# Client side logic (.js file)

This is the client-side .js file for the component. (running in browser)

It is the file declared in the .spec file under "definition".

The javascript file of the bean component is heavily linked to the specification and template of that component. Every web component is an angular module of its own, with at least 1 directive that describes the component itself and it's behavior. The module and the one directive should be named

like the bean (camel case notation so a servoydefault-name will result in servoydefaultName). The web component's module should also have a dependency to the 'servoy' module so that the web component can use the various servoy provided directives (starting with svy) and utilities.

Here is an example of such a .js file:

```
angular.module('servoydefaultTextfield',['servoy']).directive('servoydefaultTextfield', function() {
    return {
      restrict: 'E',
      transclude: true,
      scope: {
        model: "=svyModel",
        api: "=svyApi",
        handlers: "=svyHandlers",
        svyServoyapi: "=" // only needed if you really need access to this servoy provided API for more control
      },
      controller: function($scope, $element, $attrs) {
             // fill in the api defined in the spec file
             $scope.api.onDataChangeCallback = function(event, returnval) {
                     if(!returnval) {
                             $element[0].focus();
                     }
             },
                 /**
                  * Set the focus on the textfield.
                  */
             $scope.api.requestFocus = function() {
                      $element[0].focus()
             },

                 /**
              * Get the selected text.
                   * @return {string} the text selected in the field.
                   */
             $scope.api.getSelectedText = function() {
                     var elem = $element[0];
                     return elem.value.substr(elem.selectionStart, elem.selectionEnd - elem.selectionStart);
             }

          /**
                        * Set the selected text.
                        * @param {int} start the start index in the text
                        * @param {int} end the end index in the text
                        */
             $scope.api.setSelection = function(start, end) {
                     var elem = $element[0];
                     if (elem.createTextRange) {
                          var selRange = elem.createTextRange();
                          selRange.collapse(true);
                          selRange.moveStart('character', start);
                          selRange.moveEnd('character', end);
                          selRange.select();
                          elem.focus();
                     } else if (elem.setSelectionRange) {
                                elem.focus();
                                elem.setSelectionRange(start, end);
                     } else if (typeof elem.selectionStart != 'undefined') {
                                elem.selectionStart = start;
                                elem.selectionEnd = end;
                                elem.focus();
                     }
             }

         $scope.onClick = function(event){
           if ($scope.model.editable == false && $scope.handlers.onActionMethod) {
               $scope.handlers.onActionMethod(event);
           }
         }
      },

      templateUrl: 'servoydefault/datatextfield/datatextfield.html',
      replace: true
    };
  })
```

**It is a good practice to document your components, please note the jsdoc tags in the *code above*.** Servoy developer will use that JSDoc information from either the component *client-side scripting file* or from the component *server-side scripting file* to show tooltips in Solution Explorer, code editor and so on - for the API functions declared in it's .spec file. Some useful tags that can be inserted in jsdoc: **@example** (used when moving sample code from solution explorer), **@deprecated**. For information about documenting model properties and handlers please have a look at Documenting what properties do / Documenting handlers and handler templates.

First you can see the directive declaring that it uses a few things Servoy provides:

- svyModel: this is the object that contains all properties that the .spec file declares in it's "model" section.
- svyApi: this is the object that *the web component .js file must populate* with all the client-side API functions that the .spec file declares in it's "api" section. Some apis can also be implemented in serverside scripting, see "Serverside scripting" section of the Specification page.
- svyHandlers: this is the object that contains all handlers that the .spec file declares in it's "handlers" section. It is pre-populated by Servoy, so the handlers are already there, ready to be used.
- svyServoyapi: this is an API that Servoy provides to the component - if the component needs to do manual 'apply' or other operations. See the servoyApi section below.

The onDataChangeCallback function in the example above is used as a callback of the ondatachange spec configuration for a dataprovider.

The template/html for the component above looks like this:

```
<input type="text" style="width:100%; height:100%; background-color:{{model.background}};"
   ng-model="model.dataProvider" title="{{model.toolTipText}}"
    svy-autoapply svy-format="model.format" ng-click="onClick($event)"/>
```

There the various properties are taken from the model for specific html attributes. You can also see some servoy-provided directives used (the ones starting with "svy-").

A handler call (handlers.x()) to the server returns a promise (http://docs.angularjs.org/api/ng.$q) to which the web component can register a callback - so that an event/handler that executes on the server can return a value to the web component's call.

**Note about directive/WebComponent received attributes**

A WebComponent directive should expect that the attributes it receives (except for svyApi) can completely change.

For example when the record shown by a component changes the svyModel gets changed by reference. So be careful about caching model contents and accessing some model content only in the link method of the directive for example (which will not be called again when only the displayed record changes).

If you write any repeater components (such as custom portals/table views/list views) you should make sure you don't change the svyApi that you give to the same child WebComponent directive (and which was populated by that directive initially) when the record that it displays changes.

## ServoyApi

The servoyApi is a Servoy specific api which can be used by web components to interact with the server. It must be declared in the private scope of the component.

**component.js**

```
angular.module('mypackageComponent',['servoy']).directive('mypackageComponent', function() {
    return {
      restrict: 'E',
      scope: {
              model: "=svyModel",
          svyServoyapi: "="
      },
```

The **servoyApi** provides the following methods:

| Method | Parameters | Description |
|---|---|---|

| | | |
|---|---|---|
| **apply** | ***propertyName***<br><br>**–** the name of the property of type dataprovider | Pushes a changed dataprovider value of the component to the server (sets it in the actual record). So this is the only way to acutally make the server's data aware of browser-side changes to 'dataprovider' typed properties.<br><br>It is used internally by the svy-autoapply directive, but it can also be called directly by the web component itself.<br><br>For example the radio button manually pushes the new value to the server when the radio is clicked (without using svy-autoapply):<br><br>**radio.js**<br><br>`$scope.radioClicked = function()`<br>`{`<br>`  $scope.model.dataProvider =`<br>`  $scope.model.valuelistID[0].realValue;`<br>`  $scope.svyServoyapi.apply('dataProvider')`<br>`}`<br><br>**IMPORTANT**<br><br>In order to be able to change server-side dataprovider values using apply, 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. |

<table>
<tr>
<td>callServerSideApi</td>
<td>methodName<br><br>– the name of the serverside method to call<br><br>args<br><br>– the arguments of the serverside method</td>
<td>Used on the client side to call a function which is defined in the server side api of the component. Since Servoy 8.2, api must be defined in special category of the spec: internalApi<br><br>component_server.js<br><br><pre>"internalApi":
 {
     "mycallback": {
        "returns": "string",
        "parameters": [
            { "name":"name", "type":"string" },
            { "name":"type", "type":"string" }
        ]
    }
}</pre><br>It returns a promise of angular where the then function will give you the return value of the callback.<br><br>component_server.js<br><br><pre>$scope.mycallback = function(name, type)
{
    return "something";
}</pre><br>In the controller or link function of the component, "mycallback" can be invoked via<br><br>callServerSideApi:<br><br>component.js<br><br><pre>$scope.servoyApi.callServerSideApi("mycallback",
    ["string", "1"]).then(
        function(retValue) {
            console.log(retValue);
        }
);</pre><br>IMPORTANT<br><br>Beware that callServerSideApi does not send outstanding model changes to server, this should be handled by sending new values as parameters and modifying model server-side.</td>
</tr>
<tr>
<td>formWillShow</td>
<td>formname<br><br>– the name of the form which will be shown<br><br>relationname<br><br>– the name of the relation (optional)<br><br>formIndex<br><br>– the formIndex in the tabpanel (optional)</td>
<td>Prepare the form for showing. Example switching tabs in the tabpanel component:<br><br>tabpanel.js<br><br><pre>function setFormVisible(tab,event)
{
        if (tab.containsFormId)
      $scope.svyServoyapi.formWillShow(
             tab.containsFormId, tab.relationName);</pre><br>It returns a $q promise.</td>
</tr>
</table>

<table>
<tr>
<td>hideForm</td>
<td>formname<br><br>– the name of the form to hide<br><br>relationname<br><br>– the name of the relation (optional)<br><br>formIndex<br><br>– the formIndex in the tabpanel (optional)<br><br>formnameThatWillShow<br><br>– the name of the form to hide (optional)<br><br>relationnameThatWillShow<br><br>– the name of the relation (optional)<br><br>formIndexThatWillShow<br><br>– the formIndex in the tabpanel (optional)</td>
<td>Hides the form. The outcome of the returned angular promise is a boolean, which is true if the form was hidden.<br><br>Used by the tabpanel to hide the previously selected tab.<br><br>tabpanel.js<br><br><pre>$scope.select = function(tab)
{
    if (tab && tab.containedForm && !tab.active)
    {
                //first hide the previous form
                var promise =  $scope.svyServoyapi.hideForm(
                        $scope.model.tabs[$scope.model.tabIndex -1]);
                promise.then(function(ok)
                {
                        $scope.model.tabIndex = getTabIndex(tab)+1;
                        //show the selected form
                        $scope.svyServoyapi.formWillShow
(
            tab.containedForm, tab.relationName);
                tab.active = true;
                })
           }
}</pre><br>use the last 3 arguments to let the server directly know if the form that was give can be hidden then show immediately the other form and push that data . This way you won't notice stale data, because the new forms data is pushed sooner then when you ask in 2 points in time first to hide the current one and then to show the next one. The code would then be something like this:<br><br><pre>$scope.select = function(tab)
{
    if (tab && tab.containedForm && !tab.active)
    {
                //first hide the previous form
                var promise =  $scope.svyServoyapi.hideForm(
                        $scope.model.tabs[$scope.model.tabIndex -1],
          null, null,tab.containedForm, tab.relationName);
                promise.then(function(ok)
                {
                        $scope.model.tabIndex = getTabIndex(tab)+1;
                        //show the selected form
                tab.active = true;
                })
           }
}</pre></td>
</tr>
<tr>
<td>getFormUrl</td>
<td>formName</td>
<td>Return the URL of a form. It can be used together with ng-include to include a<br><br>form in the component template:<br><br>component.html<br><br><pre><div ng-include="svyServoyapi.getFormUrl(myFormName)" ></div></pre></td>
</tr>
</table>

| | | |
|---|---|---|
| **startEdit** | ***propertyName***<br><br>– the name of the property which is edited | Signal that the editing of a property has started, usually at focus gained.<br><br>It is automatically used internally by the svy-autoapply directive.<br><br>In case svy-autoapply is not used, **startEdit** can be called manually by the component itself to notify the server that the record should go into edit mode (by giving the name of the model 'dataprovider' typed property):<br><br>**component.html**<br>`<input ng-focus="svyServoyApi.startEdit('dataprovider')"/>` |
| **getFormComponentElements** | **propertyName**<br><br>– the name of the property form component property where the template should be get for<br><br>**templateUUID**<br><br>– the template UUID that the property has as its model value. | This api is used form component that use the "formcomponent" property type for 1 or more of its model properties. The model value should be given and the name of the property itself.<br><br>This method returns that a compiled dom elements which can be copied/appended into the right place in the dom of the component. |
| **isInDesigner** | | returns true when the component is shown in the designer at runtime this method will return false.<br><br>This way a component can show something more, like some sample data (a valuelist component can show a few rows of data so it displays nicely) |
| **isInAbsoluteLayout** (since Servoy 8.2.2) | | returns true when the component is in absolute layout form, false when it is in responsive form |