Foundset property type

Purpose of this property type

This page is written mostly for NG web component creators, not for Servoy developers that just want to use web components. You might want to view general Servoy Foundset documentation instead.

The 'foundset' property type can be used by web components to access/change a foundset's data/state directly from the browser.

The foundset typed property in the browser will work based on a 'viewport' of the server's foundset. The viewport is controlled directly by the component's code. Server will adjust foundset viewport bounds/contents only when needed due to data changes, deletes, inserts...

The foundset property also gives the possibility of knowing/changing the selection of the foundset.

For advanced uses, the foundset property can be linked to/interact with other property types (dataprovider, tagstring, component, ...), so that those other properties will provide a viewport as well - representing the same rows/records as in the foundset's viewport. The properties that support foundset view of data will allow the web component to specify a "forFoundset: "[foundsetPropertyName]" in their own property's description in the .spec file.

For foundset property types Servoy Developer allows (in properties view) one of the following:

- a (parent) form's foundset
- a related foundset
- a separate foundset (of any table; similar to JSDatabaseManager.getFoundset()). When this option is chosen the user can also choose whether or not the separate foundset should load all records initially. (if not checked, contents can be loaded at any time from scripting)
- "- none -" which means that you are going to set that foundset at runtime through scripting.

Foundset property value in browser scripting

In browser js, a foundset property value has the following content:

Browser side provided property content in model

```
myFoundset: {
    foundsetId: 2, // an identifier that allows you to use this foundset via the 'foundsetRef' type;
                   // when a 'foundsetRef' type sends a foundset from server to client (for example
                   // as a return value of callServerSideApi) it will translate to this identifier
                   // on client (so you can use it to find the actual foundset property in the model if
                   // server side script put it in the model as well); internally when sending a
                   // 'foundset' typed property to server through a 'foundsetRef' typed argument or prop,
                   // it will use this foundsetId as well to find it on server and give a real Foundset

    serverSize: 44, // the size of the foundset on server (so not necessarily the total record count
                    // in case of large DB tables)
    viewPort: {
        // this is the data you need to have loaded on client (just request what you need via provided
        // loadRecordsAsync or loadExtraRecordsAsync)
        startIndex: 15,
        size: 5,
        rows: [ { _svyRowId: 'someRowIdHASH1', name: "Bubu", type: 2 },
                { _svyRowId: 'someRowIdHASH2', name: "Ranger", type: 1 },
                { _svyRowId: 'someRowIdHASH3', name: "Yogy", type: 2 },
                { _svyRowId: 'someRowIdHASH4', name: "Birdy", type: 3 },
                { _svyRowId: 'someRowIdHASH5', name: "Wolfy", type: 4 } ]
    },
```

```
    selectedRowIndexes: [16], // array of selected records in foundset; indexes can be out of current
                              // viewPort as well
        sortColumns: 'orderid asc', // sort string of the foundset, the same as the one used in scripting for
                                                                // foundset.sort and foundset.getCurrentSort
    multiSelect: false, // the multiselect mode of the server's foundset; if this is false,
                        // selectedRowIndexes can only have one item in it
    hasMoreRows: false, // if the foundset is large and on server-side only part of it is loaded (so
                        // there are records in the foundset beyond 'serverSize') this is set to true;
                        // in this way you know you can load records even after 'serverSize' (requesting
                        // viewport to load records at index serverSize-1 or greater will load more
                        // records in the foundset)
    columnFormats: { name: (...), type: (...) }, // columnFormats is only present if you specify
                        // "provideColumnFormats": true inside the .spec file for this foundset property;
                        // it gives the default column formatting that Servoy would normally use for
                        // each column of the viewport - which you can then also use in the
                        // browser yourself

    /**
     * Request a change of viewport bounds from the server; the requested data will be loaded
     * asynchronously in 'viewPort'
     *
     * @param startIndex the index that you request the first record in "viewPort.rows" to have in
     *                   the real foundset (so the beginning of the viewPort).
     * @param size the number of records to load in viewPort.
     *
     * @return a $q promise that will get resolved when the requested records arrived browser-
     *                   side. As with any promise you can register success, error callbacks, finally, ...
     *                   See JSDoc of RequestInfoPromise.requestInfo and ChangeEvent.requestInfos
     *                   for more information about determining if a listener event was caused by this call.
     */
    loadRecordsAsync(startIndex: number, size: number): RequestInfoPromise<any>;

    /**
     * Request more records for your viewPort; if the argument is positive more records will be
     * loaded at the end of the 'viewPort', when negative more records will be loaded at the beginning
     * of the 'viewPort' - asynchronously.
     *
     * @param negativeOrPositiveCount the number of records to extend the viewPort.rows with before or
     *                                after the currently loaded records.
     * @param dontNotifyYet if you set this to true, then the load request will not be sent to server
     *                      right away. So you can queue multiple loadLess/loadExtra before sending them
     *                      to server. If false/undefined it will send this (and any previously queued
     *                      request) to server. See also notifyChanged().
     *
     * @return a $q promise that will get resolved when the requested records arrived browser-
     *                   side. As with any promise you can register success, error callbacks, finally, ...
     *                   That allows custom component to make sure that loadExtra/loadLess calls from
     *                   client do not stack on not yet updated viewports to result in wrong bounds.
     *                   See JSDoc of RequestInfoPromise.requestInfo and ChangeEvent.requestInfos
     *                   for more information about determining if a listener event was caused by this call.
     */
    loadExtraRecordsAsync(negativeOrPositiveCount: number, dontNotifyYet: boolean): RequestInfoPromise<any>;

    /**
     * Request a shrink of the viewport; if the argument is positive the beginning of the viewport will
     * shrink, when it is negative then the end of the viewport will shrink - asynchronously.
     *
     * @param negativeOrPositiveCount the number of records to shrink the viewPort.rows by before or
     *                                after the currently loaded records.
     * @param dontNotifyYet if you set this to true, then the load request will not be sent to server
     *                      right away. So you can queue multiple loadLess/loadExtra before sending them
     *                      to server. If false/undefined it will send this (and any previously queued
     *                      request) to server. See also notifyChanged().
     *
     * @return a $q promise that will get resolved when the requested records arrived browser
     *                   -side. As with any promise you can register success, error callbacks, finally, ...
     *                   That allows custom component to make sure that loadExtra/loadLess calls from
     *                   client do not stack on not yet updated viewports to result in wrong bounds.
     *                   See JSDoc of RequestInfoPromise.requestInfo and ChangeEvent.requestInfos
     *                   for more information about determining if a listener event was caused by this call.
     */
    loadLessRecordsAsync(negativeOrPositiveCount: number, dontNotifyYet: boolean): RequestInfoPromise<any>;
```

```
    /**
     * If you queue multiple loadExtraRecordsAsync and loadLessRecordsAsync by using dontNotifyYet = true
     * then you can - in the end - send all these requests to server (if any are queued) by calling
     * this method. If no requests are queued, calling this method will have no effect. It returns nothing.
     */
    notifyChanged: function(),

    /**
     * Sort the foundset by the dataproviders/columns identified by sortColumns.
     *
     * The name property of each sortColumn can be filled with the dataprovider name the foundset provides
     * or specifies. If the foundset is used with a component type (like in table-view) then the name is
     * the name of the component on who's first dataprovider property the sort should happen. If the
     * foundset is used with another foundset-linked property type (dataprovider/tagstring linked to
     * foundsets) then the name you should give in the sortColumn is that property's 'idForFoundset' value
     * (for example a record 'dataprovider' property linked to the foundset will be an array of values
     * representing the viewport, but it will also have a 'idForFoundset' prop. that can be used for
     * sorting in this call; this 'idForFoundset' was added in version 8.0.3).
     *
     * @param {JSONArray} sortColumns an array of JSONObjects { name : dataprovider_id,
     *                     direction : sortDirection }, where the sortDirection can be "asc" or "desc".
     * @return (added in Servoy 8.2.1) a $q promise that will get resolved when the new sort
     *                    will arrive browser-side. As with any promise you can register success, error
     *                    and finally callbacks.
     *                    See JSDoc of RequestInfoPromise.requestInfo and ChangeEvent.requestInfos
     *                    for more information about determining if a listener event was caused by this call.
     */
    sort(sortColumns: Array<{ name: string, direction: ("asc" | "desc") }>): RequestInfoPromise<any>;

    /**
     * Request a selection change of the selected row indexes. Returns a promise that is resolved
     * when the client receives the updated selection from the server. If successful, the array
     * selectedRowIndexes will also be updated. If the server does not allow the selection change,
     * the reject function will get called with the 'old' selection as parameter.
     *
     * If requestSelectionUpdate is called a second time, before the first call is resolved, the
     * first call will be rejected and the caller will receive the string 'canceled' as the value
     * for the parameter serverRows.
     * E.g.: foundset.requestSelectionUpdate([2,3,4]).then(function(serverRows){},function(serverRows){});
     *
     * @return a $q promise that will get resolved when the requested selection was updated server-
     *                    side. As with any promise you can register success, error callbacks, finally, ...
     *                    See JSDoc of RequestInfoPromise.requestInfo and ChangeEvent.requestInfos
     *                    for more information about determining if a listener event was caused by this call.
     */
    requestSelectionUpdate(selectedRowIdxs: number[]): RequestInfoPromise<any>;

    /**
     * Sets the preferred viewPort options hint on the server for this foundset, so that the next
     * (initial or new) load will automatically return that many rows, even without any of the loadXYZ
     * methods above being called.
     *
     * You can use this when the component size is not known initially and the number of records the
     * component wants to load depends on that. As soon as the component knows how many it wants
     * initially it can call this method.
     *
     * These can also be specified initially using the .spec options "initialPreferredViewPortSize" and
     * "sendSelectionViewportInitially". But these can be altered at runtime via this method as well
     * because they are used/useful in other scenarios as well, not just initially: for example when a
     * related foundset changes parent record, when a search/find is performed and so on.
     *
     * @param preferredSize the preferred number or rows that the viewport should get automatically
     *                      from the server.
     * @param {boolean} sendViewportWithSelection if this is true, the auto-sent viewport will contain
     *                                            the selected row (if any).
     * @param {boolean} centerViewportOnSelected if this is true, the selected row will be in the middle
     *                                           of auto-sent viewport if possible. If it is false, then
     *                                           the foundset property type will assume a 'paging'
     *                                           strategy and will send the page that contains the
     *                                           selected row (here the page size is assumed to be
     *                                           preferredSize).
     */
    setPreferredViewportSize: function(preferredSize, sendViewportWithSelection, centerViewportOnSelected),
```

```
    /**
     * It will send a data update for a cell (ros & column) in the foundset to the server.
     * Please make sure to adjust the viewport value as well not just call this method.
     *
     * This method is useful if you do not want to add angular watches on data (so calculated
     * pushToServer for the foundset property is set to just 'allow'). Then server will accept
     * data changes from this property, but there are no automatic watches to detect the changes
     * so the component must call this method instead - when it wants to change the data in a cell.
     *
     * @param rowID the _svyRowId (so $foundsetTypeConstants.ROW_ID_COL_KEY) column of the client side row
     * @param columnID the name of the column to be updated on server (in that row).
     * @param newValue the new data in that cell
     * @param oldValue the old data that used to be in that cell
     */
    updateViewportRecord(rowID: string, columnID: string, newValue: any, oldValue: any): void;

    /**
     * Add a change listener that is interested in knowing of any incoming changes (from server)
     * for this foundset property. See the "Adding a change listener" section below for more information.
     */
    addChangeListener : function(listener),

    /**
     * Removes the given change listener from this foundset property.
     */
    removeChangeListener: function(listener),

    /**
     * Receives a client side rowID (taken from myFoundsetProp.viewPort.rows[idx]
     * [$foundsetTypeConstants.ROW_ID_COL_KEY]) and gives a Record reference, an object
     * which can be resolved server side to the exact Record via the 'record' property type;
     * for example if you call a handler or a $scope.svyServoyapi.callServerSideApi(...) and want
     * to give it a Record as parameter and you have the rowID and foundset in your code,
     * you can use this method. E.g: $scope.svyServoyapi.callServerSideApi("doSomethingWithRecord",
     *                     [$scope.model.myFoundsetProp.getRecordRefByRowID(clickedRowId)]);
     *
     * NOTE: if in your component you know the whole row (so myFoundsetProp.viewPort.rows[idx])
     * already - not just the rowID - that you want to send you can just give that directly to the
     * handler/serverSideApi; you do not need to use this method in that case. E.g:
     * // if you have the index inside the viewport
     * $scope.svyServoyapi.callServerSideApi("doSomethingWithRecord",
     *           [$scope.model.myFoundsetProp.viewPort.rows[clickedRowIdx]]);
     * // or if you have the row directly
     * $scope.svyServoyapi.callServerSideApi("doSomethingWithRecord", [clickedRow]);
     *
     * This method has been added in Servoy 8.3.
     */
        getRecordRefByRowID: function(rowId)

}

// where the return value for some of the client side foundset methods is:

/**
 * Besides working like a normal IPromise that you can use to get notified when some action is done
 * (success/error/finally), chain etc., this promise also contains field "requestInfo" which can be set
 * by the user and could later be reported in some listener events back to the user (in case this same
 * action is going to trigger those listeners as well).
 *
 * @since 2021.09
 */
interface RequestInfoPromise<T> extends angular.IPromise<T> {

    /**
     * You can assign any value to it. The value that you assign - if any - will be given back in the
     * event object of any listener that will be triggered as a result of the promise's action. So in
     * case the same action, when done, will trigger both the "then" of the Promise and a separate
     * listener, that separate listener will contain this "requestInfo" value.
     *
     * This is useful for some components that want to know if some change (reported by the listener)
     * happened due to an action that the component requested or due to changes in the outside world.
     * (eg: FoundsetPropertyValue.loadRecordsAsync(...) returns RequestInfoPromise and
```

```
     * ChangeEvent.requestInfos array can return that RequestInfoPromise.requestInfo on the event that
     * was triggered by that loadRecordsAsync)
     */
    requestInfo?: any;

}
```

- **foundsetId** is controlled by the server; you should not change it

- **serverSize** is controlled by the server; you should not change it
- **viewPort** initial size can be changed using setPreferredViewportSize. When the component detects that more records that it needs are available, it care request viewPort contents using one of the two load **async** methods
    - **viewPort.startIndex** and **viewPort.size** will have the values requested by the async load methods. But if for example you are using data at the end of the foundset and records are deleted from there then viewport.size will be corrected/decreased from server (as there aren't enough records). A similar thing can happen to viewPort.startIndex. Do not modify these directly as that will have no effect. Use the load async methods instead.
    - **viewPort.rows** contains the viewPort data. Each item of the array represents data from a server-side record. Each item will always contain a "_svyRowId" ($foundsetTypeConstants.ROW_ID_COL_KEY in angular world) entry that uniquely identifies the record on server. Then there's one entry for every dataprovider that the component needs to use (how those are selected is described below). You should never change the "_svyRowId" entry, but it is possible to change the values of any of the other entries - the new values will be pushed back into the server side record that they belong to (if pushToServer is set on the foundset property to allow/shallow or deep; see "Data synchronization" section of https://wiki.servoy.com/display/public/DOCS/Specification).
- **selectedRowIndexes** is an array of selected foundset record indexes. This can get updated by the server if foundset selection changes server side. You can change the contents of this array to change foundset selection (new selection will be pushed to server). However, the preferred way of changing the record selection is by using "requestSelectionUpdate".
- **sortColumns** is a string containing the sort columns of the foundset, like 'columnA desc,columnB asc'
- **multiselect** represents the foundset multiselect state; do not change it as it will not be pushed to server.
- **columnFormats** represents the default column formats for the columns given in the viewport; do not change this - only server pushes this information to the client if asked to do so by the .spec file. It is only present if you specify "provideColumnFormats": true inside the .spec file for this foundset property.
- **hasMoreRows** true if the server side foundset has loaded only a part/chunk of it's records (in case of very large foundsets). In that case there are records even after 'serverSize'. It is controlled and updated by the server; you should not change it.

Adding a change listener

(available starting with Servoy 8.2)

When updates are received from the server for this foundset property, any listeners registered via **.addChangeListener()** - see above - will get notified.

This was added in order to improve performance by removing the need for angular watches. You no longer need to add lots of angular watches, deep or collection watches in order to be aware of incoming server changes to the foundset property. Each such watch would slow down the page - as watches are triggered a lot for all kinds of user actions or socket traffic. Also the listener can give more detailed information in order to do more granular updates to the UI easier.

Look at this change listener from the client side foundset property's point of view, not from the server's point of view. For example a NOTIFY_FULL_VALUE_CHANGED does not necessarily mean that the server side foundset has changed by reference. It actually means that all client side contents of the foundset property did change - or might have changed. So it is meant to notify about changes in client side property value.

To add an incoming server change listener to this property type just call:

Adding a change listener (for incomming changes from server)

```
var l = function(changes) {
    // check to see what actually changed and update what is needed in browser
};
$scope.model.myFoundset.addChangeListener(l);
```

If you are using foundset linked properties with your foundset property you might want to add the listener as shown here.

The "changes" parameter above is a javascript Object containing one or more keys, depending on what changes took place. The keys specify the type of change that happened; they can be any of the constants starting with NOTIFY_... from "$foundsetTypeConstants" service. The value gives any extra information needed for that type of change. Here is what "changes" can contain (one or more of the keys/values listed below):

what "changes" parameter can contain:

```
// ChangeEvent
{

    // If this change event is caused by one or more calls (by the component) on the IFoundset obj
    // (like loadRecordsAsync requestSelectionUpdate and so on), and the caller then assigned a value to
```

```
    // the returned RequestInfoPromise's "requestInfo" field, then that value will be present in this array.
    //
    // This is useful for some components that want to know if some change (reported in this ChangeEvent)
    // happened due to an action that the component requested or due to changes in the outside world. (eg:
    // IFoundset.loadRecordsAsync(...) returns RequestInfoPromise and ChangeEvent.requestInfos array can
    // contain that RequestInfoPromise.requestInfo on the event that was triggered by that loadRecordsAsync)
    //
    // @since 2021.09
    $foundsetTypeConstants.NOTIFY_REQUEST_INFOS: any[],

    // If a a full value update was received from server, this key is set; if newValue is non-null:
    //   - prior to Servoy version 2021.06: newValue is a new reference, but it will automatically get
    //     the old value's listeners registered to itself
    //   - starting with Servoy 2021.06: the old value's reference will be reused (so the reference of
    //     the foundset property doesn't change, just it's contents are updated) and oldValue given
    //     below is actually a shallow-copy of the old value's properties/keys; this can help
    //     in some component implementations
    $foundsetTypeConstants.NOTIFY_FULL_VALUE_CHANGED: { oldValue : ..., newValue : ... },

    // the following keys appear if each of these got updated from server; the names of those
    // constants suggest what it was that changed; oldValue and newValue are the values for what changed
    // (e.g. new server size and old server size) so not the whole foundset property new/old value
    $foundsetTypeConstants.NOTIFY_SERVER_SIZE_CHANGED: { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_HAS_MORE_ROWS_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_MULTI_SELECT_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_COLUMN_FORMATS_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_SORT_COLUMNS_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_SELECTED_ROW_INDEXES_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_VIEW_PORT_START_INDEX_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_VIEW_PORT_SIZE_CHANGED:  { oldValue : ..., newValue : ... },
    $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROWS_COMPLETELY_CHANGED:  { oldValue : ..., newValue : ... },

    // (ADDED in Servoy 2022.03)
    // This key is in the change event if the foundset spec property is configured "foundsetDefinitionListener":
true,
    // see "Defining initial load options for a foundset property" on this page. for more info
    $foundsetTypeConstants.NOTIFY_FOUNDSET_DEFINITION_CHANGE:  boolean,

    // if we received add/remove/change operations on a set of rows from the viewport, this key
    // will be set; as seen below, it contains "updates" which is an array that holds a sequence of
    // granular update operations to the viewport; the array will hold one or more granular add, remove
    // or update operations;
    //
    // BEFORE Servoy 8.4: all the "startIndex" and "endIndex" values below are relative to the viewport's
    // state after all previous updates in the array were already processed (so they are NOT relative to
    // the initial or final state of the viewport data!). Updates can come in a random order so there is
    // NO guarantee related to each change/insert/delete indexes pointing to the correct new data in the
    // final current viewport state
    //
    // STARTING WITH Servoy 8.4: all the "startIndex" and "endIndex" values below are relative to the
    // viewport's state after all previous updates in the array were already processed. But due to some
    // pre-processing that happens server-side (it merges and sorts these ops), the indexes of update
    // operations THAT POINT TO DATA (so ROWS_INSERTED and ROWS_CHANGED operations) are relative also to
    // the viewport's final/current state, so after ALL updates in the array were already processed
    // (so these indexes are correct both related to the intermediate state of the viewport data
    // and to the final state of viewport data).
    // This means that it is now easier to apply UI changes to the component as these granular updates
    // GUARANTEE that if you apply them in sequence (one by one) to the component's UI (delete, insert and
    // change included) you can safely use the indexes in there to get new data from the present state
    // of the viewport.
    //
    // indexes are 0 based
    $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES_RECEIVED: {

        // DEPRECATED in Servoy 8.4: granular updates are much easier to apply now; see comment above
        // Added in 8.3.2; sometimes knowing the old
        // viewport size helps calculate incomming granular updates easier
        $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES_OLD_VIEWPORTSIZE: ...,

        // starting with 8.3.2 you can use instead of 'updates' below the new constant;
        // $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES
        // before 8.3.2 just use 'updates';
        updates : [
```

```
            {
                    type : $foundsetTypeConstants.ROWS_CHANGED,
                    startIndex : ...,
                    endIndex : ...
            },
            {
                    // NOTE: insert signifies an insert into the client viewport, not necessarily
                    // an insert in the foundset itself; for example calling "loadExtraRecordsAsync"
                    // can result in an insert notification + bigger viewport size notification,
                    // with removedFromVPEnd = 0
                    type : $foundsetTypeConstants.ROWS_INSERTED,
                    startIndex : ...,
                    endIndex : ...,

                    // DEPRECATED starting with Servoy 8.4; it would always be 0 here
                    // as server-side code will add a separate delete operation instead - if necessary
                    // BEFORE 8.4: when an INSERT happened but viewport size remained the same, it was
                    // possible for some of the rows that were previously at the end of the viewport
                    // to slide out of it; "removedFromVPEnd" gives the number of such rows that were
                    // removed from the end of the viewport due to this insert operation;
                    removedFromVPEnd : ...
            },
            {
                    // NOTE: delete signifies a delete from the client viewport, not necessarily
                    // a delete in the foundset itself; for example calling "loadLessRecordsAsync" can
                    // result in a delete notification + smaller viewport size notification,
                    // with appendedToVPEnd = 0
                    type : $foundsetTypeConstants.ROWS_DELETED,
                    startIndex : ...,
                    endIndex : ...,

                    // DEPRECATED starting with Servoy 8.4; it would always be 0 here
                    // as server-side code will add a separate insert operation instead - if necessary
                    // BEFORE 8.4: when a DELETE happened inside the viewport but there were more rows
                    // available in the foundset after current viewport, it was possible for some of those
                    // rows to slide into the viewport; "appendedToVPEnd " gives the number of such rows
                    // that were appended to the end of the viewport due to this delete operation
                    appendedToVPEnd : ...
            }
        ]
    }

}
```

To make the "updates" part above clearer:

Let's say you had in your viewPort (before the incomming changes got applied to it):

```
row1
row2
row3
row4
row5
```

Then you got these "updates" from the listener (before Servoy 8.4):

```
updates: [
  // "newRow1" inserted
  { type: $foundsetTypeConstants.ROWS_INSERTED,
    start: 2, end: 2, removedFromVPEnd: 1 },

  // update row to "newRow2" contents
  { type: $foundsetTypeConstants.ROWS_CHANGED,
    start: 4, end: 4 }
]
```

that would be equivalent to the following (starting with Servoy 8.4):

```
updates: [
  // "newRow1" inserted
  { type: $foundsetTypeConstants.ROWS_INSERTED,
    start: 2, end: 2 },

  // update row to "newRow2" contents
  { type: $foundsetTypeConstants.ROWS_CHANGED,
    start: 4, end: 4 }

  // "row5" slides out of viewport due to
  // the initial insert
  { type: $foundsetTypeConstants.ROWS_DELETED,
    start: 5, end: 5 },
]
```

that means that the viewport has changed like this after the first update got applied (< 8.4):

```
row1
row2
newRow1
row3
row4
```

and for (>= 8.4)

```
row1
row2
newRow1
row3
row4
row5
```

and like this after the second (and third for >= 8.4) update got applied:

```
row1
row2
newRow1
row3
newRow2
```

Please note the when your listener is called the actual contents of the viewPort are already updated. So, at that time your viewport already looks like the last version above. You might find (for Servoy < 8.4) what $foundsetTypeUtils below provides useful depending on how you plan on using this listener. For 8.4 and higher you no longer need to compute indexes like that client-side (because the server pays attention to process all changes in such a way that the indexes in the ones that need to change data are the correct ones compared to the end state of the changed viewport).

⚠ **Always make sure to remove listeners when the component is destroyed**

It is important to remove the listeners when your component's scope is destroyed. For example if due to a tabpanel switch of tabs your form is hidden, the component and it's angular scope will be destroyed - at which point **you have to remove any listeners that you added on model properties** (like the foundset property), because the model properties will be reused in the future (for that form when it is shown again) and will keep any listeners in it. When that form will be shown again, it's UI will get recreated - which means your (new) component will probably add the listener again.

If you fail to remove listeners on $scope destroy this will lead to memory leaks (model properties will keep listeners of obsolete components each time that component's form is hidden, which in turn will prevent those scopes and other objects that they can reference from being garbage collected) and probably weird exceptions (obsolete listeners executing on destroyed scopes of destroyed components).

Example of removing a listener:

How to remove listeners on scope destroy

```
                        $scope.$on("$destroy", function() {
                                if (foundsetListener && $scope.model.myFoundset) $scope.model.myFoundset.
removeChangeListener(foundsetListener);
                        });
```

Defining/using a foundset property with a random set of dataproviders

A web component might want to work with as many dataproviders available in the viewport as the developer wants. Servoy Developer will allow selecting any number of dataproviders of the foundset to be sent to browser web component property value (through the properties view for the foundset typed property; use sub-property 'dataproviders').

For example a component that shows graphical representation of data might allow adding as many 'categories' to it as the developer wants to (each category getting data from one viewport column/dataprovider) .

.spec file

```
"myFoundset": { "type": "foundset", "dynamicDataproviders": true }
```

So the component has a property called "myfoundset" that it wants linked to any foundset chosen in ServoyDeveloper, and it allows the developer to choose in properties view any number of dataproviders from the foundset.

browser js

Let's say the developer has chosen a foundset and 3 dataproviders (for example 3 database columns) from it. Those would generate for example a viewPort like this inside the browser property.

Browser side provided property content in model

```
myFoundset: {
    (...)
    viewPort: {
        startIndex: 15,
        size: 2,
        rows: [ { _svyRowId: 'someRowIdHASH1', dp0: (...), dp1: (...), dp2: (...) },
                { _svyRowId: 'someRowIdHASH2', dp0: (...), dp1: (...), dp2: (...) } ],
        (...)
    },
    (...)
}
```

Notice the fixed column names: dp0, dp1, ... dp[N-1] where N is the number of foundset dataproviders that the developer has chosen.

> ⚠ **Only foundset dataproviders are supported**
>
> When using the dataproviders inside foundset property type (static or dynamic), only the record dataproviders are supported - so no form/global variables.

Defining/using a foundset property with a fixed set of dataproviders

A web component can specify in it's .spec file that it requires a foundset property and a fixed number of dataproviders from it. The foundset and required dataproviders are then selected by the developer when creating a solution (using the properties view, 'dataproviders' sub-property).

.spec file

```
"myFoundset": { "type": "foundset", "dataproviders": ["firstName", "lastName"] }
```

So the component has a property called "myfoundset" that it wants linked to any foundset chosen in ServoyDeveloper, and it needs two dataproviders from that foundset to be present in the foundset's property viewport.

browser js

Let's say the developer has chosen a foundset and selected for "firstName" a foundset dataprovider (for example a database column called parentFirstName) and for lastName another dataprovider (for example a database column called parentLastName). Those would generate for example a viewport like this inside the browser property:

Browser side provided property content in model

```
myFoundset: {
    (...)
    viewPort: {
        startIndex: 15,
        size: 2,
        rows: [ { _svyRowId: 'someRowIdHASH1', firstName: (...), lastName: (...) },
                { _svyRowId: 'someRowIdHASH2', firstName: (...), lastName: (...) } ],
        (...)
    },
    (...)
}
```

In this way any foundset dataprovider/column can be mapped to one of the two dataproviders that the component requires. The actual foundset dataprovider name is not even used in browser js.

Defining/using a foundset property that provides default formatting information for columns

A web component can specify in it's .spec file that it requires the foundset property to provide default formatting information for it's columns. We will use a foundset property with fixed number of dataproviders as an example, but it will work the same for other ways of specifying the dataproviders.

.spec file

```
"myFoundset": { "type": "foundset", "dataproviders": ["image", "age"], "provideColumnFormats": true }
```

So the component has a property called "myfoundset" that it wants linked to any foundset chosen in ServoyDeveloper, and it needs two dataproviders from that foundset to be present in the foundset's property viewport. For each of the two columns it will also receive default formatting information.

browser js

Let's say the developer has chosen a foundset and selected for "image" a foundset dataprovider (for example a database column called 'photo') and for age another dataprovider (for example a database column called 'estimatedStructureAge'). Those would generate a viewport and formatting information similar to the following inside the browser property (note that the column format actual contents might change as needed - this is what Servoy default components receive as well for their component properties):

Browser side provided property content in model

```
myFoundset: {
    (...)
    viewPort: {
        startIndex: 15,
        size: 2,
        rows: [ { _svyRowId: 'someRowIdHASH1', image: (...), age: (...) },
                { _svyRowId: 'someRowIdHASH2', image: (...), age: (...) } ],
        (...)
    },
    columnFormats: {
        image: {
            placeHolder: null,
            maxLength: 2147483647,
            isNumberValidator: false,
            edit: null,
            isMask: false,
            display: null,
            type: "MEDIA",
            allowedCharacters: null
        },

        age: {
            placeHolder: null,
            percent: "%",
            isNumberValidator: false,
            edit: null,
            isMask: false,
            display: "#,#00.###",
            type: "NUMBER",
            allowedCharacters: null
        }
    }
}
```

The formatting information is similar to what default Servoy components get for their format properties, so it could be used in a similar way (for example through $formatterUtils; for more details check out the source code - servoyformat.js, textfield.js).

Defining initial load and listener options for a foundset property

A web component can specify in it's .spec file that initially, at first show or each time the foundset gets completely refreshed it wants to automatically receive a number of rows in the viewport. This is useful to avoid some round trips between client and server and send data directly. It is configurable because some components may want to send no records initially while others might need to send many. This is done via the ***initialPreferredViewPortSize*** option. Default value is 50.

There is another option ***sendSelectionViewportInitially*** which allows a component to say wether this set of initial rows should contain the selected row (if any) or start at first row. The selected row will be in the center of this initial viewPort if possible when this option is "true". This option is "false" by default.

Both of these options can be altered at runtime by browser-side scripting using "setPreferredViewportSize(...)"; see above.

A foundset based component can specify (starting with Servoy 2022.03) if it wants to know when the foundset's definition was changed, by adding a **foundsetDefinitionListener** property to the foundset property with the value true. If that is true, then the foundset's **ChangeListener** will also get a foundsetDefinitionChanged event passed in when the definition (sql query) of the underlying foundset is changed. This can be handy if you are in a grouped mode and the root foundset only has 200 records loaded but through grouping you really show 1000 records and because of a filter that is applied to the root the first 200 are not changed but the change is somewhere visible after that. Then a grouping table should reflect that by refreshing the groups. Don't use this property if you don't need it because it is not without cost - to calculate the query change and fire the event.

.spec file

```
"myFoundset": { "type": "foundset", "dataproviders": ["image", "age"],
                                    "initialPreferredViewPortSize": 130,
                                    "sendSelectionViewportInitially": true,
                                                                      "foundsetDefinitionListener": true
 }
```

Linking other "foundset aware" property types to a foundset property

Other property types can have content that is 'linked' to foundset records in some way. These property types can be configured in the .spec file as shown below - linking to any other property they have defined with 'foundset' type. When they are linked to a foundset, their javascript value in the browser is no longer only one value, but a viewPort of values (or it will also contain a viewPort of values - the exact content is property type specific) - corresponding to the records loaded in the linked foundset property's viewPort.

In this scenario, the viewPort of the foundset value only contains '_svyRowId' if it's own .spec property configuration doesn't list a dynamic or static list of dataproviders ("dataproviders" : [...] or "dynamicDataproviders": true), and the "foundset aware" property type value will have the viewPort contents in it (check each "foundset aware" type to see how that works, as it could differ from type to type).

Component type (child components that are linked to a foundset - for tables, lists, ...) or custom object types built of/containing other "foundset aware" property types (let's call them 'configurations' - can be used to build lightweight pure HTML tables, lists, ...) are the most common uses in this area.

Examples of foundset aware types are 'component', 'dataprovider', 'tagstring'.

.spec file

One child ***component linked to the foundset*** :

```
"myFoundset": "foundset",
"childElement" : { "type" : "component", "forFoundset": "myFoundset" }
```

Multiple child components (array of them, notice 'elementConfig' that specifies a config value for each contained element) linked to the foundset. This type of linking is currently used by Servoy's tableviews, listviews and portals:

```
"myFoundset": "foundset",
"childElements" : { "type" : "component[]", "elementConfig" : {"forFoundset": "myFoundset"} }
```

Have a look at 'component' page to see how these two properties above will look like in browser js.

'Configuration' object for sending ***other "foundset aware" types*** as viewPorts follows. In this case the value of those properties - so for example 'myconfigurations[0].mydataprovider' or 'myconfiguration.mydataprovider' will be arrays representing the foundset's viewport, not simple values. If the property is really linked to the record (so not global/form variables but record DP/column) then it will get a special 'idForFoundset' value - for example 'myconfiguration.mydataprovider.idForFoundset' - in it as well; that can be used with the foundset property's sort API. This 'idForFoundset' was added in version 8.0.3. (Note: "forFoundset" usage for "dataprovider" does not yet allow changing the value, but there is a case for making that work)

```
"myFoundset": "foundset",
"myconfiguration": "MyConfig", // or:
"myconfigurations": "MyConfig[]"
(...)
"types": {
    "MyConfig": {
        "mydataprovider" : { "type" : "dataprovider", "forFoundset": "myFoundset"}
        "mytagstring" : { "type" : "tagstring", "forFoundset": "myFoundset"}
    }
}
```

Foundset change listener & other foundset linked properties (starting with 8.2)

In case you want to use a foundset property type change listener (for incomming changes from server) combined with other foundset linked properties such as dataproviders with "forFoundset", a change of a row on server will send changes both to the foundset property and to the dataprovider properties linked to that foundset. In order to make sure that your foundset notification update code executes after all property changes have been applied (so the dataprovider properties are also up-to-date) you can use:

```
var l = function(changes) {
    // wait for all incoming changes to be applied to properties first
    $webSocket.addIncomingMessageHandlingDoneTask(function() {
        // now check to see what actually changed and update what is needed in browser
        // because even other "forFoundset" properties are up-to-date
    }
};
$scope.model.myFoundset.addChangeListener(l);
```

Runtime property access

At runtime, the foundset property is accessible in (server-side) javascript. If a bean named "myFoundsetBasedBean" has a foundset property named "myFoundset" it can be accessed like this:

```
elements.myFoundsetBasedBean.myFoundset
```

That property gives access in scripting to:

- the real underlying ***foundset***
- to the ***dataproviders*** that the property will send to the client webcomponent (***dataproviders*** contains key-value pairs where key is the name of the column used in web component client side scripting and value is the name of the foundset column attached to that).

Both myFoundset. ***foundset*** and myFoundset. ***dataproviders*** are read-write properties under the foundset property type.

Setting myFoundset. ***foundset*** is only allowed if at design time you selected either *"- none -"* or a *separate foundset* for that property (so they are not related to the form directly). Parent form foundset and foundsets related to the form foundset are managed by Servoy automatically and they cannot be set through scripting at runtime. Of course you can alter the contents loaded by those form/related foundsets at runtime, but you cannot change completely the foundset by reference.

Examples:

```
// elements.myFoundsetBasedBean.myFoundset.foundset gives access to the
// underlying Servoy foundset used by this property
application.output(elements.myFoundsetBasedBean.myFoundset.foundset.getSelectedIndex())
elements.myFoundsetBasedBean.myFoundset.foundset.loadRecords(someQBSelect)

// elements.myFoundsetBasedBean.myFoundset.dataproviders gives access to the
// configured dataproviders of the Servoy foundset property; probably most useful
// in combination with dynamicDataproviders: "true"
elements.myFoundsetBasedBean.myFoundset.dataproviders = {
        dp1: "userNickname",
        dp2: "userReviewRating",
        dp3: "numberOfPurchasedItems"
}
if (!elements.myFoundsetBasedBean.myFoundset.dataproviders.rating)
        elements.myFoundsetBasedBean.myFoundset.dataproviders.rating = "userReviewRating";
```

(Starting with Servoy 8.1.3) Foundset typed properties can be assigned directly to as well. This will create a completely new foundset type property value (if you are not assigning a new foundset). Assigning a completely new foundset value to a foundset type property allows you to configure as well some of the things that are normally defined in the .spec file:

```
// elements.myFoundsetBasedBean.myFoundset is the foundset typed property
var myNewFoundset = ...; // some Servoy foundset
elements.myFoundsetBasedBean.myFoundset = {
            foundset: myNewFoundset,
            dataproviders: {
                dp1 : "customerName",
                dp2 : "city"
            },
            sendSelectionViewportInitially: false,
            initialPreferredViewPortSize: 15
};
```

All keys in the descriptor object above are optional except for "foundset". So if you don't provide "dataproviders" or "sendSelectionViewportInitially" or "initialPreferredViewPortSize" default values will be used for them. In a similar way you can simply set the foundset directly:

```
// elements.myFoundsetBasedBean.myFoundset is the foundset typed property
var myNewFoundset = ...; // some Servoy foundset
elements.myFoundsetBasedBean.myFoundset = myNewFoundset; // this will create a new foundset type property value
// that only sends the rowId (no other columns as dataproviders were not specified) and uses defaults for
sendSelectionViewportInitially and initialPreferredViewPortSize
```

Combining Foundset Property Type, Foundset Reference Type, Record type and client-to-server scripting calls

You might wonder - "why is setting a complete new foundset into a foundset typed property from server side scripting helpful?". This is helpful for example in implementing more advanced tree-like components, that need to operate with multiple foundsets.

In combination with Foundset Reference type ("foundsetRef"), Record Finder type ("rowRef") and calls from client-side scripting to server-side component scripting, such components can query/create foundsets on server on-the-fly according to different requirements, put them in the model of the component (for example in a foundset array property that is initially empty []). Then they also store in the properties the "unique id" using the Foundset Reference type and return that id as well from the server-side scripting call. This means that on the client it has access to the new foundset and it can identify it via the "unique id". Also if server-side scripting needs a record from a foundset that is already on the client to create it's new foundset (maybe they need to be related in some way), then all the client has to do is send to the server the foundset reference "unique id" together with the rowId (from the foundset property type's viewport) of that record and on the server you will be able to find the record using the Record Finder type.

Here is a partial example of what a tree-table might need to do in order to handle large amounts of data properly on all levels:

Client-side .js

```
function getChildFoundSetHash(parentFoundsetHash, rowId, parentLevelGroupColumnIndex,
                                                    newLevelGroupColumnIndex) {
        // parentFoundsetHash comes from the foundset referece type property
    // rowId comes from the foundset property type's viewport
    // parentLevelGroupColumnIndex and newLevelGroupColumnIndex are indexes in
    // an array property that holds dataproviders
    var childFoundsetPromise;

    if (newLevelGroupColumnIndex) {
        childFoundsetPromise = $scope.svyServoyapi.callServerSideApi("getGroupedChildFoundsetUUID",
                    [parentFoundsetHash, rowId, parentLevelGroupColumnIndex, newLevelGroupColumnIndex]);
    } else {
        childFoundsetPromise = $scope.svyServoyapi.callServerSideApi("getLeafChildFoundsetUUID",
                    [parentFoundsetHash, rowId, parentLevelGroupColumnIndex]);
        }

    childFoundsetPromise.then(function(childFoundsetUUID) {
        var childFoundset = getFoundSetByFoundsetUUID(childFoundsetUUID);
        mergeData(..., childFoundset);
    }, function() {
        // some error happened
        (...)
    });
}
(...)
function getFoundSetByFoundsetUUID(foundsetHash) {
    if ($scope.model.hashedFoundsets)
        for (var i = 0; i < $scope.model.hashedFoundsets.length; i++) {
            if ($scope.model.hashedFoundsets[i].foundsetHash == foundsetHash)
                return $scope.model.hashedFoundsets[i].foundset;

    return null;
}
```

Server-side .js

```
$scope.getGroupedChildFoundsetUUID = function(parentFoundset, parentRecordFinder, parentLevelGroupColumnIndex,
                                                              newLevelGroupColumnIndex) {
        if (!parentFoundset) parentFoundset = $scope.model.myFoundset.foundset;
        var childQuery = parentFoundset.getQuery();

        if (parentLevelGroupColumnIndex == undefined) {
            // this is the first grouping operation; alter initial query to get all first level groups
            (...)
        } else {
                        // this is an intemediate group expand; alter query of parent level for the child
level
                        childQuery.groupBy.clear();
            childQuery.groupBy.add(childQuery
                      .columns[$scope.model.columns[newLevelGroupColumnIndex].dataprovider]);
            var parentGroupColumnName = $scope.model.columns[parentLevelGroupColumnIndex].dataprovider;
            childQuery.where.add(childQuery.columns[parentGroupColumnName]
                      .eq(parentRecordFinder(parentFoundset)[parentGroupColumnName]));
        }

        var childFoundset = parentFoundset.duplicateFoundSet();
                childFoundset.loadRecords(childQuery);

                var dps = {};
                for (var idx = 0; idx < $scope.model.columns.length; idx++) {
                        dps["dp" + idx] = $scope.model.columns[idx].dataprovider;
                }

                $scope.model.hashedFoundsets.push({ foundset: {
                        foundset: childFoundset,
                        dataproviders: dps,
                        sendSelectionViewportInitially: false,
                        initialPreferredViewPortSize: 15
                }, foundsetUUID: childFoundset}); // send it to client as a foundset property with a UUID

                return childFoundset; // return the UUID that points to this foundset (return type will make
it UUID)
        };
```

For versions prior to **Servoy 8.2** please use **"*api*"** instead of **"*internalApi*"** below:

.spec file

```
        "serverscript": "mycomppck/mycompname/mycomp_server.js",
(...)
        "model":
        {
        "columns": { "type": "columnDef[]", "droppable": true },
        "hashedFoundsets": { "type": "hashedFoundset[]", "default": [] }
(...)
        "types":
        {
        "columnDef": {
            "dataprovider": { "type": "dataprovider", "forFoundset": "myFoundset" }
            (...)
        },
        "hashedFoundset" : {
                          "foundset": "foundset",
                          "foundsetUUID": "foundsetRef"
                }
        },
        "internalApi" : {
                "getGroupedChildFoundsetUUID" : {
                        "returns" : "foundsetRef",
                        "parameters" :
                        [{
                                        "name" : "parentFoundset",
                                        "type" : "foundsetRef"
                                }, {
                                        "name" : "parentRecordFinder",
                                        "type" : "rowRef"
                                }, {
                                        "name": "parentLevelGroupColumnIndex",
                                        "type": "int"
                                }, {
                                        "name": "newLevelGroupColumnIndex",
                                        "type": "int"
                                }
                        ]
                },
(...)
```

For Servoy 8.3 and higher:

In combination with Foundset Reference type ("foundsetRef"), Record type ("record") and calls from client-side scripting to server-side component scripting, such components can query/create foundsets on server on-the-fly according to different requirements, put them in the model of the component (for example in a foundset array property that is initially empty []). Then they return from the server-side scripting call the "foundsetId" using the Foundset Reference return type (so return a Foundset on an api call that has return type 'foundsetRef'). This means that on the client it has access to the new foundset and it can identify it via the "foundsetId" in the array-of-foundsets-property.

If **server-side scripting needs a record from a client-side foundset** in order to create it's new foundset (maybe they need to be related in some way), then all the client has to do is send to the server the row from client side foundset property's viewport and on the server it will automatically be translated to a Record by the 'record' property type that is used as argument. Once you have the Record on server you have the foundset as well via Record.foundset.

Similarly, **if one needs to send (from client-side) only a foundset as argument to server-side code**, it can just give the value of the foundset property to an argument of type 'foundsetRef' and it will automatically be translated on server to a Foundset.

Here is a partial example of what a tree-table might need to do in order to handle large amounts of data properly on all levels:

Client-side .js

```
function getChildFoundSet(rowObjFromFoundsetsViewport, parentLevelGroupColumnIndex,
                                                    newLevelGroupColumnIndex) {
    // 'rowObjFromFoundsetsViewport' comes from the foundset (that contributed the expanded row)
    //     property type's viewport so equivalent to something like
    //     $scope.model.foundsetProps[i].viewPort.rows[expandedRowIndex]
    // 'parentLevelGroupColumnIndex' and 'newLevelGroupColumnIndex' are indexes in
    //     an array property that holds the grouping column dataproviders
    // if 'newLevelGroupColumnIndex' is undefined, then we are requesting tree leafs (not groups)
    //     and then the server-side query is a bit different

    // foundset query needed for leaf level: Select pk from orders where Country = ? and City = ? and ...
    // foundset query needed for intermediate grouped level; ie. when you want to expand a country group
    //     to next level that is grouped by city: SELECT DISTINCT MIN(pk) FROM Customers Where Country =
    //     "Mexico" GROUP BY City;

    // as you can see below I try to send abstract things to the server (the client shouldn't really know
    // real datasource names, real column/dataprovider names and so on (those can be determined on server)
    // so both client and server code should be done so that these kinds of information never reach the
    // client in the first place - only as abstract ids or indexes/names of component properties)

    // so server needs to be given the expanded row (it can get the foundset of the Record from that) and
    // the groupColumn (index of the column) of the child level if that one is an grouped intermediate level
    // otherwise, if it is going to request leafs it should set undefined for "newLevelGroupColumnIndex"

    // NOTE: if we could get it nicely from the parent foundset's query there would be no use sending the
    // expanded node's group column because that is already available on the server from that foundset's
    // query (group by clause)

    var childFoundsetPromise;

    if (newLevelGroupColumnIndex) {
        childFoundsetPromise = $scope.svyServoyapi.callServerSideApi("getGroupedChildFoundsetId",
                    [rowObjFromFoundsetsViewport, parentLevelGroupColumnIndex, newLevelGroupColumnIndex]);
    } else {
        childFoundsetPromise = $scope.svyServoyapi.callServerSideApi("getLeafChildFoundsetId",
                    [rowObjFromFoundsetsViewport, parentLevelGroupColumnIndex]);
        }

    childFoundsetPromise.then(function(childFoundsetId) {
        var childFoundset = getFoundSetByFoundsetId(childFoundsetId);
        mergeData(..., childFoundset);
    }, function() {
        // some error happened
        (...)
    });
}
(...)
function getFoundSetByFoundsetId(foundsetId) {
    if ($scope.model.childFoundsets)
        for (var i = 0; i < $scope.model.childFoundsets.length; i++) {
            if ($scope.model.childFoundsets[i].foundsetId == foundsetId)
                return $scope.model.childFoundsets[i];

    return null;
}
```

Server-side .js

```
$scope.getGroupedChildFoundsetId = function(parentRecord, parentLevelGroupColumnIndex,
                                            newLevelGroupColumnIndex) {

        var parentFoundset = parentRecord.foundset;
        var childQuery = parentFoundset.getQuery();

        if (parentLevelGroupColumnIndex == undefined) {
            // this is the first grouping operation; alter initial query to get all first level groups
            (...)
            return;
        } else {
                        // this is an intemediate group expand; alter query of parent level for the child level
                        childQuery.groupBy.clear();
            childQuery.groupBy.add(childQuery
                      .columns[$scope.model.columns[newLevelGroupColumnIndex].dataprovider]);
            var parentGroupColumnName = $scope.model.columns[parentLevelGroupColumnIndex].dataprovider;
            childQuery.where.add(childQuery.columns[parentGroupColumnName]
                      .eq(parentRecord[parentGroupColumnName]));
        }

        var childFoundset = parentFoundset.duplicateFoundSet();
                childFoundset.loadRecords(childQuery);

                var dps = {};
                for (var idx = 0; idx < $scope.model.columns.length; idx++) {
                        dps["dp" + idx] = $scope.model.columns[idx].dataprovider;
                }

                $scope.model.childFoundsets.push({
                        foundset: childFoundset,
                        dataproviders: dps,
                        sendSelectionViewportInitially: false,
                        initialPreferredViewPortSize: 15
                }); // send it to client as a foundset property in the array of foundsets

                return childFoundset; // return the foudnsetId that points to this foundset (return type
                              // 'foundsetRed' will make a foundsetID from the childFoundset)
        };
```

.spec file

```
        "serverscript": "mycomppck/mycompname/mycomp_server.js",
(...)
        "model":
        {
        "columns": { "type": "columnDef[]", "droppable": true },
        "childFoundsets": { "type": "foundset[]", "default": [] }
(...)
        "types":
        {
        "columnDef": {
            "dataprovider": { "type": "dataprovider", "forFoundset": "myFoundset" }
            (...)
        }
        },
        "internalApi" : {
                "getGroupedChildFoundsetId" : {
                        "returns" : "foundsetRef",
                        "parameters" :
                        [ { "name" : "parentRecord", "type" : "record" },
              { "name": "parentLevelGroupColumnIndex", "type": "int" },
              { "name": "newLevelGroupColumnIndex", "type": "int" } ]
                },
(...)
```

$foundsetTypeUtils helper methods

Starting with Servoy 8.3.2 $foundsetTypeUtils service was added. It's purpose is to help make the foundset property type easier to use.

$foundsetTypeUtils

```
/**
 * NOTE: Starting with Servoy 8.4 you no longer need to use this method; see @deprecated
 * comment.
 *
 * The purpose of this method is to aggregate after-the-fact granular updates with indexes
 * that are relevant only when applying updates 1-by-1 into indexes that are
 * related to the new/final state of the viewport. It only calculates new indexes
 * for updates of type $foundsetTypeConstants.ROWS_CHANGED. (taking into account
 * any insert/delete along the way)
 *
 * @param viewportRowUpdates what a foundset/component property type (viewport) change listener
 * would receive in changeEvent[$foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES_RECEIVED]
 * [$foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES]
 *
 * @param oldViewportSize what a foundset/component property type (viewport) change listener
 * would receive in changeEvent[$foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES_RECEIVED]
 * [$foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES_OLD_VIEWPORTSIZE]
 *
 * @deprecated starting with 8.4 this is no longer needed as foundset/component/foundsetlinked
 * property change listeners guarantee that the rows in inserts and updates have their indexes
 * relative to the already changed viewport (data in the viewport at those indexes at the
 * moment these listeners trigger does match correctly). So basically calling this method would
 * not alter any update operations - they would remain the same.
 *
 * @returns an array of $foundsetTypeConstants.ROWS_CHANGED updates with their indexes corrected
 * to reflect the indexes in the final state of the viewport (after all updates were applied).
 */
coalesceGranularRowChanges: function(viewportRowUpdates, oldViewportSize);
```