Property Types

Not all property types are really meant for usage as api parameters, some of them are really meant for model property types only like:

dataprovider, tagstring (normal string will translate i18n, tags can't be resolved through api), media and valuelist (create a dataset out of a valuelist and with application.getValueListItems("vlname"))

Provided property value types

<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>boolean</td><td>Boolean value, either true or false</td></tr>
<tr><td>border</td><td>CSS border string representation</td></tr>
<tr><td>color</td><td>

String color value

Example

```
#FFFFFF
```

</td></tr>
<tr><td>dataprovider</td><td>

Reference to a dataprovider, either a record or scope/form variable.

Can become a complex definition if it needs to support onDataChange, it needs to be able to send changes to server or display tags.

The ondatachange handler will receive as parameter the old datarovider value, the new dataprovider value and a JSEvent.

Starting with 2021.06 release the JSEvent.data will contain an object: {dataprovider : dataProviderID, scope : dataproviderScope, scopeid : variable_scope_or_datasource}.

Definition

```
{
    "type": "dataprovider",
    "pushToServer": "allow", // optional: needed if you want this component
        // to be able to change the dataprovider's value on the server
    "ondatachange": { // optional
        "onchange": "onDataChangeMethodID", // the name of a handler
            // defined in spec
        "callback": "onDataChangeCallback" // the name of an API func
            // defined by the webcomponent that will get called when
            // pushed data change is successfully applied on server
    },
    "forFoundset" : "myFoundsetPropertyName", // optional; if specified
        // then on the client instead of one value you will get an
        // array of values one for each record in the viewport of given
        // foundset property (see also 'foundset' property type)
    "displayTagsPropertyName" : "myDisplaysTagsProp", // default is null.
        // If specified then the property will initially read the
        // boolean value of the property with given name; If that
        // property is true, tags (%%x%%) will be replaced, otherwise
        // they will be ignored (in which case it will just replace i18n
        // and html); if null, tags will be replaced or not based on
        // config option "displayTags"
    "displayTags" : false, // default is false. If displayTagsPropertyName
        // is non-null it will be ignored. Otherwise, only if it's value
        // is true, this property will parse tags (%%x%%)
        "resolveValuelist" : false // optional, default value is false
                // this is an optimization for read-only dataproviders
                // if true, the dataprovider will send directly the display value
                // from linked valuelist instead of real value; valuelist will
                // have no values sent to client; this way client doesn't need
                // to resolve the valuelist value anymore
}
```

</td></tr>
</table>

Starting with 8.4, if 'forFoundset' above is used, then the client-side (in browser) value also provides two methods: addChangeListener / removeChangeListener. 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. Remember to always remove any added change listeners when the component's $scope is destroyed - similar to what is described in the link for the foundset property type listener.

In order for dataprovider typed properties to 'push' their value changes to the server (so changes in the browser to end up in records or scope /form variables) you **must** either use autoApply or directly apply / startEdit. If you do not 'apply' (auto or manually) the change, the updates will not be sent to server.

IMPORTANT

In order to be able to change server-side dataprovider values using apply or svy-autoapply, the .spec file **must** declare that dataprovider property from the model as **pushToServer: allow** or higher. Otherwise the server will reject the dataprovider value updates and log a change denied warning.

Simple example of ***read-only*** dataprovider:

.spec

```
{
    "name": "my-component",
    "displayName": ...,
    (...)
    "model":
    {
            "myDataProvider" : "dataprovider",
            (...)
    },
    (...)
}
```

.html

```
<span ng-bind-html='model.myDataProvider'/>
```

Simple example of ***read-write*** dataprovider (with auto-apply):

.spec

```
{
    "name": "my-component",
    "displayName": ...,
    (...)
    "model":
    {
            "myDataProvider" : {
                   "type" : "dataprovider",
                   "pushToServer" : "allow"
            },
            (...)
    },
    (...)
}
```

.html

```
<input (...) ng-model="model.myDataProvider" svy-autoapply />
```

In server-side (solution) scripting, **accessing** a dataprovider property - for example "elements.myField.myDataprovider" - **will yield a string: the dataprovider id** (name of variable/column). You can also **assign a dataprovider id (string)** to that property.

Client side, the value of a dataprovider property depends on the actual value of the dataprovider (column/variable/...) as well as the type of the dataprovider (is it a number in the database, or a blob / string / date etc.).

	For primitives dataprovider types, it will be the primitive value. For date dataprovider types, it will be a Date object on client. For MEDIA typed dataproviders the client-side value depends on the server side value's content: - if the value is a BLOB / byte array then client-side you will get an object with two keys: { url: ... , contentType: ... } that can be used to load that content in the browser - if the value is a date, you will get a Date on client - if the value is a string: - if it starts with "media:///" it will be an URL string on the client that points to that server side media resource from the solution - other strings will be treated as strings - primitive types will just be sent 'as-is' to client

dataset	JSDataSet type equivalent. Currently can be used only for runtime api or model (to send a dataset to client).
	Example
	`"jsDataSet":{"type" :"dataset", "includeColumnNames": true,`
	`                               "columnTypes":{ "icon" : "media" }}`
	The type can have two configuration properties:
	1. includeColumnNames , default false, will also send the column names as a separate array to client as the first row
	2. columnTypes can specify some column types which will be used for converting the value to send client side, this is needed for non basic columns/types

date	Date value

dimension	Dimension representation
	Example
	`{`
	`        width: 100,`
	`        height: 20`
	`}`

double	A floating point number
font	CSS font string
form	URL string pointing to a form (like tab in tabpanel)

format	Format string.
	Given an object in the .spec file - to be able to specify which dataprovider and/or valuelist property to map this format property on. The ***'for'*** value can be one property name (pointing to a dataprovider-type property or a valuelist property), but it can also contain an array of 2 properties (one being a dataprovider-type property and the other a valuelist property).
	It needs the 'for' clause to be able to deduct the type of the format (format depends on DP type, whether or not the valuelist has real/display values or not, ...). If you put a valuelist property in the "for" and that valuelist property has itself a "for" that points to a dataprovider property (that might be linked to a foundset or not) then **you must** add that same dataprovider property to the "for" clause of the format property as well.
	Example 1
	`myFormatProp : { type: 'format', for: 'myDataProviderProp' }`
	Example 2
	`myFormatProp : { type: 'format', for: ["myValuelistProp", "myDataProviderProp"] }`
	Format property **access** from solution scripting will give the ***string value of the parsed format*** .**Assignment** to a 'format' typed property allows you to set a new ***format string*** .

<table>
<tr>
<td>function</td>
<td>

Callback function information that can be called on server from client.
A callback function set for this type of property will be represented in the component model as a pair of "formname" and "script" that can be used as parameters for "$window.executeInlineScript" to execute it.
Example:

in the component spec

```
        "model": {
                "myFunction" : { "type": "function"}
        },
```

in the component js implementation we use that executeInlineScript function that is exposed on the window object

```
$window.executeInlineScript($scope.model.myFunction.formname,
                            $scope.model.myFunction.script, [arg1, arg2]);
```

Note that the component need to have reference to the "$window" service, and the "formname" and "script" values are already encrypted in the model, and you need to use those values, and not plain text values for formname and script. Arguments for the callback can be set using the last parameter of "$window.executeInlineScript"

</td>
</tr>
<tr>
<td>int</td>
<td>An integer number</td>
</tr>
<tr>
<td>insets</td>
<td>

Padding insets representation

Example

```
{
        paddingTop: 10px,
        paddingLeft: 20px,
        paddingBottom : 10px,
        paddingRight: 10px
}
```

</td>
</tr>
<tr>
<td>JSEvent</td>
<td>type only used in handler call that go from client to server, see Specification (.spec file) - Servoy 2020 Documentation - Servoy Wiki</td>
</tr>
<tr>
<td>labelfor</td>
<td>Type for labelfor property of the label.</td>
</tr>
<tr>
<td>map</td>
<td>A map of key/value pairs. It's main purpose is converting string values, like "true"/"false" to boolean types, numbers as strings to number types (useful when using developer's properties view to assign values), and supporting i18n tags as values. It only supports primitive values; do not try nesting other objects/arrays/maps in it's values.</td>
</tr>
<tr>
<td>media</td>
<td>Reference to a Media entry. Media can be given in different ways on server (media name, media uuid, media url - for example "media:///servoy.jpg" -, media id). Client-side it will be an URL (string) that points to the media resource from the server.</td>
</tr>
</table>

<table>
<tr>
<td>record</td>
<td>

Reference to a Record. You can use this type:

- to send a Record identifier to browser/client code from server that can then be sent back to server later using the same type to become/be translated to the initial Record on server.
- (starting with Servoy **8.3**) to send a client/browser 'foundset' property type's row directly to server (using "record" property type for that); it will be translated to the actual Record on server. E.g.:

```
$scope.handlers.recordClicked($scope.model
                    .myFoundset.viewPort.rows[idx]);
```

- (starting with Servoy **8.3**) to send a client/browser 'foundset' property type's row to server using "foundsetProp.getRecordRefByRowID(id)". E.g:

```
// if the component uses the rowId internally to identify rows
var rowId = $scope.model.myFoundset.viewPort
                .rows[idx][$foundsetTypeConstants.ROW_ID_COL_KEY];
// (...)
// then it can just send a Record ref. to server using that
// ID directly without the need to get the actual row
$scope.handlers.recordClicked($scope.model.myFoundset
                    .getRecordRefByRowID(rowId));
```

See also Foundset property type - Combining Foundset Property Type, Foundset Reference Type, Record type and client-to-server scripting calls if you you are curious about how 'record' type can be used in combination with other types to create complex components.

</td>
</tr>
<tr>
<td>relation</td>
<td>Reference to a Relation (this is a string, so if used as a api parameter, this would just mean the relation name)</td>
</tr>
<tr>
<td>runtimecomponent</td>
<td>The type of component (so that components can be passed as parameters)</td>
</tr>
<tr>
<td>protected</td>
<td>

boolean security property, can be used to protect the entire component or specific properties or handlers in the component. For more information about 'protected' property type see this page.

Configuration:

<table>
<tr><th>setting</th><th>description</th><th>example</th></tr>
<tr><td>for</td><td>list of properties to protect,
when not specified the entire component is protected</td><td>"for": ["streetname", "updateInfoFunc"]</td></tr>
<tr><td>blockingOn</td><td>when the property is set to this value, protection is active, default: true</td><td>"blockingOn": true</td></tr>
<tr><td>blockingChanges</td><td>when the property is set to false changes sent for the component properties and handler calls are allowed, default: true</td><td>"blockingChanges": false</td></tr>
</table>

Example

```
"enabled" : { "type": "protected", "blockingOn": false, "default": true }
```

</td>
</tr>
<tr>
<td>point</td>
<td>

Point representation

Example

```
{
        x: 10,
        y: 20
}
```

</td>
</tr>
<tr>
<td>object</td>
<td>Generic JSON type. When using this type, you should pass primitive values or objects/array of primitive values and objects/arrays - because it will generally not be able to handle correctly other types of values (for example you cannot give a foundset or a border and expect them to work). You should use more specific types from this list whenever possible instead of this generic type. The 'object' type relies on standard JSON to transfer it's contents to the browser; it tries to use some default conversions to make the value be a JSON-valid value (but the safest way is for you to give only what JSON allows as value).</td>
</tr>
</table>

<table>
<tr><td>scrollbars</td><td>An integer value which represents scrollbar definition. This type is used for designer to display special editor.</td></tr>
<tr><td>string</td><td>Plain string property. I18n keys will be resolved by default.</td></tr>
<tr><td>styleclass</td><td>

String with space separated CSS classnames.Possible values supported by the component can be specified as hint for the developer

Example specifying special classnames supported by the component as hint for the developer

```
{
        type:'styleclass',
        values:[
                'btn',
                'btn-default',
                'btn-lg',
                'btn-sm',
                'btn-xs'
        ]
}
```

In Servoy version below 8.3 if you name the property "styleClass" that has the "styleclass" type also element.addStyleClass and element.removeStyleClass will work and will add or remove style classes from that property. With 8.3 this still works but you can mark any styleclass property type to be the "mainStyleClass" by adding the tag 'mainStyleClass' and set the value to true for this property:

```
{
        "type": "styleclass",
    "tags": { "mainStyleClass":true}
}
```

</td></tr>
<tr><td>tabseq</td><td>Tab sequence integer value. See sablo-tabseq</td></tr>
</table>

<table>
<tr>
<td>tagstring</td>
<td>

String property that can contain %%tags%%, be an i18n key or <html>. It can also be a simple static string.
Will be pre-processed before being provided to web component depending on configuration options below.

Examples:

Example

```
Hello %%name%%, welcome to %%i18n:com.mycompany.mykey%% ...
    or
i18n:com.mycompany.mykey
    or
<html>...some html that can also contain JS callbacks and media...</html>
    or
just some static string
```

Here is how this type of property can be configured in the component's .spec file:

Definition

```
myTextProperty : {
        "type" : "tagstring",
        "displayTagsPropertyName" : "myDisplaysTagsProp", // default is null.
        // If specified then the property will initially read the
        // boolean value of the property with given name; If that
        // property is true, tags (%%x%%) will be replaced, otherwise
        // they will be ignored (in which case it will just replace i18n
        // and html); if null, tags will be replaced or not based on
        // config option "displayTags"
    "displayTags" : false, // default is true. If displayTagsPropertyName
        // is non-null it will be ignored. Otherwise, only if true this
        // property will parse tags (%%x%%)
    "useParsedValueInRhino" : true, // default is false. Server side
        // scripting read/modify will work with: if false the parsed
        // (with tags/i18n already replaced, so static) value, if true
        // the non-parsed (value containing %%x%% or i18n:..., which
        // will be after modify parsed) value
    "forFoundset" : "myFoundsetPropertyName" // optional; if specified
        // then on the client instead of one value you will get an array
        // of values one for each record in the viewport of given
        // foundset property (see also 'foundset' property type)
},
displaysTags : { "type" : "boolean", "scope" : "design" } // needed only
        // if "displayTagsPropertyName" is used as above
```

or simply (to use default config values):

Example

```
myTextProperty: "tagstring"
```

Starting with 8.4, if 'forFoundset' above is used, then the client-side (in browser) value also provides two methods: addChangeListener / removeChangeListener. 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. Remember to always remove any added change listeners when the component's $scope is destroyed - similar to what is described in the link for the foundset property type listener.

</td>
</tr>
<tr>
<td>clientfunction</td>
<td>

This property must be used in NG2 if the value of the property represents a javacript function that must be run client side. In NG1 this was mostly just a (tag)string property type, but for ng2 we must know that it is meant to be a client side javascript function.

For NG1 it works the same as a tagstring property type. But for NG2 it is handled differently so the component can get a function directly from the model instead of getting a string and do an eval(string) on it to generate the function. This is because in NG2 it is not allowed to do eval in the browser by default, Content Security Policy doesn't allow that for security reasons.

</td>
</tr>
</table>

titlestring

String property similar to tagstring, but with support to use as default value the title string form a table column. It has a setting key: "for", that contains the dataprovider property from where the table column would be determined.

Configuration:

setting	description	example
for	dataprovider property name from where to get the table column's title for default value	"for": "dp"

Example

```
"headerTitle": {"type" : "titlestring", "for": "dp"},
"dp": { "type": "dataprovider", "forFoundset": "myFoundset", "resolveValuelist" : true}
```

valuelist

Reference to a ValueList.

Supported configuration (in .spec file, next to the type of the property)

setting key	description	example
lazyLoading	Added in 8.2 : Setting for Global Method Valuelists only (does not affect any other type of valuelist); it helps not load values on initial display (if not needed), but only when they are really needed. "Lazy loading" must be set on the Valuelist - in the Valuelist Editor as well, otherwise it has no effect. Default and bootstrap typeahead components have this configuration set to true (default is false).	`"lazyLoading": true`
max	Allows limiting the number of items the valuelist will send to the browser.	`"max": 500`
for	Allows you to provide a dataprovider-type property name (that this valuelist property will use to determine for example the type of the data - this is needed for some types of valuelist like custom valuelist)	`"for": "myDPProperty"`
`forFoundset`	Allows you to specify that this valuelist can contain different values for each record of a foundset property (for example if you have a valuelist based on a relation). That means that client-side/in browser you get an array of valuelist entries for this valuelist property, each item in the array corresponding to the row at the same index in the specified "foundset" typed property's viewport. If the valuelist that was chosen by developer does not have different values for each row, you still get an array but with the same valuelist at all indexes. Starting with 8.4, if 'forFoundset' is used, then the client-side (in browser) value also provides two methods: addChangeListener / removeChangeListener. 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. Remember to always remove any added change listeners when the component's $scope is destroyed - similar to what is described in the link for the foundset property type listener. @since Servoy 8.3.2	`"forFoundset": "myFoundset"`

Valuelist property **access** from solution scripting will give the ***valuelist name***. Assignment to a 'valuelist' typed property allows two types of values:

- if you **assign a String** to it that will be interpreted as a ***valuelist name*** - and the valuelist with the given name is put in that property
- if that property **already contains a custom valuelist** and you assign a ***Dataset*** to it then it will alter the items of the custom valuelist for that property of that element only.

For example:

```
// will output the name of the valuelist that is currently in "myValuelistProp"
application.output(elements.mycomponent.myValuelistProp);

// the following code would change the items of a custom valuelist inside one
// of the columns of a servoy-extra table component
elements.myExtraTable.columns[1].valuelist = databaseManager.convertToDataSet(
    new Array({ d: 'Item 1', r: 1}, { d: 'Item 2', r: 2},
              { d: 'Item 3', r: 3}, { d: 'Item 4', r: 4},
              { d: 'Item 5', r: 5}, { d: 'Item 6', r: 6}), ["d", "r"]);
```

NOTE: From **2022.09** on the valuelist property type at runtime did change a bit. Now it is more a complex object that has currently 2 properties:

name: to get the current name of the valuelist or to set it and then the valuelist of the given name is put in that property

<table>
<tr>
<td></td>
<td>
<p>dataset: this will return the dataset of the current values, or set the current values if it was a CustomValueList.</p>
<p>Client side, the valuelist will be an array of objects (items), each item will have two properties: "realValue" and "displayValue". If Valuelist is defined only having display values, realValue client side will be the same as display value:</p>
<pre>
$scope.$watch('model.valuelistID', function() {
    var valuelist = $scope.model.valuelistID;
    if (!valuelist || valuelist.length == 0) return;

    hasRealValues = false;
    for (var i = 0; i < .length; i++) {
        var item = valuelist[i];
        if (item.realValue != item.displayValue) {
            hasRealValues = true;
            break;
        }
    }
});
</pre>
<p>Client side, valuelist also has some API that can be used: filterList and getDisplayValue. Filter list api takes a filter parameter and applies it to valuelist server side, then filtered valuelist will be sent to client (in same property). This is used for typeahead component:</p>
<pre>
uib-typeahead="value.displayValue |
               formatFilter:model.format.display:model.format.type
               for value in model.valuelistID.filterList($viewValue)"
</pre>
<p>Get display value API is used to retrieve a display value taking real value as parameter from server. This is needed in cases where valuelist has many items and we don't send all items client side for performance reasons (see max property, also settings server side).</p>
<pre>
$scope.model.valuelistID.getDisplayValue($scope.model.dataProviderID)
.then(function(displayValue) {
        $scope.value = displayValue;
});
</pre>
<p>Both functions return a promise that will be resolved when response comes from server, will receive as parameter the new filtered valuelist (beware also valuelist from model will be filtered) respectively the display value.</p>
</td>
</tr>
<tr>
<td>visible</td>
<td>
<p>boolean security property, when set to false the component is protected from client data changes and function calls, data changes from the server are not sent to the client. For more information about 'visible' property type see this page.</p>
<table>
<tr><th>setting</th><th>description</th><th>example</th></tr>
<tr><td>for</td><td>list of properties to protect,
when not specified the entire component is protected</td><td>"for": ["streetname", "updateInfoFunc"]</td></tr>
</table>
</td>
</tr>
<tr>
<td>foundset</td>
<td>Used to interact with server-side foundset from the browser (component properties can have this type). See Foundset property type.</td>
</tr>
<tr>
<td>foundsetRef</td>
<td>Can be used to send a 'reference' to a server side foundset to the client. Useful mostly in handlers or server-side-API-calls as argument type or return type. If server-side scripting returns/gives a foundset to such a type, browser side scripting will receive an unique foundsetId/reference to that foundset. Browser/client side scripting can later send that id back to server-side scripting and it will automatically be transformed into the original foundset object by this property type. See also Foundset property type - Combining Foundset Property Type, Foundset Reference Type, Record type and client-to-server scripting calls. So the foundset that you want to pass a reference to should already be in the model on the client to send the reference back to the server. Its not use to use this as a foundset reference with a Server to Client call if you want to pass on the foundset as an api argument.</td>
</tr>
<tr>
<td>rowRef</td>
<td>
<p>Starting from Servoy 8.3, 'record' type should be used instead. See 'record' type above.</p>
<p><b>DEPRECATED: (to be used only in 8.2)</b> Also known as "Record Finder type". Can be used to send a rowId (from a foundset type property's viewport) from browser scripting to server side scripting. The property type will return a 'finder' function. When you call that function with the foundset as an argument it will return the record that corresponds to that rowId from the client. See Foundset property type - Combining Foundset Property Type, Foundset Reference Type, Record type and client-to-server scripting calls for and example of how it can be used.</p>
</td>
</tr>
</table>

Property value type modifiers:

Modifier	Description	Applicable for type
tags	See the Tags Section in Specification page.	

values	Fixed values, can have real/display values.
	Example with display and real values
	`[{"LEFT":2}, {"CENTER":0},{"RIGHT":4}]`
	Example with plain values
	`[`
	`    'btn',`
	`    'btn-default',`
	`    'btn-lg',`
	`    'btn-sm',`
	`    'btn-xs'`
	`]`	

default	Defines the default value being used.	
onDataChange	See dataprovider type	dataprovider
for	Reference to another property. Can be value or array of values.	

Note: defaults now also get applied to the model. This might change. Best practice is to initialize the model properties with values in the directive's link method and specify the same values as defaults in the .spec file for proper display in Servoy Developer.

See also:

- Array property type
- Component (child) property type
- Custom object property types
- Findmode property type
- Foundset property type