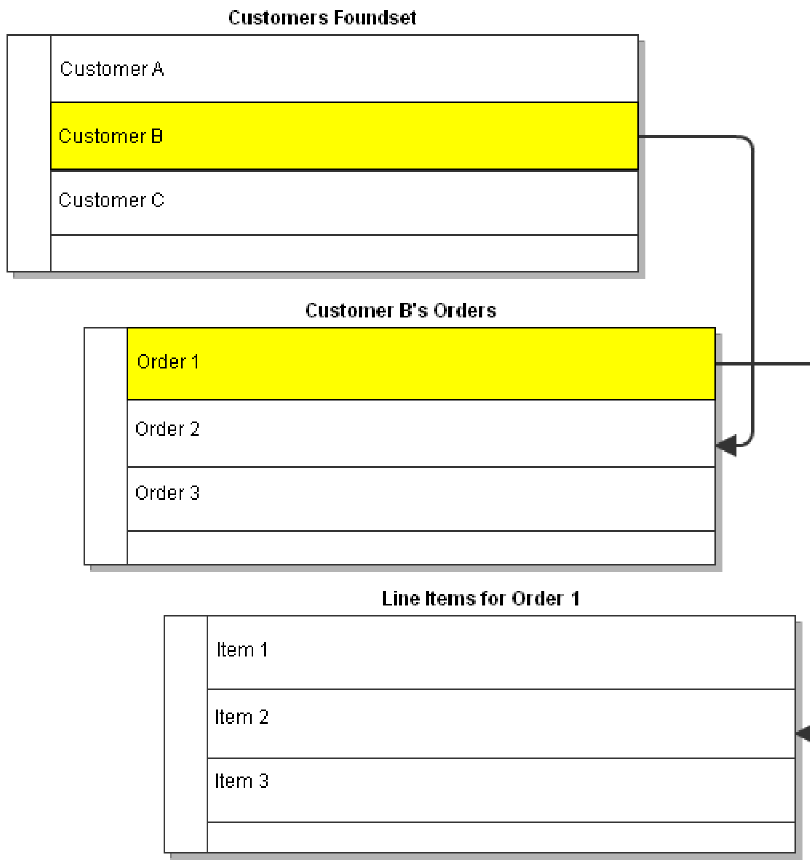
Take a predefined Relation, customers_to_orders, which models a one-to-many relationship between a customers table and an orders table. The following three lines of code, executed within the scripting scope of a form based on the customers table, all produce the same result.

```
// 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.

For 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 JSFoundsetUpdater 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.

```
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.

Updating each record individually

The Foundset Updater API can also be used to update records individually, but still holds a performance advantage over iterating on a foundset, which has more overhead and can cause the foundset's cache size to increase unnecessarily. In the example below, each record in the foundset is updated with a unique value (A simple counter is incremented, which is arbitrary, but demonstrates that each record can be updated with a unique value.)

```
var count = 0;
var fsUpdater = databaseManager.getFoundSetUpdater(foundset)
while(fsUpdater.next())
{
        fsUpdater.setColumn('degrees',count++);
}
```

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

Find Mode

Find Mode is a special mode that can be assumed by a foundset object to perform data searches using a powerful, high-level abstraction. When in Find Mode, the foundset's *Data Providers*, normally used to read/write data, are instead used to enter search criteria. Any data provider can be assigned a search condition which should evaluate to a String, Number or Date. Because forms typically bind to a foundset, criteria may be entered from the GUI by the user or programmatically.

A foundset enters Find Mode when its *find* method is invoked. This method returns a Boolean, because under certain circumstances, the foundset may fail to enter find mode. Therefore, it is good practice to enclose a find in an an *if* statement, so as not to accidentally modify the selected record. A foundset exits Find Mode when its *search* method is executed, upon which the foundset's SQL query is modified to reflect the expressed criteria and the matching records are loaded. The *search* method returns an integer, which is the number of records loaded by the find. However this doesn't necessarily represent the total number of matching records as foundset records are loaded in blocks.

Example:

```
// Find all customers in the city of Berlin
if(foundset.find()){   // Enter find mode
    city = 'Berlin';   // Assign a search criteria
    foundset.search(); // Execute the query and load the records
}
```

Results in the foundset's SQL query:

```
SELECT customerid FROM customers WHERE city = ? ORDER BY customerid ASC //Query params: ['Berlin']
```

Search Criteria with Logical AND

When multiple search criteria are entered for multiple data providers, the criteria will be concatenated with a SQL *AND* operator.

Example:

```
// Find all customers in the city of Berlin AND in the postal code 12209
if(foundset.find()){     // Enter find mode    city = 'Berlin';     // Assign city search criterion
    city = 'Berlin';     // Assign a search criteria
    postalcode = '12209' // Assign postal code criterion
    foundset.search();   // Execute the query and load the records
}
```

Results in the foundset's SQL query:

```
 SELECT customerid FROM customers WHERE city = ?  AND postalcode = ? ORDER BY customerid ASC //Query params:
['Berlin','12209']
```

Multiple Find Records for Logical OR

It's important to note that when in Find Mode, a foundset will initially contain one record object. However, multiple record objects may be used to articulate search criteria. This has the effect that the criteria described in each record are concatenated by a SQL *OR*.

Example:

```
// Find customers in the city of Berlin AND in the postal code 12209...
// OR customers in the city of San Francisco AND in the postal code 94117
if(foundset.find()){        // Enter find mode    city = 'Berlin';
    city = 'Berlin';   // Assign a search criteria
    postalcode = '12209';
    foundset.newRecord();   // Create a new search record
    city = 'San Francisco'
    postalcode = '94117';
    foundset.search();      // Execute the query and load the records
}
```

Results in the foundset's SQL query:

```
 SELECT customerid FROM customers WHERE (city = ?  AND postalcode = ?) OR (city = ?  AND postalcode = ?) ORDER
BY customerid ASC //Query params: ['Berlin','12209','San Fransisco','94117']
```

Finding records through a relation

Find Mode is very flexible as searches can traverse the entire data model. When a foundset enters find mode, any foundset related to a search record can be used to enter criteria. Moreover, related foundsets can use multiple search records so any permutation of Logical AND / OR is possible.

Example:

```
// Find customers that have 1 or more orders which were shipped to Argentina
if(foundset.find()){                                // Enter find mode
    customers_to_orders.shipcountry = 'Argentina'; // enter criteria in a related foundset
    foundset.search();                              // Execute the query and load the records
}
```

Results in the foundset's SQL query:

```
SELECT DISTINCT customers.customerid FROM customers
LEFT OUTER JOIN orders ON customers.customerid=orders.customerid
WHERE orders.shipcountry = ? ORDER BY customers.customerid ASC
```

And there are no limitations to the number of traversals across related foundsets.

Example:

```
// Find customers with one or more orders containing one or more products supplied by a vendor in USA
if(foundset.find()){
    customers_to_orders.orders_to_order_details.order_details_to_products.products_to_suppliers.country = 'USA';
    foundset.search();
}
```

Finding records within a related foundset

It is worth pointing out that related foundsets may be put into Find Mode as well. The foundset will maintain the constraints imposed by the relation in addition to the criteria specified in the data providers.

Example: This operation is nearly identical to the previous search on ship country, however it matters which foundset is in Find Mode. The difference is that this operation searches for order records of a particular customer.

```
// Find orders of THE SELECTED CUSTOMER that were shipped to Argentina
if(customers_to_orders.find()){
    customers_to_orders.shipcountry = 'Argentina';
    customers_to_orders.search();
}
```

Results in the foundset's SQL query (notice the relation constraint is preserved):

```
SELECT orderid FROM orders  WHERE customerid = ? AND shipcountry = ? ORDER BY orderid ASC
```

Special Operators

Servoy's Find Mode provides several special operators that when used in combination can articulate the most sophisticated search requirements. Operators and operands should be concatenated as strings.

Operator	Description	Applicable Data Types	Example

||	OR: Used to implement a logical OR for two or more search conditions in the same data provider	Any	`// Cities of London or Berlin`
			`city = 'Berlin||London';`

|	Format: Used to separate a value and an implied format.	Date	`// exactly 01/01/2001 (00:00:00 implied)`
			`orderdate = '01/01/2001|MM/dd/yyyy';`

!	Not: Used to implement a logical NOT for a search condition.	Any	`// Anything but Berlin`
			`city = '!Berlin';`

#	Sensitivity Modifier: Implies a case-insensitive search for text columns. Implies a match on entire day for date columns.	Text, Date	`// i.e. Los Angeles, lOS aNGeLES`
			`city = '#los angeles';`
			
			`// any time on 01/01/2001`
			`orderdate = '#01/01/2001|MM/dd/yyyy';`

^	Is Null: Matches records where a column is not null.	Any	`// All null contact names, not including empty strings`
			`contactname = '^';`

^=	Is Null/Empty/Zero: Matches records where a column is null, empty string value or zero numeric value	Text, Numeric	`// All freights which are null or 0`
			`freight = '^=';`

<	Less than: Matches records where the column is less than the operand	Any	`// i.e. 50, 99.99, but not 100, 101`
			`freight = '<100';`

<=	Less than or equal to: Matches records where the column is less than or equals the operand	Any	`// i.e. Atlanta, Baghdad, Berlin, but not Buenos Aires, Cairo`
			`city = '<=Berlin';`

>=	Greater than or equal to: Matches records where the column is greater than or equals the operand	Any	`// Any time on/after 12am new year's day 2001`
			`orderdate = '>=01/01/2001|MM/dd/yyyy';`

>	Greater than: Matches records where the column is greater than the operand	Any	`// i.e. 100.01, 200, but not 99,100`
			`freight = '>100';`

...	Between: Matches records where the column is between (inclusive) the left and right operands.	Any	`// Any time during the year 2001`
			`orderdate = '01/01/2001...01/01/2002|MM/dd/yyyy';`
			
			`// i.e.`
			`freight = '100...200';`
			
			`// i.e. London, Lyon, Madrid, Omaha, Portland`
			`city = 'London...Portland';`

%	Wild Card String: Matches records based on matching characters and wild cards	Text	`city = 'New%';    // Starts with: i.e. New York, New Orleans`
			`city = '%Villa%; // Contains: i.e. Villa Nova, La Villa Linda`
			`city = '%s';     // Ends with: i.e. Athens, Los Angeles`

_	Wild Card Character: Matches records based on	Text	`// i.e. Toledo, Torino`
			`city = '%To___o%';`

\	Escape Character: Used to escape other string operators	Text	`// Escape the wild card, i.e. ...50% of Capacity...`
			`notes = '%\%%';`

now	Now: Matches records where the condition is right now, including time	Date	`// exact match on this second`
			`creationdate = 'now';`

today	Today: Matches records where the condition is any time today	Date	`// match on anytime today`
			`orderdate = 'today';`

Find Mode and the User Interface

The above examples deal with find mode in which find mode is entered, criteria are expressed and the search is run, all in a single action. The effect of the search is entirely up to the developer. However, find mode can also be entered in one action and searched in another action. In between, the user may manually enter values into fields to express the search criteria. They can then run the search action and a form's foundset will show the results of the search. Any of the above search criteria may be used.

Example In this example there is a method which can both enter find mode as well as run a search when in find mode. In between the two different invocations of this method, the user interface is ready to receive input from the user. When complete, the user may run the method again, this time the foundset will search for results.

```
/**
 * @AllowToRunInFind
 *
 * @properties={typeid:24,uuid:"088B830C-2A4F-483C-A135-5FA32A010AE9"}
 */
function doFind(){
        if(foundset.isInFind()){    // if the foundset is already in find mode, run the search
                foundset.search();
        } else {
                foundset.find();    // otherwise, enter find mode
        }
}
```

Find mode blocks the execution of any methods which are normally invoked from the user interface. This is a good thing as these methods may have unintended consequences when a form's foundset is in find mode. Notice the JSDocs tag **@AllowToRunInFind** in the comment block which precedes the method. This tag provides the metadata to let Servoy know that this method should be allowed to run while the form's foundset is in find mode. Without this exception, this method would be blocked from execution, and there would be no recourse to programmatically exit find mode.

Read-Only Fields

By default, even read-only fields will become editable for the duration of the find mode. This is often useful, because while a data provider may not be available to edit, in find mode, it becomes a vehicle to enter a search criterion and should be editable to the user. However, in some cases it may be desired that read-only fields remain so for the duration of find mode as well. Servoy provides a UI property which may be set through the Application API using the method setUIProperty.

Example This example is identical to the above example with the exception, that for the duration of this find, the read-only property of fields is maintained. After a find, it is set back to the default so as not to interfere with other functionality throughout the rest of the application.

```
/**
 * @AllowToRunInFind
 *
 * @properties={typeid:24,uuid:"088B830C-2A4F-483C-A135-5FA32A010AE9"}
 */
function doFind(){
        if(foundset.isInFind()){
                foundset.search();
                application.setUIProperty(APP_UI_PROPERTY.LEAVE_FIELDS_READONLY_IN_FIND_MODE, false)    //
reset to the default
        } else {
                application.setUIProperty(APP_UI_PROPERTY.LEAVE_FIELDS_READONLY_IN_FIND_MODE, true);    //
before entering find mode, enforce read-only fields
                foundset.find();
        }
}
```

Canceling Find Mode

Find mode can be programmatically cancelled by invoking the loadAllRecords method of the foundset. The foundset will revert to the query prior to entering find mode. Within the Smart Client the user can cancel Find mode by pressing Escape. This will trigger the loadAllRecords command of the Form to which the foundset is bound.

Complex Searches

Servoy's find mode can be used to easily satisfy even complex search requirements. Remember that any related foundset may be used to enter criteria and that any number of search records may be used in any foundset and any operators may be used in combination for every data provider.

Table Definitions

Servoy's data layer offers an assortment of design-time settings, defined at the table and column level,
to specify metadata and data-bound business rules.

Column Properties

Servoy will read column definitions from relational database tables. The properties that it reads include:

Column Name

This is the name that is returned by the JDBC Driver. The name will always be displayed as lower case. The column name is also used as the Data Provider ID when working with foundsets and records.

Data Type

While relation databases support many different data types, Servoy will generalize the data type of a column into one of five general types. In this way, Servoy can support a wide range of database vendors. The five generalized data types include the following:

1. Text: Any alpha-numeric characters (i.e. char, varchar, memo, CLOB, etc.)
2. Integer: Whole numbers (i.e. bit, short, long, bigint, etc.)
3. Number: Decimal numbers (i.e. float, double, etc)
4. Datetime: Temporal values (i.e. date, datetime, timestamp, etc.)
5. Media: Binary data (i.e. BLOB)

Column Length

For certain data types, databases must enforce the amount of storage allocated to single column for a single record. Data types which accommodate variable length entries, such as text, decimal numbers and binary data will have a length property. Servoy will infer and display this property in the column definition.

Row Identifiers

Servoy is designed to work with regular database tables as well as *SQL Views*. Regular database tables will have a *primary key*, consisting of one or more columns, who's value uniquely identifies a record in the table. Servoy will infer the primary key from the database table. However, in the case of SQL Views, which don't have a built-in primary key, the developer must specify which column(s) can be considered the unique row identifier.

Null Values Allowed

Relational database tables may enforce non-null constraints on certain columns, typically for primary key and other essential columns. Servoy will infer from any such constraints from database table and reflect

Column Meta Data

Column definitions include several metadata properties, which store information that is used both in development and at runtime.

Title

The *Title* property of a column is simply the human-readable name for a column. When a field is placed on a form with the *Place with labels* option, the label's *text* property will be initialized to the *title* property for the column to reach the field is bound. For multilingual applications, it is ideal to populate a column's *title* property with an *i18n message key*, thus allowing field labels to default to message key, which is translated at runtime.

Default Format

The *Default Format* property of a column will enforce the formatting that is used when the column is bound to a field element. The field element's format property will assume the *default format* of the column unless it is overridden in the element.

Foreign Type

This is a simple metadata property to indicate that a column is a foreign key to another table in the same database. One can set the Foreign Type property to the target table. This provides metadata so developers will know that a column is used as a foreign key. Servoy will use this information when new relations are created between the tables and auto-fill the keys. This property is also used by the mergeRecords method of the databaseManager API to update any affected related records, such that they'll reference a new key.

Exclude Flag

Enabling a column's *Excluded Flag* will completely exclude a column from the Servoy runtime environment. This means that Servoy will exclude this column for every query that it issues. This option is ideal to enforce that certain columns are never available in a Servoy application.

UUID Flag

Servoy supports the use of *Universally Unique Identifiers* (UUID). A UUID is a 16-byte number which can be (practically) guaranteed to be unique across computing environments, making it ideal to use for sequences in scenarios where traditional numeric sequences are not adequate, for example when syncing data which is generated offline. It is generally not feasible to store UUIDs as numeric data types because the number is so large. Rather UUIDs are most easily stored as 36-character strings. When using a text column to store UUIDs, one should mark the column's UUID flag. Thus, Servoy will provide programmatic access to this column in the form of a built-in UUID data type, which allows both string and byte representation.

Description

A column's *description* property is a simple container for additional metadata, such as programmer notes about the column's purpose, etc.

Auto-Enter Definitions

Servoy provides several ways in which a column may be automatically populated when a record is created. Some of the auto-enter options are also applicable when an existing record is updated.

System Values

Database-Managed

Indicates that the value is deferred to the database at the time of insert. The value is populated and controlled by the database and it will not be overwritten from Servoy.

Creation User UID

The UID parameter that was supplied at the time of login, entered at the time of record creation.

Modification User UID

The UID parameter that was supplied at the time of login, reentered each time the record is modified.

Creation Datetime

The current date and time on the client, entered at the time of record creation.

Creation Server Datetime

The current date and time on the application server, entered at the time of record creation.

Modification Datetime

The current date and time on the client, reentered each time the record is modified.

Modification Server Datetime

The current date and time on the application server, reentered each time the record is modified.

Custom Value

A custom value is simply a literal value (i.e. 'Blue', 1.5) which may be used as a default. This option is only available for Integer, Number and Text data types.

Database Default

This indicates that the value is deferred to the database at the time of insert. However, unlike Database-Managed system values, this value can be modified from Servoy after the record is inserted.

Lookup Value

Lookup Values provide the option to auto-enter a value that is contextual to the record being inserted. Options include any of the record's data providers, any data providers from foundsets related to the record, as well as an global relations or variables.

Sequence

Sequences may be used to auto-increment a column's value. This is ideal for populating primary key columns, which must be unique.

Servoy Sequence

This is a sequence which is defined in the application tier and managed by Servoy. The sequence will generate integer values using a given *next* value and *step* value.

i.e a step value of 1 will yield sequential values of 1,2,3,4...

A step value of 2 will yield 1,3,5,7...

In deployment, Servoy Sequences are stored in the repository database and there are options to recalculate the sequence's *next value* from existing data.

Database Sequence

Servoy will call a named sequence in the database to populate the value. The column will be populated and available prior to inserting the record

Database Identity

The sequential values are managed and populated by the database. The column is not populated until after the record is inserted

UUID Generation

Servoy will automatically populate a text column with a textual representation of a UUID. Be sure that the column's UUID Flag is also enabled.

Column Validation

Servoy provides an opportunity to implement validation rules at the column level. There are several built-in validation rules, which may be implemented at design-time. Additionally, custom validation rules may be written in as a JavaScript method which is bound to a column. A validation event occurs at the moment a record's value for a column changes. This may be the result of a user's action or some code which is executed. When validation fails, a Servoy Exception is raised for Invalid Input, which may be trapped in a solution's onError event hanlder.

Numeric Range Validation

Servoy provides built-in numeric validation for Integer and Number data types. Providing upper and lower bounds will automatically enforce that any value entered is between (inclusive) the range provided. Providing only a lower bound will enforce that any value entered is greater-than-or-equal-to the bound. Providing only an upper bound will enforce that any value entered is less-than-or-equal-to the bound.

Size/Length Validation

Servoy provides built-in validation for the size/length of a value in a column. This rule is applicable to Text and Media data types. Setting the length property for Text columns will enforce that value entered has a length of characters which is less-than-or-equal-to the length specified in the rule. Setting the size property for Media columns will enforce that value entered has a size, measured in number of bytes, which is less-than-or-equal-to the size specified in the rule.

RegEx Validation

Servoy offers the flexible pattern matching capability of Regular Expressions as a means to apply validation rules to Text columns. Providing a RegEx value will enforce that any value entered into the Text column must match on the expression. RegEx is an excellent way to match on patterns, such as phone numbers, email addresses, and much more. RegExLib is a useful site containing user-generated libraries of expressions to suit many needs.

Email Validation

Servoy provides a built-in email validation rule, which enforces that any Text column matches a pattern which is similar to email addresses. This pattern is ideal for most use cases. However, developers may implement their own RegEx validation to ensure an exact match on the pattern of their choice.

Custom Validation

Apart from the built-in validation rules, Servoy allows developers to author business logic to enforce their own validation rule for a column. A Global Method may be bound to a column, such that when a validation event occurs for the column, the method is invoked. The value that is entered is passed into the method and a developer may then execute any evaluation of the value before returning a boolean value; true indicates that validation is successful.

```
/**
 * Custom Validation rule: Must be Dog or Cat (case insensitive)
 * @param {Object} obj The value that will be validated
 * @returns {Boolean} True when successful
 * @properties={typeid:24,uuid:"655B9F0E-A1A2-4B0B-84CD-8E299546DB57"}
 */
function validateColumn(obj) {
        return 'Dog'.equalsIgnoreCase(obj) || 'Cat'.equalsIgnoreCase(obj);
}
```

Column Conversion

Some scenarios require that a value be stored in a database column in one form and written to and read from the database column in another form. Servoy supports this requirement with a feature called *Column Conversion* and it has two applications, *String Serialization* and *Global Method Conversion.*

String Serialization

Servoy supports object persistence using *String Serialization*, which involves the conversion of a runtime object into a string format, which can then persist in a database column. When the column is read from the database, the persistent string will be deserialized back into a runtime object. Because Servoy uses JavaScript as its scripting language, runtime objects will be serialized into standard JSON format.

```
//  Construct an object to capture some custom settings and write it directly to a Text column called
'custom_settings'
var obj = new Object();
obj.name = 'foobar';
obj.message = 'Hello World';

// at this point it is serialized into the string: "{name:'foobar',message:'Hello World'}"
custom_settings = obj;
databaseManager.saveData();

// ...read object properties at a later time...
application.output(custom_settings.message + 'My name is: ' + custom_settings.name);
```

Remember that only by assigning an object to a data provider will you actually store the serialized string. You cannot set individual instance properties of an object to directly modify the serialized string.

```
// For Example
my_data_provider.property = 'Foobar'; This will have no effect on the data provider

// Instead
var obj = my_data_provider; // read the data provider into a runtime object
obj.property = 'Foo Bar';   // Modify the Object's instance properties
my_data_provider = obj;     // And reassign it to the data providerdatabaseManager.saveData();
```

Global Method Conversion

Servoy allows a database column to be bound to custom business logic, giving developers control over how a value is converted when it is written to, and read from the data provider.

The nomenclature refers to the **Object Value**, seen in the GUI, as well as used programmatically, and the **Database Value**, the value stored in the data provider and persisting in the database.

The column is bound to two methods which facilitate the conversion between the Object Value and the Database Value. A developer may also specify an optional **Object Data Type**, prompting Servoy to provide the data in an alternate data type in lieu of the default column type. This is useful when values are stored in a non-standard storage type to accommodate legacy systems, but should be treated like standard data type in the runtime.

Object to Database Method

This method is called anytime a value is written to the data provider. It will be called regardless of the origin of the action, i.e. GUI event or programmatically. It will be called before data is committed to the database.

Parameters

Object - The value that is being written to the data provider

String - The column's data type: TEXT, INTEGER, NUMBER, DATETIME, MEDIA

Returns

Object - The converted value that will actually be written to the data provider.

Example

Perhaps the most classic use case is the conversion between SI Units, where a database is standardized on a certain unit, but an application requires that values be written and read in multiple units, often to support different locales / preferences. Imagine a database column for temperature, which is standardized on Celsius, but an application which allows data entry in Celsius, Fahrenheit and Kelvin.

```
/**
 * This method auto-converts from client units to Celsius as the value is being written to the data provider
 * @parameter {Object} value The value of the runtime object
 * @parameter {String} columnType The data type of the column
 * @returns {Object} The value converted into celsius
 * @properties={typeid:24,uuid:"303ACB93-3B0E-4B9C-9550-D78FF17343C2"}
 */
function objectToDB(value, columnType) {

        // evaluate client unit settings
        switch(tempUnits){

                // Already in C, just return it as is
                case C :
                        return value;

                // Fahrenheit,use conversion formula
                case F :
                        return (5/9)*(value-32);

                // Kelvin,use conversion formula
                case K :
                        return value - 273;
        }
}
```

Database to Object Method

This method is called anytime a value is read from the data provider. It will be called when it is displayed in the GUI or read programmatically.

Parameters

Object - The value that is being read from the data provider

String - The column's data type: TEXT, INTEGER, NUMBER, DATETIME, MEDIA

Returns

Object - The converted value that will actually be displayed in the GUI and read programmatically.

Example

Perhaps the most classic use case is the conversion between SI Units, where a database is standardized on a certain unit, but an application requires that values be written and read in multiple units, often to support different locales / preferences. Imagine a database column for temperature, which is standardized on Celsius, but an application which allows data entry in Celsius, Fahrenheit and Kelvin.

```
/**
 * This method converts database values (Celsius) into the current client units for degrees
 * @parameter {Object} value The value stored in the column
 * @parameter {String} columnType The data type of the column
 * @returns {Object} The value that was converted into current client units
 * @properties={typeid:24,uuid:"63C4D552-531C-48DB-A6C6-ED02F4603C20"}
 */
function dbToObject(value, columnType) {

        //        evaluate client unit settings
        switch(tempUnits){

                // Already using C, just return it as is
                case C :
                        return value;

                // Fahrenheit, use conversion formula
                case F :
                        return (9/5) * value + 32;

                // Kelvin, use conversion formula
                case K :
                        return value + 273;
        }
}
}
```

Converted Object Type

One can optionally specify the data type of the Object Value. This is useful in situations where the stored value is a different data type than the object value

Example

The application talks to a database that is storing dates as 8-character text columns to support legacy applications. By setting the **Converted Object Type** setting to DATETIME, Servoy will treat the column as a date object. Moreover, the two conversion methods written by the developer should assume the Object Value is a Date object.

```
/**
 * This method converts Text data stored in the database column, presenting it as Date object
 * @parameter {Object} value The value stored in the column
 * @parameter {String} columnType The data type of the column
 * @returns {Object} The value that was converted
 * @properties={typeid:24,uuid:"16BDC049-E63B-47C4-B49C-595D916FD51B"}
 */
function dbToObj(value, columnType) {
        return utils.dateFormat(value,'MMddyyyy');
}
```

Calculations

A Calculation is very much like a database column, except that its value, rather than being stored, is dynamically computed each time it is requested. This is achieved by creating a JavaScript function which is bound to a database table. The function takes no arguments and returns a value, which like a real database column, is declared to be one of the five general data types. The scope of the JavaScript function is an individual record object. Therefore any of the record's other data providers, and related foundsets are immediately available, as well as global variables and globally related foundsets.

A calculation is declared at the solution level, and is available throughout the solution in which it is declared, as well as any modules containing it. To support this, there is one JavaScript file per table, per solution, which holds calculation functions. For example, the database tables 'orders' may have a corresponding file 'orders_calculations.js' in any solution. And this file could contain many individual calculation functions.

Just like real database columns, calculations may be placed as fields on forms, used in data labels, and requested programmatically.

Performance

Calculations may be called often. Therefore, developers should use discretion when implementing their JavaScript function. Most in-memory operations are very fast. However, actions which result in SQL queries or file operations may be slower and less predictable. For example, a calculation may be shown in a scrolling table-view form, in which case it may be called hundreds of times.

Example

A simple calculation *subtotal* on a database table *order_details* which yields a Number value; the unit price, multiplied by the quantity, with a discount applied.

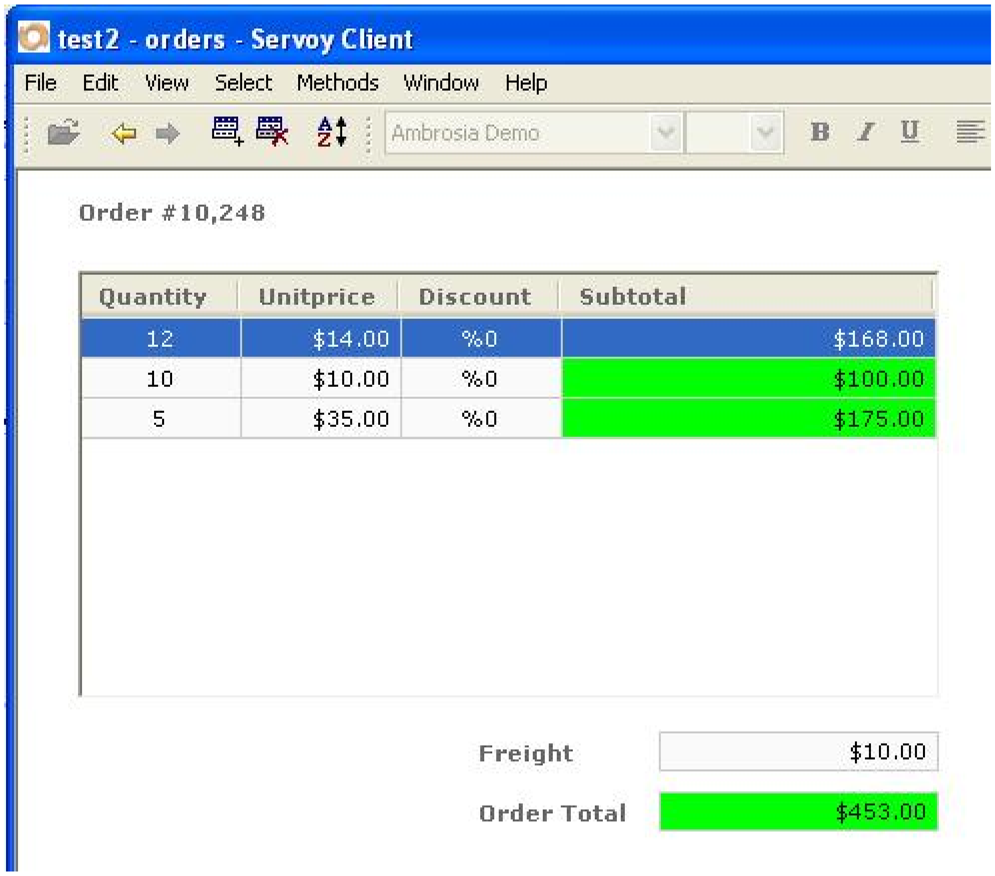
```
/**/**
 * @properties={type:6,typeid:36,uuid:"644DCF7D-11C7-475E-82CD-F4F60ED00D77"}
 */
function subtotal()
{
	return unitprice * quantity * (1 - discount);
}
```

This function is executed for the scope of an individual record. Notice that the record's other data providers, *unitprice* and *quantity* are immediately available.

This calculation could now be placed on a form or used in a method, just like any other real database column. The next code example shows another calculation, this time declared for the orders table, which uses the previous calculation to determine the total amount for the entire order.

```
/**/**
 * @properties={type:6,typeid:36,uuid:"d5458801-bbd2-43a1-994d-4ac2f6d12595"}
 */
function order_total()
{
	var sum = 0;
	for(var i = 1; i <= orders_to_order_details.getSize(); i++){
		sum += orders_to_order_details.getRecord(i).subtotal;
	}
	return sum + frieght;
}
```

Notice that the related foundset, *orders_to_order_details*, a property of an order record, is immediately available. Also notice that the previous calculation, *s ubtotal*, may be referenced, just like any other column. Note that while, calculations are a JavaScript function, unlike regular methods, one does not use the parentheses to invoke their value. Both calculations may be placed on forms like any other data provider. (See image below - calculations are highlighted in green)


test2 - orders - Servoy Client
File Edit View Select Methods Window Help
Ambrosia Demo

Order #10,248

Quantity	Unitprice	Discount	Subtotal
12	$14.00	%0	$168.00
10	$10.00	%0	$100.00
5	$35.00	%0	$175.00

Freight	$10.00
Order Total	$453.00

Stored Calculations

Calculations may be optionally stored back to a real database column. This is called a *Stored Calculation,* and is achieved by creating a calculation which is the same name and data type as a real column. When the calculation is referenced, it will be executed and its result will be stored back in the database column.

Bear in mind that a stored calculation is not guaranteed to be recently calculated when used in a find, a sort definition, or as the right-hand key in a relation, because its value is computed in memory and the database value only represents the most recent execution of the calculation

Calculation as a Record-Level Variable

In general, a calculation is a ready-only data provider, its value being generated each time it is read. However, in one exception a calculation may be used to cache data for a record, thus becoming a read/write in-memory data provider. This is done by creating a calculation which has no *return* statement. Such a calculation will be treated a little differently. It can actually store a value in memory for an individual record.

Example In this example, a record-level variable is created by defining an empty calculation.

```
/**/**
*  defined in my table.js
 * @properties={type:12,typeid:36,uuid:"32BE69DF-289E-45A6-A347-50271F3F29D7"}
 */
function record_flag()
{
        // NO return statement / value
}
```

Next a method illustrates how the calculation may be used to store, in memory, information about the record.

```
/**
 * @param {JSEvent} event
 *
 * @properties={typeid:24,uuid:"75E3667B-AED9-4E79-ADE8-CCBED66A8D2F"}
 */
function flagSelectedRecord(event){

        // toggles the flag between 0 & 1
        if(record_flag){
                record_flag = 0
        } else {
                record_flag = 1;
        }
}
```

Aggregations

An aggregation is a data provider which represents a database column that is aggregated over a set of records. At design-time aggregations have the following properties:

- **Name** - The name by which the aggregation will be available as a data provider throughout the solution in which it is declared, as well as any modules containing it.
- **Type** - There are five types of aggregations:
  - **Count** - The number of records in an entire foundset containing a non-null value for a column
  - **Sum** - The sum of all the values for numeric column in an entire foundset
  - **Minimum** - The smallest value for a numeric column in an entire foundset
  - **Maximum** - The largest value for a numeric column in an entire foundset
  - **Average** - The average value for a numeric column in an entire foundset
- **Column** - The column containing values that are aggregated

At runtime, aggregations are computed in the context of a foundset. The value is derived from a SQL query, which takes the form of a SQL Aggregate Function and appends the WHERE clause used by the foundset's query.

Example

Assume an aggregation, *record_count*, declared on the *customer* table. The aggregation type is *count* and the column is *customerid*. The aggregation is available for any foundset based on the *customers* table.

```
function printRecordCcount(event) {function printRecordCcount(event) {
        application.output(record_count);                // print record count before find
        if(foundset.find()){                                // Find all customers where city starts with 'B'
                city = 'B%';
                foundset.search();
                application.output(record_count);        // print the record count after find
        }
}
```

After the find, the aggregation is re-queried using the foundset's new WHERE clause.

```
SELECT COUNT(customerid) AS record_count FROM customers WHERE city LIKE ? LIMIT ?
```

Performance

Because aggregations are derived from SQL queries, they may not reflect data changes, seen in the client, but not yet committed to the database. Aggregations will refresh after outstanding changes are committed.

SQL Aggregate Functions may be expensive operations, depending on the size and structure of a database table and the nature of the aggregation. Developers are encouraged to use discretion when working with aggregations. For example, when an aggregation is shown in a table-view form, it may result in a query for each record in displayed on the form, and performance may degrade.

Table Events

Servoy provides an event model at the data layer, giving developers the opportunity to implement validation and execute business logic just before and after data changes are committed to the database. There are six *Table Events,* each of which may be bound to global methods.

The first three events occur just prior to the change being committed to the database. Moreover, the event handler has the opportunity to veto the event, preventing the change from being committed. This is and ideal location to implement fail-safe data rules.

- **onRecordInsert** - occurs prior to a new record being inserted into the table
- **onRecordUpdate** - occurs prior to an existing record being updated in the database
- **onRecordDelete** - occurs prior to an existing record being deleted from the database

An *onRecordXXX* event is bound to a global method, which is invoked when the event occurs. The record that is modified is passed in as an argument and the method can veto the change by returning false or throwing an exception.

Parameters

JSRecord - the record object that is to be inserted, updated or deleted

Returns

Boolean - Return true from this method to allow the change to commit to the database. Returning false will result in a Servoy Exception with a SAVE FAILED error code, which may be handled in the solution's onError event handler.

Example

This is an example of an *onRecordDelete* handler for an *invoices* table. The data rule is that posted invoices will never be deleted by a the application.

```
/**/**
 * Record pre-delete trigger.
 * Validate the record to be deleted.
 * When false is returned the record will not be deleted in the database.
 * When an exception is thrown the record will also not be deleted in the database but it will be added to
databaseManager.getFailedRecords(),
 * the thrown exception can be retrieved via record.exception.getValue().
 *
 * @param {JSRecord} record record that will be deleted
 * @returns {Boolean} true to allow a delete
 * @properties={typeid:24,uuid:"A3F02F99-B899-46BE-9125-66E4189F043F"}
 */
function onRecordDeleteInvoice(record) {

        if(record.is_posted)
                throw "Cannot delete a posted invoice";

        return true;
}
```

The next three events occur immediately following the commit to the database. This is an ideal mechanism to update the data model after data is known to have changed.

- **afterRecordInsert** - occurs subsequent to a new record being inserted into the table
- **afterRecordUpdate** - occurs subsequent to an existing record being updated in the database
- **afterRecordDelete** - occurs subsequent to an existing record being deleted from the database

An *afterRecordXXX* event is bound to a global method, which is invoked when the event occurs.

Parameters

JSRecord - the record object that was recently inserted, updated or deleted

Example

This is an example of an *afterRecordInsert* handler for a *projects* table. The data rule is that a new project record will be linked, via the *projects_users* table, to the current user.

```
/**/**
 * Record after-insert trigger.
 *
 * @param {JSRecord} record record that is inserted
 *
 * @properties={typeid:24,uuid:"92834B20-1CAC-472F-B022-DD97FEFEA792"}
 */
function afterRecordInsert(record) {
        if(record.projects_to_projects_users.newRecord()){                                // create a link
record
                record.projects_to_projects_users.user_id = globals.currentUserID;        // associate to
current user
                databaseManager.saveData();
        }
}
```

Using Table Filters

Servoy provides high-level filtering functionality which may be applied to any database table using the addTableFilterParam method of the databaseManager API. Table filters have the following properties:

- **Scope** - A filter may be applied to a single database table or, if no table is specified, an entire server connection. A filter will constrain the records which are returned by any queries that are issued from the Servoy client to the table(s). A filter takes effect immediately upon being added and remains in effect for the duration of the client session unless programmatically removed. The constraints of a filter apply to all facets of a Servoy solution, including:
  - foundsets
  - related foundsets
  - value lists

> There are only two ways to circumvent a table filter:
> # by issuing a custom SQL query String through the getDataSetByQuery method of the databaseManager API. (Note: the version of getDataSetByQuery that takes a QueryBuilder object as first parameter DOES take into account applicable TableFilters
>
> # by using the rawSQL plugin
>
> Therefore, if one wishes to maintain the effects of a filter, it is important to remember to modify queries with an appropriate SQL WHERE clause in case any of the above two cases apply

- **Logical Expression** - A filter will contain a logical expression which is evaluated on behalf of records in the filtered table(s). Only records, for which the expression evaluates to true, will be returned by any queries issued to the filtered table(s). At runtime, the filter will be translated into an SQL WHERE clauseand appended to the query of any foundset which is bound to the filtered table(s). An expression contains the following components:
  - Data Provider Name - This is the left-hand operand. It is the name of a single column by which to filter. When filtering an entire server connection, only tables which contain the named column will be filtered.
  - Operator - The following operators are supported

=	Only records whose column **equals** the specified value
<	Only records whose column is **less than** the specified value
>	Only records whose column **greater than** the specified value
>=	Only records whose column **greater than or equals** the specified value
<=	Only records whose column **less than or equals**the specified value
!=	Only records whose column does **not equal** the specified value
^	Only records whose column value **is null**
LIKE	Only records whose column matches using the **SQL LIKE** construct (use wildcard '%' characters)
IN	Only records whose column value is in (using the SQL IN construct) a list of values
#	Modifier, used to make case-insensitive queries
||	Modifier used to concatenate two conditions w/ a logical OR

> Operators and modifiers may be combined, producing more complex conditions. For example #^||!= would translate to: is null OR case-insensitive not equals

  - Data Provider Value - This is the right-hand operand and should evaluate to a literal value to be compared with the named column.

When using the IN operator, one should provide an array of values or a String, which may be used as a sub select for the SQL IN clause.

- **Filter Name** - When adding a table filter parameter, a filter name may be used to allow for the later removal of a named filter. Multiple parameters or conditions may be set using the same filter name. In this case, all parameters may be removed at the same time.

*Example *This is a simple example which filters records in a products table based on the criterion that the status is not discontinued

```
var success = databaseManager.addTableFilterParam('crm_server','products','product_status','!=',globals.
STATUS_DISCONTINUED,'productfilter');
```

Example This example shows a two filters using the **IN** operator

```
// Filter products within an array of product codes
var success = databaseManager.addTableFilterParam('crm', 'products', 'productcode', 'in', [120, 144, 200]);


// Filter orders using a subselect
var success = databaseManager.addTableFilterParam('crm', 'orders', 'countrycode', 'in', 'select country code
from countries where region = "Europe"')
```

Example This example shows how to filter an entire server connection by passing <null> for the table name. This is ideal for multi-tenant architectures as an entire server connection can be filtered by a single expression, i.e. the current company

```
// all tables that have the companyid column should be filtered
var success = databaseManager.addTableFilterParam('crm', null, 'companyidid', '=', globals.currentCompanyID)
```