Component (child) property type

Purpose of this property type

The 'component' property type can be used by web components to nest other web components inside them.

The nested components can be regular components - based of the parent's environment (foundset, ...) or linked to a .spec defined 'foundset' typed property, in which case they will receive data corresponding to records in that foundset.

The 'component' property value client side receives in JS all it needs to 'instantiate' a child web component and make it operational just like a normal form level web component.

Currently such child components will be available Servoy server side JS scripting in form.elements - as any other web components in the form at root level.

Servoy Developer's form editor handles adding/setting/removing child components through this property type behind-the-scenes. So the property itself will not appear in the properties view during development.

When component is not linked to a foundset

A child component that is not linked to another foundset, so it is only nested within another web component is easy to implement.

Let's say you want to create a component that has 2 child web components.

.spec file

```
    "model":
    {
        (...)
        "childComponent1": { "type": "component" },
        "childComponent2": { "type": "component" },
                (...)
    },
```

So our component's spec file defines two properties of type "component" for the two child components it wants to have. Note: some containers might want a variable number of child components - that can be done using a property of type 'component[]'.

html template

```
<div (...)>
    (...)
    <svy-component-wrapper component-property-value="model.childComponent1"></svy-component-wrapper>
    <svy-component-wrapper component-property-value="model.childComponent2"></svy-component-wrapper>
    (...)
</div>
```

The template of our parent component directive uses the svy-component-wrapper directive that Servoy provides to easily integrate child web components in the right place.

That's all you have to do; no special scripting is required.

NOTE: do not try directly <svy-component-wrapper component-property-value="model.childComponent1"/> as that will not work.

Listening for changes in "model"

You can listen for changes in the model of this component by defining in it a model change notifier similar to what is described here in the 'performance' section. So:

The model change listener

```
Object.defineProperty($scope.myComponentProperty.model, $sabloConstants.modelChangeNotifier,
                        { configurable: true, value: function(property,value) {

                switch(property) {
                    case "borderType":
                        (...)
                        break;
                    case "background":
                    case "transparent":
                        (...)
                        break;
(...)
```

Advanced usage of non-foundset-linked components

For advanced usage (most will probably never want to do this), if the parent component needs for some reason to manipulate child model/behavior, it can do that by specifying each attribute to be set on child separately - and it can intercept 'component' property content or provide there whatever it wants instead of directly the 'component' property type content. The template snippet below will produce identical results as the one above that only sets the "component-property-value" attribute:

```
<svy-component-wrapper tagname="model.childComponent1.componentDirectiveName"
     name="model.childComponent1.name"
     svy-model="model.childComponent1.model"
     svy-api="model.childComponent1.api"
     svy-handlers="model.childComponent1.handlers"
     svy-servoyApi="model.childComponent1.servoyApi">
</svy-component-wrapper>
```

Child components linked to a foundset

Portals, table view, list view use this. They have a set of 'component' properties (an array of them) which is linked to a foundset. The components represent a "row" logically. In this case the browser JS value for the properties will contain the needed data to build up one set of child components for each row in the 'foundset' typed property's viewport. See the 'foundset' property type page for more info about it's usage.

When components linked to a foundset are requested by a custom web component, that component will need to deal itself with how it creates child components in the browser (how will it visually display separate rows/columns of the foundset as child components) and how it links model/behavior between them and the 'component' typed property/properties. The 'component' typed property provides all that is needed to make that work.

.spec file

```
    "model":
    {
        (...)
        "myFoundset" : "foundset",

        "childElement" : { "type" : "component", "forFoundset": "myFoundset" }, // or
        "childElements" : { "type" : "component[]", "elementConfig" : {"forFoundset": "myFoundset"} },
                (...)
    },
```

Above we defined 2 properties: 'childElement" for one child component linked to a foundset, and 'childElements' as an array of child components linked to the given foundset. The foundset is specified using "forFoundset" configuration value. In case of the array property - the configuration value is specified for each element of the array using "elementConfig".

Foundset linked component property value in browser

In browser js, a component property value that is linked to a foundset has the following content (example contents of a child text field in a portal parent):

Browser side provided property content in model

```
childElement: {
    "componentDirectiveName": "servoydefault-textfield",
    "name": "shipname",
    "foundsetConfig": {
        "recordBasedProperties": ["dataProviderID"]
    },
    "model": {
        "enabled": true,
        "text": "Ship Name",
        "visible": true,
        "tabSeq": 0,
        (...)
    },
    "modelViewport": [{ "_svyRowId": ".null;5.10643;_0", "dataProviderID": "Alfreds Futterkiste" },
                      { "_svyRowId": ".null;5.10692;_1", "dataProviderID": "Alfred's Futterkiste 2" },
                      (...)]
    "handlers": {
        "onActionMethodID": function(args, rowId),
        (...)
    },
    "api": {
        "getSelectedText": function(),
        (...)
    },
    "servoyApi": {
        "startEdit": function(propertyName, rowId),
        "apply": function(propertyName, componentModel, rowId)
    },

    "addViewportChangeListener": function(listener),
    "removeViewportChangeListener": function(listener)
}
```

These contents can be used to generate what's needed and provide it to a "svy-component-wrapper" (see usage above) or can be used with other angular components out there that generate their own templates for individual components per record (such as uiGrid).

- **componentDirectiveName**: also present when not linked to a foundset; read-only; the directive tag name of the child web component.
- **name**: also present when not linked to a foundset; read-only; the name (property) of the web component.
- **forFoundset.recordBasedProperties**: not present when not linked to a foundset; read-only; a list of child component property names that are delivered for each record in the foundset viewport (these properties have different values for each record and will be available in **modelViewport**).
- **model**: also present when not linked to a foundset; read-write; properties of the child component that are the same for all records in the foundset's viewPort.
- **modelViewport**: not present when not linked to a foundset; read-write; model properties of the child component that have different values for each record in the foundset; this array's indexes correspond directly to the indexes of the linked 'foundset' type property's 'viewPort'; it's contents change when the 'foundset' property's viewPort change. See foundset property type documentation for more about how the viewPort is controlled.
  handlers: also present when not linked to a foundset; read-only; whatever handlers the web component has attached (chosen by developer at design-time). See discussion about '**rowId**' from **servoyApi. startEdit** below
- **api**: also present when not linked to a foundset; read-write; whatever API functions the textfield component provided; initially empty object. When component is linked to a foundset and the server side JS code calls such an API function it should get called in the end on the selected foundset record's child component that represents this component property (there is some support choosing to call on childComponents corresponding to all records, through a **forFoundset.apiCallTypes** entry, but that is not currently used nor documented here).
- **servoyApi**: also present when not linked to a foundset; read-only; Servoy specific API provided to be used with web components.
  - **startEdit(propertyName, rowId)**: provided by the 'component' property; it is used by the child component's svy-autoapply directive (if used by child component) or by custom child component directive code to signal that the editing of that cell has started; as you will probaly have one child component representing this 'component' property value for each record in the foundset's viewPort - startEdit requires receiving a '**rowId**' argument to know which record entered edit mode. For example the parent component creates a wrapper **startEdit (propertyName)** function (closure around the rowId of that record - that is always available in modelViewport as "_svyRowId" that will call the 'component' provided startEdit using that rowId) for each record and feeds it to the corresponding angular component. (the rowId argument is not relevant when using non-foundset linked components)
  - **apply(propertyName, componentModel, rowId)**: also present when not linked to a foundset; read-only; 'componentModel' and 'rowId' are only relevant when the component is linked to a foundset. '**componentModel**' is the real object that is given by parent component to the per-record child component at record 'rowId' as model - this is most likely made up of a combination between **model** and a **modelViewport** row. See discussion about '**rowId**' from **servoyApi. startEdit** above. Apply pushes a changed dataprovider value of the child component to the record on the server. It is used by svy-autoapply directive or directly by custom child component code.
- **addViewportChangeListener / removeViewportChangeListener**: discussed below (starting with 8.3.2).

Listening for changes in "modelViewport" (part of the model that differs based on record) - starting with Servoy 8.3.2

The property provides client-side (in browser) two methods: **addViewportChangeListener / removeViewportChangeListener**.

So you can add a listener that will receive updates (row insert/delete/change or full viewport update) similar to how the change listener in foundset property type works:

Adding a viewportChangeListener

```
var l = function(viewportChanges) {
    // check to see what actually changed in the component and update what is needed in browser
};
$scope.model.myComponentThatIsFoundsetLinked.addViewportChangeListener(l);
```

If you want your listener code to execute after all other properties (foundset property, other child component properties) get their updates, you can use something similar to this.

The "viewportChanges" 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. The value gives any extra information needed for that type of change:

what "viewportChanges" parameter can contain:

```
{

    $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROWS_COMPLETELY_CHANGED:  { oldValue : ..., newValue : ... },

    // if we received add/remove/change operations on a set of properties from modelViewport
    // corresponding to a record 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 or remove or change 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.
    //
    $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_UPDATES_RECEIVED: {

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

        $foundsetTypeConstants.NOTIFY_VIEW_PORT_ROW_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"
                    // on the foundset property
                    // 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"
                    // on the foundset property
                    // 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 : ...
            }
        ]
    }

}
```

In Servoy < 8.4, you might find what $foundsetTypeUtils provides useful depending on how you plan on using this listener. Starting with 8.4 granular updates are easier to use and you don't need to process those indexes anymore before using them. See comments above.

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 child component 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 (l && $scope.model.myComponentThatIsFoundsetLinked) $scope.model.
myComponentThatIsFoundsetLinked.removeViewportChangeListener(l);
                        });
```

Runtime property access

Since Servoy 8.0.3 , component type is scriptable. Type can be accessed like:

Scripting access

```
elements.mycomponent.childElements.mybean.beanProperty
```

Mycomponent is the name of the component which contains a childElements property of type component or component[] . From there, you can access the component inside type via name or index (if type is an array). Then you can access or assign properties of the mybean type.