Custom object property types

Purpose of custom (object) property types

The custom object property type can be used by web components to logically group a number of sub-properties (that can be of different types) for easier usage.

Such custom object types are able to **send granular updates between server and client**. For example if you change from Rhino one subproperty value (or for more complex element property types - such as 'component' type - if something changes in only one of the subproperties) then only that change is sent over to the browser/client instead of the whole custom object (and the other way around - a subproperty change on client doesn't send the full object, just the change to server).

Custom object types are defined in .spec files with a fixed set of subproperties as below:

.spec file

```
    "model": {
        (...)
        "myPerson": "person"
                (...)
    },
    (...)
    "types": {
        "person": {
          "firstName": "string",
          "lastName": "string",
          "photo": "dataprovider"
          "styleClass" : { "type": "styleclass",
                           "tags": { "scope" :"design" },
                           "values": ["form-control", "input-sm", "svy-padding-xs"] }
        }
    }
```

"pushToServer" .spec setting of a custom object property **is currently automatically inherited by the sub-properties** of that property (so any "pushToServer" setting defined on sub-properties of the array will be ingnored). That means that for example if you define the custom object prop. to be "shallow" watched, all it's sub-properties will be shallow watched. If you don't define pushToServer or define "reject" then the sub-properties of that custom object will not be watched inside the browser and any changes to them will not be sent to the server.

You **should not use** pushToServer: **"deep"** on custom object property types, as that will actually add a deep watch on the custom object and any change within a sub-property of that custom object will be interpreted as an custom object prop. change and will send the full custom object value to the server. If you use "shallow" then the changes in custom object sub-properties are watched anyway (so in the end it is similar to a deep watch), so you normally do not need "deep" anyway; "deep" is meant more for properties of types like 'object' where you can have random JSON content that you want to be deep watched (so it sends it's whole value to the server for any change - so without granular updates).

Advanced .spec options

A configuration option (that you will most likely never need) for custom objects (available starting with Servoy 2019.06) is to be able to convert to null at runtime objects that have any of a set of specific keys set to null at design time (**"setToNullAtRuntimeIfAnyOfTheseKeysAreNull": ["a", "c"]**):

```
    "model":
    {
                "customObjConvertedToNullIfSpecificKeyIsNull": { "type" : "myCustomObj",
"setToNullAtRuntimeIfAnyOfTheseKeysAreNull": [ "a", "c" ] }
                (...)
        },
    "types": {
        "myCustomObj": {
            "a": "string",
            "b": "int",
            "c": "string"
        }
    }
```

For example if in developer properties view you set on this property { "a": null, "b": 5, "c": "something"} then, at runtime, the property will be null because "a", one of the "setToNullAtRuntimeIfAnyOfTheseKeysAreNull" from .spec, is null. You will probably never need to use this option except for when you want to create advanced custom components that have arrays of columns that contain child components (so *"model": { "columns": { "type" : "column[]", "skipNullItemsAtRuntime": true, "elementConfig": { "setToNullAtRuntimeIfAnyOfTheseKeysAreNull": ["columnComponent"] } }, "types": { "column": { "columnHeaderText": "string", "columnComponent": "component" } }*) and where the security settings of a form might not allow some of the child components (columns) to be visible at runtime - depending on the user that logs in. In that case those items in the array would be set to null automatically by Servoy because the *columnComponent* is set to null automatically if it's not supposed to be visible - and most of the times in this case you want to just get browser-side the columns who's child components you can show in that array and not worry about nulls and not send anything related to that column to client. See also array type to understand what *"elementConfig*" above does.

Browser/client side value

The browser value of such a property is a Javascript object containing the defined sub-properties:

```
{
    "firstName": "John",
    "lastName": "Doe",
    "photo": "https://...."
}
```

It is able to send granular updates (so if you change only one property it will only send that one), depending on it's pushToServer configuration.

Server side javascript value

The server side JS value of such a property is a custom implementation based on Javascript object - so you should be able to use it just like a normal JS object.

There is one difference though. In order to be able to send fine-grained updates to the client/browser, those values are 'watched'. That means that whenever you assign a completely new javascript object directly to the property (or if you assign a new object/array to one of it's sub properties on any level), that new value (reference) you assign will not be 'watched' directly; you have to take/read it back from the property (which will give you an equivalent 'watched' value) before using it further in code. Or you can access the value of the property directly every time, not kept as a reference.

Whenever you assign a full new value to that property, it will be replaced by a copy of it (starting with 8.2), *but the prototype of the copy will be the same as the one in the initial value*. That means that in that prototype you can have for example some methods if you want to build your component/service API like that and those methods - if they are defined in the prototype - will not be lost in the 'instrumentation' process.

For example:

DO it like this

```
var newPropertyValue = { mySybproperty2 : 10 };
// here you assign a new object to the property
elements.myCustomComponent.myObjectProperty = newPropertyValue;
// here you update the reference that you want to use later in code with the 'watched' new value
newPropertyValue = elements.myCustomComponent.myObjectProperty;

(...then later on, maybe during another event handler execution...)

// this modification will be detected because it's using the new 'watched' value you got from elements.
myCustomComponent.myObjectProperty after it was assigned - and the change will be sent to the browser
newPropertyValue.mySybproperty1 = 5;
```

OR like this

```
var newPropertyValue = { mySybproperty2 : 10 };
// here you assign a new object to the property
elements.myCustomComponent.myObjectProperty = newPropertyValue;

(...then later on, maybe during another event handler execution...)

// this modification will be detected because it's using the property value directly not through
'newPropertyValue'
elements.myCustomComponent.myObjectProperty.mySybproperty1 = 5;
```

DON'T do it like this

```
// DO NOT DO IT LIKE THIS
var newPropertyValue = {};
// here you assign a new object to the property
elements.myCustomComponent.myObjectProperty = newPropertyValue;

(...then later on, maybe during another event handler execution...)

// this will modify the value in newPropertyValue but myCustomComponent.myObjectProperty will not be aware of
that to send changes to client/browser
newPropertyValue.mySybproperty = 5;
```

Developer handling of custom object properties

Custom object properties can be edited at design-time from Servoy Developer's properties view and/or using drag and drop operations depending on type and configuration options. (TODO add more details here)

Nesting with custom object and array types

Custom object types can be nested with array types. This allows you to organize your model's properties better. For example (in .spec file):

```
    "model": {
        (...)
        "persons": { "type": "person[]" }
                (...)
    },
    (...)
    "types": {
        "person": {
          "firstName": "string",
          "lastName": "form",
          "profilePhotos": "dataprovider[]"
        }
    }
```

So the 'persons' property at runtime (client side) could look like this:

```
[
    { "firstName": "John", lastName: "Doe",
      "profilePhotos": [ "https://....", "https://...." ] },
    { "firstName": "Bob", lastName: "Smith",
      "profilePhotos": [ "https://....", "https://...." ] },
    { "firstName": "Jane", lastName: "Doe",
      "profilePhotos": [ "https://....", "https://...." ] },
]
```