Specification (.spec file)

A (WYSIWYG) form designer needs component meta data to be able to handle components. This meta data is expressed in a specification.

- Definition and basic structure
  - Grouping of library dependency options
  - Deprecation support
  - Model
    - Array types
    - Custom types
    - Configuration options
      - Data synchronization
      - Tags
      - Default / Initial / Predefined values
    - Security
      - Protecting properties
      - Visibility properties
      - Container security
    - Enabled properties
    - Readonly/Findmode
    - Documenting what properties do
  - Handlers
    - Documenting handlers and handler templates
  - Api
    - Sync functions
      - Sync functions that do not block event processing
    - Async
      - Simple async functions
      - Async-now service functions
      - Delay-until-form-loads component functions
      - Discard-previously-queued-similar-calls functions
      - Allow server side calls when component or parent form is not visible
  - Server Side Scripting
  - Doc file
  - Palette categories
- Example

Definition and basic structure

A simple metadata specification is expressed in json format. (called the .spec definition file)

The specification defines:

- identifier information (component name, icon, display name, ...)

  > Name convention: the provided string is of the form package name, followed by a dash sign ('-'), followed by the component name. The package name is the name of the package the component is part of, and component name is simply the name of the component. Both package name and component name should be written in lower-case.

- dependencies (any extra js & css library dependencies that the component or service need to use)
- definition of component structure, to support:
  - model properties
  - event handlers
  - callable APIs
- additional/child property type info (information about custom types of data that can be used in model, api/handler parameters and return values)

In a json format:

spec file definition

```
{
        "name": "packagename-componentname", // String
        "displayName": "more descriptive name that is shown in the designer", // String
        "version": the component version (integer)
        "icon": "A reference to the icon shown in designer" ,
    "preview": "A reference to the preview gif to be displayed" ,

        "definition": "A reference to the js file that implements this component's in the browser",
    "serverscript": "[optional] A reference to the js file that implements this component's server-side logic,
if any.",
        "doc": "[optional] A reference to the js file that contains the jsdocs of the component api or model
properties.",
    "group": true // default true, so the definition file can be grouped when creating the .war file for
deployment
    "deprecated": "This component will be replaced in the next versions.", // (optional) some string to mark the
component as deprecated - if it is deprecated
        "replacement": "package-compname", // (optional) in case of deprecation, developer will provide a quick
fix for such components to be automatically changed to the given replacement component
                                        // (make sure they have compatible .spec defined; in most cases this is
useful when moving components from a package to another package or when
                                        // rewriting a component but keeping it's contract unchanged)
        "libraries": /* Array of js/css definitions (which are JSON objects with
                            'name'-the lib name, 'version'-the lib version, 'url'-the lib url,
                            'mimetype'-the lib mimetype, one of 'text/javascript' or 'text/css',
                            'group' - give false here when this lib dependency should not be grouped when
exported
                            as .war; default true)
                    that need to be included for this component. */,

        "keywords": /* Array of keywords used for searching components in the palette.
                                    For instance, for the calendar component some appropriate keywords that might
be used are: "day", "month", "year"*/,

    "categoryName": "Advanced", // category for form designer palette; only makes sense for components, not
services

        "model": {
            "property1Name": /* type description (optionally including optional default value and others) */,
            "property2Name": /* type description (optionally including optional default value and others) */
        },

        "handlers": {
            "handler1Name": { /* handler function type definition*/ },
            "handler2Name": { /* handler function type definition*/ }
        },

        "api": {
            "apiFunction1Name": { */ api function signature description json*/ },
            "apiFunction2Name": { */ api function signature description json*/ }
        },

    "internalApi" : {
            "internalApiFunction1Name": { */ internal api function signature description json*/ },
            "internalApiFunction2Name": { */ internal api function signature description json*/ }
    },

        "types": {
            "customType1Name": {
            "subProp1Name": /* type description" */,
            "subProp2Name": /* type description" */
        },
            "customType2Name": {
            "subProp1Name": /* type description" */,
            "subProp2Name": /* type description" */
       }
    }
}
```

A ng web component or ng service specifies all its properties in the **model** section of the .spec file, all the events under **handlers** section and the **api** section has the javascript callable api that the webcomponent exposes (that the server/solution can call).

Starting with 8.2 (before these were defined in "api") there is also the **internalApi** section that is better described in Server Side Scripting page (calls between client and server that are only meant for the code inside the component/service to use).

You can find an example .spec file below.

Grouping of library dependency options

The "group" property on the top level spec or in the libraries section tells Servoy if that the definition or library can be grouped or not; by default this is allowed. This is used when a WAR is generated by the WAR Exporter.

> **The "group" property**
>
> The libraries which contain references to external files cannot be grouped because in the deployed applications the relative paths to those resources are lost, therefore the components will not work.
>
> Such libraries are **font-awesome.css** or **tinymce.js,** they should always have *"group": false* in the specification file of the components that use them.

Deprecation support

A component/service/layout can be marked as deprecated by using the "deprecated" and/or "replacement" properties.

The deprecated components/layouts will not be shown anymore in the palette.

The deprecated services will not be shown anymore under the Solution Explorer plugins node. Markers will be added in case they are used in scripting.

The following combinations are possible:

```
"deprecated": "true" // a boolean as a string; this just means "deprecated" without any extra info (it is a string and not a boolean directly in case you want to provide more information about the deprecation; see below)
```

```
"deprecated": "This component will be removed in version X.", // some extra message explaining to the developer why it was deprecated or what to use instead
```

In the case when the "replacement" property is used (**supported for component level deprecation only**), the generated markers for the used deprecated components will also have a **quick-fix**.

```
"deprecated": "This component will be removed in version X.",//some extra message
"replacement": "packagename-compname"
```

Using only the "replacement" property will also mark the component as deprecated, and the generated marker will have a quick-fix.

```
"replacement": "packagename-compname"
```

The "deprecated" property

The "deprecated" property can also be used to deprecate service/component api or properties:

e.g. service function

```
"removeKL": {
    "returns": "boolean",
    "deprecated": "Please use removeKeyListener(...) instead.",
    "parameters": [ { "name": "callbackKey", "type": "string" } ]
 }
```

e.g. component property

```
"model": {
    "enabled" : {"type": "boolean", "default": true, "deprecated": "true"},
    ...
}
```

In this case, the deprecated property will not be shown in the properties view.

Model

A ng web component or ng service specifies all its properties and the type of each property in the **model** section of the .spec file.

Apis or handlers are not supported under "model" or "types" because these are just meant as container objects that transmit data/information between the component/service and the server.

For giving types of each property you can use any of the default provided property types that you can find here as well as any custom types defined in the "types" section. Arrays are also allowed (append "[]" to the type name).

Property types under **model** can be defined in two ways:

- simply by specifying it's type:

  ```
  "someTextProperty": "string"
  ```

- with additional **configuration options** if you want/need to tell Servoy more about how this property should be treated (this is just a sample, all configuration options are optional and each one will be detailed later); "type" is mandatory.

  ```
  "someTextProperty": { "type": "string", "tags": { "scope": "design" },
                                          "values": [ { "Predefined Text 1": "sample text 1" }, {
  "Predefined Text 2": "sample text 2" } ],
                                          "default": "nothing interesting" }
  ```

Array types

To define an array property just append "[]" to the type name.

```
"someTextProperties": "string[]"
```

To specify configuration options for each element of the array you can do:

```
"someTextProperties": { "type": "string[]", "elementConfig" : {
                                                "tags": { "scope": "design" },
                                                "values": [ { "Predefined Text 1": "sample text 1" }, {
"Predefined Text 2": "sample text 2" } ],
                                                "default": "nothing interesting" }
                                            } }
```

For more information on array types see the array property types page.

Custom types

types section defines custom internal types that can then be used by this web component/service for one or more of its properties in the model as well as for parameter and return value types defined in other sections. (for example a tabpanel component has a "tab" type that describes one tab inside the tab-panel). Custom type definitions each support for defining subproperties whatever is supported under **model.** For example:

```
    "types": {
        "someCustomTypeName": {
            "name": "string",
            "containsForm": "form",
            "texts": "tagstring[]",
            "relationName": "relation",
            "active": "boolean"
        }
    }
```

For more information on custom property types and how they work see the custom object property types page.

Configuration options

There is a set of standard configuration options and each specific property type (for more advanced types like "foundset", "dataprovider", ...) can have some extra configuration options that are detailed by each type.

Stardard tags are the ones that control **data synchronization**, **tags** and **default/initial/predefined values**.

Data synchronization

Data modifications are automatically propagated from server to client.

For performance (and security) reasons, data modifications from client to server are not propagated by default. To enable this configure *pushToServer* setting on the property.

pushToServer value	description
reject	this is the default, no data synchronization from client to server for this property
allow	data changes from client to server are allowed
shallow	same as *allow*, but sablo will also add a watch on the model (in the client) and send changes to the server; the watch is based on object reference/primitive value (more performant, but in case of structured/nested objects it will only trigger data synchronization when the objects are different by reference, even if they are actually 'equal' by content)
deep	same as *allow*, but sablo will also add a watch on the model (in the client) and send changes to the server; the watch is based on object equality (compares old and new values of structured/nested objects, less performant because it keeps and compares two copies of structured/nested objects); "deep" is mostly meant to be used for properties of type 'object' - that can contain nested JSON - and that for any change (doesn't matter how little) will send over the complete value to the server.

Information about how "pushToServer" works in particular with Array property type and with Custom object property types can be found on the wiki pages of these property types.

For example:

pushToServer example

```
"model": {
   "searchfield": { "type": "string", "pushToServer": "shallow" }
}
```

Note that in Servoy, data-provider property changes are sent to server using svy-apply (servoy api call) and svy-autoapply (directive); for these to work properly, these properties should set pushToServer to *allow*. Otherwise those data provider properties will be read-only.

Tags

Properties can be configured with special tags that may be interpreted by the deployment and/or development environment.

Supported tags are: **scope, doc, addToElementsScope**, **logWhenOverMax**, **allowaccess**, **directEdit**, **useAsCaptionInDeveloper** + **captionPriority**, **showInOutlineView, main** and **mode**.

scope: Restricts the model property to: '**design**', '**runtime**' or '**private**'.

Design means property can only be assigned from designer (not from scripting). Runtime means the property cannot be assigned from designer (will be hidden in Properties View). Private is an internal property, that should only be used by component itself (so component.js or component_server.js). Will not show in Properties View and cannot be assigned from scripting.

doc: a string (can have some basic html tags in it) that describes what the property does to the developer. See Documenting what properties do for more details.

addToElementsScope : boolean

For component type, specify whether component should be added to elements scope. Default is false, so component can be accessed only via component property. If true, will be accessible like any other element of the form.

logWhenOverMax: boolean

For the valuelist property type. If set to false, no logging will be done when only sending "max" number of items to the browser even though the valuelist does contain more than "max" items. Default is true.

allowaccess: string or array of strings

This tag can define if an edit (sent from browser to server) of this property is allowed even if properties such as "visible" or "enabled" (of the component or parent form) are false. By default, if the visibility of a component or its parent form is false, Servoy doesn't allow the modification (coming from browser) of any other property (it blocks it and logs it on server). With this tag you can say for a property that changing it should be allowed even in these situations; for example on "size" property you could say: I allow it for "visible"; or for both: ["visible", "enable"]

directEdit: boolean

One property of a component can be tagged as directEdit for designer, this way that property can be edited directly by double clicking the component (for example text property on label/button).

useAsCaptionInDeveloper : boolean and **captionPriority** : integer (> 0) (starting with Servoy 8.3)

Can be used to provide a caption for Custom Object Types in developer based on the value of a subproperty. For example if you create a table component that defines in .spec custom object types (that could also be "droppable" in developer) for columns, you might want to show as caption in developer for each such 'column' the header text subproperty value or - if that is not set - the dataprovider subproperty value. The captionPriority tag will allow you to specify in which order subproperties are checked for non-empty string values for the caption. That caption text ends up visible in places such as the outline view or in form designer for each column. You would specify it like this:

(...).spec

```
    (...)
    "model": {
        "columns": { "type": "column[]", "droppable": true, (...) },
    (...)
     "types": {
        "column": {
            "dataprovider": { "type": "dataprovider",
                              "tags": { "useAsCaptionInDeveloper" : true,
                                        "captionPriority" : 2 }, (...) },
            "headerText": { "type": "tagstring",
                            "tags": { "useAsCaptionInDeveloper" : true,
                                      "captionPriority" : 1 }},
    (...)
```

showInOutlineView: boolean (for Servoy < 8.3)

In the Outline View, Custom Type objects have the name of the type as lables. The showInOutlineView:true tag can be added to any property definition in order to append the design time value of that property to the label of the custom type object in the Outline View. If you are using Servoy >= 8.3 please use "useAsCaptionInDeveloper " and "captionPriority" instead. Starting with that version "showInOutlineView" : true is equivalent to "useAsCaptionInDeveloper" : true.

main: boolean (starting with Servoy 8.4)

Used only for dataprovider types. In case there are more then one dataprovider types, setting this tag to 'true' will define the main dataprovider, that is needed on different places, like sorting. As an example, having a component with 2 dataprovider properties, and using it in a table view, that component is used for column rendering, and calling sort on that column, by clicking on the column header, will need the dataprovider from the component to do the sort - using the 'main' tag, we can define the dataprovider to use. (see onrenderlabel component from servoy-extra package)

mode: string (starting with Servoy 2019.03) - Restricts the model property to: '**combobox**' or '**typeahead**' in developer properties view. (Default is combobox)

This can be used only on string properties when they have "values" attribute. Setting mode to typeahead will create a typeahead property in property editor. Setting mode to combobox or skipping mode property will create a combobox in property editor.

```
"someTextProperties": { "type": "string",
                                          "tags": { "scope": "design", "mode": "typeahead" },
                        "values": [ "off", "shipping", "billing", "phone", "address"]
                      }
```

Default / Initial / Predefined values

A property can have a "default" value like:

```
"text": { "type": "string", "default": "I am a button" }
```

But this would mean that restore to default or empty the property would always set that default property back. This is wanted behavior if the property has to have some default value like if a property must be 1 of a few values and the default is one of those.

If a property should allow the developer to choose in the properties view from a predefined set of values you can use the "*values*" attribute in combination with *"default"* and optionally "*mode"* tag (see above) (default can be one of the predefined values but can also be something else):

```
"horizontalAlignment": {
    "type": "int",
    "tags": { "scope": "design" },
    "values": [
        { "LEFT": 2 },
        { "CENTER": 0 },
        { "RIGHT": 4 }
    ],
    "default": -1
},
"mychoice": {
        "type": "string",
        "values": ["yes","no"]
}
```

But for a button or label in which you initially want some text, but reverting should really clear the text, the default value is not really usable (the label text should be able to be nothing, in case only an image is meant to be shown in that label). For this we have the attribute "initialValue":

```
 "text": { "type": "tagstring", "initialValue": "Button text", "tags": { "directEdit": "true" } }
```

This is like a constructor value, set only once when the component is added.

Security

Components can be protected using two special security types: visible and protected.

For example. when a component is made readonly from the server the component will make the its UI non-editable. However, a malicious client could still send json messages to the server and update data; that is what this properties avoid server-side.

Similarly, when a form or component is marked invisible, its data should not be sent to the client since it may contain sensitive data.

More information about how these properties work with containers can be found here.

Protecting properties

Protecting properties (of type `protected`)are used to protect an entire component or specific properties or handlers.

When the property is blocking no changes or function calls from the client are accepted.

For example, the `editable` property as defined below will be true by default, when set to `false` changes from the client to the component will be rejected. Also functions cannot be called from the client.

Example editable

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

Protection can be done for specific properties or functions.

In this example, when `protectCustomer` is set to `true`, `customerAddress` can still be changed the the client, but `customerName` and `removecustome` are protected.

Example protecting properties and handlers

```
"model:" {
    "customerAddress": "string",
    "customerName": "string",
    "protectCustomer": { "type": "protected", "for": ["removecustomer", "customerName"] }
   },
"handlers:" {
   "removecustomer": "function"
}
```

Protecting properties themselves can never be modified from the client.

You might also want to read about how protecting works with containers below.

Visibility properties

Visibility properties (of type `visible`) are similar to protecting properties. They are protecting the component and also hide the data from the client if the component is not visible.

In this example, a component's model can be filled with a customer name, but when the property `visible` is set to `false`, the component will be protected and the data will not be sent to the client. When `visible` is set to `true`, data not sent before will be pushed to the client.

Example visible

```
"model": {
   "visible": "visible",
   "customerName": "string"
}
```

Visibility properties themselves can never be modified from the client.

You might also want to read about how visibility works with containers below.

Container security

Protecting and visibility properties on containers will protect the container and also components inside the container

For example *forms are containers* and the visibility of a form will protect the components inside it as well. So only when server side makes a form visible or a web component (container) from the browser asks that it wants to make a form visible (via servoyApi.formWillShow, provided it does have access itself to that form) that form will be accessible to the browser. Container-like web components can only make forms visible if they have those forms referenced in their (sub)properties (so basically server-side gave them access to be able to make those forms visible).

Enabled properties

Similarly to properties of type `visible` , the properties of type `enabled` are also protecting properties, so they can never be modified from the client.

Example enabled

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

It is important to use type `enabled` if we want the value from the parent container (i.e. form, portal) to be pushed to the component. For instance if a portal is disabled, then all the components from the portal which have a property of type `enabled` will also be disabled.

Readonly/Findmode

Servoy has 2 special controller/foundset based properties where a component also can be notified for.

controller.readOnly = true in the developer will set this boolean to a property called "**readOnly**" of a webcomponent, this property should be a runtime or even a hidden property. It should not be a design time property, because the system can set it at any time (to true or false).

If you want also a design time property to control the editable state then add a second property, see example below, and then having a tag in the template like: ng-readonly="model.readOnly || !model.editable"

Listening to readonly

```
"model": {
  "readOnly": { "type": "protected", "blockingOn": true, "default": false, "tags": {"scope": "runtime"} },
  "editable": { "type": "protected", "blockingOn": false, "default": true }
}
```

With this property a component can do its thing to set or unset the readonly mode for itself.

It's better to have type this property as "protected" because it should only be able to change at the server, never from the client.

See as an example our bootstrap textbox: https://github.com/Servoy/bootstrapcomponents/tree/master/textbox

findmode is a special type: Findmode property type which can be used to set all kind of other properties which are normally protected from changing on the client side. Or you can just use it as a type for any property you want:

Listening to findmode

```
"model": {
  "myfindmodeproperty": "findmode"
}
```

When the find mode changes the boolean value of your property will also change. This way you can react to a form/foundset going into find mode. Like resetting a specific format, allowing any kind of input.

Documenting what properties do

Documentation for properties can be added to each property's definition via the **"tags"** section using key **"doc"** or in the doc file using a variable with same name as the property.

```
/**
 * some desciption
 * @example elements.%%elementName%%.yourName = 'myname'
 */
var yourName;
```

The description that you provide in the .spec file will be used in Servoy Developer as:

- tooltip in properties view
- tooltip in solution explorer view
- tooltip in script editor
- any other place in developer where it can help the user of your custom component understand what that property does.

For example if you want to document a simple property called "titleText" you can do it like this:

```
	(...)
	"model": {
		"titleText": { "type": "string", "tags": { "doc": "The <b>title text</b> is shown in this
component's title bar (top side).<br/>Keep it short." } },
	(...)
```

Basic html tags are supported - similar to the ones supported when documenting methods using JSDoc (in the script editor).

Arrays, custom objects, array elements and subproperties of custom objects can be documented in the same way. For example:

```
	(...)
	"model": {
		"columns": {
			"type": "column[]",
			"tags": { "doc": "Define the table's columns using this property." },
			"elementConfig": {
				"tags": { "doc": "A column definition describes all that is needed to show that
column properly in the table." }
			},
			(...)
		},
		(...)
	},
	"types": {
		"column": {
			"dataprovider": { "type": "dataprovider", "forFoundset": "foundset", (...), "tags": {
"doc" : "Choose the data that is to be shown in this column." } ),
			"format": { "type": "format", "for": ["valuelist", "dataprovider"], "tags": { "doc" :
"This format will be applied on the dataprovider's data before showing it in the table." } },
			"valuelist": { "type": "valuelist", "for": "dataprovider", "forFoundset": "foundset" },
			(...)
		},
	(...)
```

For information about documenting handlers see Documenting handlers and handler templates below. For information about documenting API functions see the documenting api functions example.

Handlers

The function description in the **handlers** section can be just **"function"** or a more complex type describing the arguments and return type:

```
"functionName": {
        "returns":  "string",
        "parameters": [
            { "name": "start", "type": "int" },
            { "name": "end", "type": "int" }
        ],
                "private": true // since Servoy 8.3, optional, default is false,
                "ignoreNGBlockDuplicateEvents": true // since Servoy 2020.12, default is false
}
```

The **"private"** configuration makes the handler only accessible from Server Side Scripting, not from the client/browser itself.
The "**ignoreNGBlockDuplicateEvents**" configuration makes the handler ignore the system NG_BLOCK_DUPLICATE_EVENTS property.

A handler can have a **JSEvent** property type which is able to map a Dom event to a JSEvent object that the handler can have in Servoy scripting. A mapping is mostly made automatically by calling using in the template:

```
svy-click='handlers.onActionMethodID($event)'
```

If you need to make one manually you can also do that by using the $utils factory service

```
$utils.createJSEvent(e,"onselectionchanged")
```

Documenting handlers and handler templates

Handlers can be documented - for use inside the developer - using the following keys (these are used in properties view tooltip or when generating new handler methods):

- **doc** (used to be description; describes when the handler will be called and what it's for)
- **code** (when a new method is created for that handler, this default code template will be inserted)
- for return value details **returns** can be an object with sub-keys: { type, **doc**, **default** (only relevant if "code" was not given) }
- for parameter details **parameters**: [{ **doc**, **optional** (is false by default) }].

For example:

```
"onDataChange": {
        "returns":  { "type": "boolean", "doc": "if it returns true then the data change is considered to be
valid, otherwise it will be blocked/reverted by the component", "default": true },
        "parameters": [
            { "name": "newValue", "type": "string", "doc": "the new value entered by the user" },
            { "name": "oldValue", "type": "string", "optional": true, "doc": "the previous value if available" }
        ],
                "doc": "<b>onDataChange</b> will be called if the user modified the field's content and then
either tabbed out/clicked outside of the field or hit enter. The handler can decide wether to allow the change
or not.",
                "code": "// if we set this we should remove the 'default' from 'returns'\nreturn !!newValue;"
}
```

For information about documenting properties see Documenting what properties do above,. For information about documenting API functions see the documenting api functions example.

Api

This section defines the signatures and behaviors of functions that can be called from the solution on this component/service.

The signature description json for functions in the **api** section describes the arguments, type of call and return type, For example:

```
"someFunctionName": {
        "parameters": [
            { "name": "startIndex", "type": "int" },
            { "name": "endIndex", "type": "int" },
            {
                "name": "theData",
                "type": {
                    "type": "dataset",
                    "includeColumnNames": true,
                    "columnTypes": { "icon": "media" }
                }
            }
        ]
}
```

The implementation of such api functions can be located either in the browser-side component/service logic implementation ("definition" in .spec file) or in the server-side component/service logic implementation ("serverscript" in .spec file)

There are several types of sync/async api calls described below.

About calling browser-side api functions of components

When a **component (not service) sync or simple async api function** (see below) that the solution can call is implemented in **browser-side** component/service logic implementation ("definition" in .spec file), it is important to note that calling such a function when the form of that component is not already created in browser DOM will result in a temporary "force-load" of that form in a hidden div in the browser - just to be able to call that api function. As this is usually not useful and will slow-down the solution due to the hidden loading of a form, this situation should be avoided (either by the solution - calling the api call later after the form was shown - or, where possible, by using delay until form loads async api functions in components - see below).

Servoy will log warning messages each time a sync API call to browser executes when the browser doesn't have the needed form present in DOM yet (triggering a force-load of the form in a hidden div). Most of the times this happens when solutions call component sync api functions to browser inside the onShow handler of a form.

For information about documenting API functions see the documenting api functions example.

Sync functions

This is the **default type of api function**; the example above is a sync api function definition. Sync functions will call the client and wait for the api function to execute/return before continuing. Sync api functions can have a return value.

Client side code of the sync api function can return either directly the intended return value or a Promise that in the end resolves with the intended return value. Server will wait for any client returned Promise to resolve before resuming server-side code execution.

In case of component sync functions, if the form is not present in browser's DOM then sync calls will force-load it in a hidden div just in order to execute the sync function on the component.

Sync functions that do not block event processing

A special kind of sync api function type is sync functions that **do not block** event processing. This is not needed - in almost all cases.

It can be defined by adding a **"blockEventProcessing" : false** (that defaults to true) to the function definition. When it is set to false, Servoy will expect that this API function call will be long-running (so it will not time out if it takes long to execute) and at the same time it will allow other events to be handled by the event dispatcher while waiting for the return value. This is helpful for example if a component would have an API function call like "askUserForSomeValue (someForm, ...)" which shows a modal dialog containing a form and returns a value. That API call has to wait for the return value an indefinite amount of time without blocking other events from being dispatched. Example:

```
        "askUserForSomeValue":
        {
            "blockEventProcessing": false,
            "returns": "string",
            "parameters": [ { "name": "formToShow", "type": "form" }, { "name": "dialogTitle", "type": "string",
"optional": "true" } ]
        }
```

Async

Async api calls will allow the calling code to continue executing and will execute the api call later/decoupled.

Async api functions cannot return a value.

There are 3 types of async calls.

Simple async functions

Simple async calls - when called from solution (server) code - will get executed in browser later, when current request to server is done.

Just add **"async": true** (another optional parameter (that by default is false)) to the call definition; it means the client side api is not called right away, but at next message sent to the client.

In case of component async functions, if the form is not present in browser's DOM when the async call is supposed to execute (later) then it will force-load it in a hidden div just in order to execute the async function on the component.

```
"executeSomethingLater":
{
    "async": true
}
```

Async-now service functions

These are currently supported only for ng services, not components. These are async functions that will be called in browser right away (rather then when the current request to server is done), but the code that calls them does not wait for them to finish execution before proceeding.

They are useful when for example you want to send progress information to the service client-side while executing a long-running-operation server-side.

Async-now function calls will not send other pending async calls to client nor update the client's state with other pending changes (model updates of components/services/etc.).

Just add **"async-now": true** (another optional parameter (that by default is false)) to the call definition.

```
"sendProgress":
{
    "async-now": true,
    "parameters": [ { "name": "percent", "type": "int" } ]
}
```

Delay-until-form-loads component functions

These are special types of async calls meant only for components, not services. These functions, when called, will execute later (similar to simple async, when a request to server is done executing) but only if the form is loaded inside the browser.

If the form is not loaded inside the browser they will wait for the form to get loaded (shown by solution somewhere) before being executed.

You can use these to avoid accidental calls from the solution to execute the api function without the solution intending to show the form. The difference compared to simple async functions is that simple async functions will load the form - if not already loaded in browser - in a hidden div just in order to execute the api call... this can be time-consuming and is usually not needed (a form that is not shown just being loaded and discarded in order to call an async function).

Just add **"delayUntilFormLoads": true** (another optional parameter (that by default is false)) to the call definition. This flag was also known previously as "delayUntilFormLoad" (but that is now deprecated).

```
"initializeMyComponent":
{
    "delayUntilFormLoads": true,
    "parameters": [ { "name": "initialProgress", "type": "int" } ]
}
```

Discard-previously-queued-similar-calls functions

This extra-flag **only makes sense in combination with** either **"async"** or **"delayUntilFormLoads"**. Does not make sense for sync or async-now functions.

Calls to async functions **with the same name** and marked as **"discardPreviouslyQueuedSimilarCalls"** that come in before previously queued such calls are really sent to the browser window will discard those previous calls completely (and keep only the latest).

So for example a requestFocus() function call you would only need to send once - after an event handler finished execution server-side. You can send that later and only for the last requestFocus() that executed. If you click on a button for example and the solution has very complex code executing there that ends up calling requestFocus() 100 times on various components from various forms, you actually only want the last requestFocus to execute after the button is clicked. Otherwise there would be a huge performance penalty. That is why all requestFocus calls from existing components are (or should be) marked as "discardPreviouslyQueuedSimilarCalls".

Just add **"discardPreviouslyQueuedSimilarCalls": true** (another optional parameter (that by default is false)) to the call definition. This flag was also known previously as "globalExclusive" (but that is now deprecated).

```
        "requestFocus": {
            "parameters": [ { "name": "mustExecuteOnFocusGainedMethod", "type": "boolean", "optional": true }],
            "delayUntilFormLoads": true,
            "discardPreviouslyQueuedSimilarCalls": true
        }
```

Allow server side calls when component or parent form is not visible

By default, server side calls coming from client are ignored if the component or the parent form is not visible (anymore). For example, a call to a server side function when switching to a new form, to do some cleanup, might get blocked. To still allow these calls, you should add **"allowaccess" : "visible"** to the function definition in the .spec file.

Server Side Scripting

A component or service can have a server-side part as well (optional); this logic is executed directly on the server; in the spec file this is configured like:

.spec file

```
"serverscript": "servoydefault/tabpanel/tabpanel_server.js",
```

See Server Side Scripting for more info about server side scripting.

Doc file

(From 2021.12 on)

A component or service can have a doc file for specifying documentation of the api and model properties. This is the same as the JSDoc in client side (.js) file, but is needed because NG2 doesn't have a js file that we can use in this way, requiring doc to be specified in another place.

The api doc/property doc can also be specified in the spec file itself using the doc property , however this is suitable only for small descriptions.

.spec file

```
"doc": "servoydefault/tabpanel/tabpanel_doc.js",
```

tabpanel_doc.js file

```
/**
 * some sample text
 * @example elements.%%elementName%%.yourName = 'myname'
 */
var yourName;

/**
 * This is a sample function from doc javascript file.
 */
function somemethod() {
}
```

Palette categories

Web Components are organized in component packages. The palette of the WYSIWYG editor shows components grouped by package name. To further group components from the same package, the property 'categoryName' can be used. 'categoryName' is a property that each component can specify in its spec file. The palette then displays components belonging to the same category grouped under the specified 'categoryName'.

Example:

Category Name

```
{
        "name": "packagename-componentname",
        "displayName": "String more descriptive name that is shown in the designer",
        "categoryName": "Advanced",
...
}
```

Example

As an example we could have a Tab Panel that has a definition like: (note: this example does not contain the entire Tab Panel spec)

tabpanel.spec

```
{
    "name": "servoydefault-tabpanel",
    "displayName": "Tab Panel",
    "version": 1,
    "icon": "servoydefault/tabpanel/tabs.gif",

    "definition": "servoydefault/tabpanel/tabpanel.js",
    "serverscript": "servoydefault/tabpanel/tabpanel_server.js",

    "libraries": [{ "name": "accordionpanel", "version": "1", "url": "servoydefault/tabpanel/accordionpanel.
css", "mimetype": "text/css" }],

    "model": {
        "background": "color",
        "borderType": { "type": "border", "stringformat": true },
        "enabled": { "type": "enabled", "blockingOn": false, "default": true },
        "fontType": { "type": "font", "stringformat": true },
        "foreground": "color",
        "horizontalAlignment": { "type": "int", "tags": { "scope": "design" }, "values": [{ "LEFT": 2 }, {
"CENTER": 0 }, { "RIGHT": 4 }], "default": -1 },
        "location": "point",
        "readOnly": "protected",
        "selectedTabColor": "color",
        "size": { "type": "dimension", "default": { "width": 300, "height": 300 } },
        "styleClass": { "type": "styleclass", "tags": { "scope": "design" }, "values": [] },
        "tabIndex": { "type": "object", "pushToserver": "shallow" },
        "tabOrientation": { "type": "int", "tags": { "scope": "design" }, "values": [{ "default": 0 }, { "TOP":
1 }, { "HIDE": -1 }] },
        "tabSeq": { "type": "tabseq", "tags": { "scope": "design" } },
        "tabs": { "type": "tab[]", "droppable": true },
        "transparent": "boolean",
        "visible": "visible"
    },

    "handlers": {
        "onTabChange": "function",
        "onTabSort": {
            "parameters": [{ "name": "columnindex", "type": "int" },
                { "name": "sortdirection", "type": "string" },
                { "name": "event", "type": "JSEvent", "optional": true }],
            "returns": "string"
    },

    "api": {
        "addTab": {
            "returns": "boolean",
            "parameters": [{ "name": "form/formname", "type": "object []" },
                { "name": "name", "type": "object", "optional": true },
```

```
                { "name": "tabText", "type": "object", "optional": true },
                { "name": "tooltip", "type": "object", "optional": true },
                { "name": "iconURL", "type": "object", "optional": true },
                { "name": "fg", "type": "object", "optional": true },
                { "name": "bg", "type": "object", "optional": true },
                { "name": "relatedfoundset/relationname", "type": "object", "optional": true },
                { "name": "index", "type": "object", "optional": true }]
        },
        "getMnemonicAt": {
            "returns": "string",
            "parameters": [{ "name": "i", "type": "int" }]
        },
        "getSelectedTabFormName": {
            "returns": "string"
        },
        "getTabRelationNameAt": {
            "returns": "string",
            "parameters": [{ "name": "i", "type": "int" }]
        },
        "getTabTextAt": {
            "returns": "string",
            "parameters": [{ "name": "i", "type": "int" }]
        }
        "isTabEnabledAt": {
            "returns": "boolean",
            "parameters": [{"name": "i", "type": "int"}]
        },
        "removeAllTabs": {
            "returns": "boolean"
        },
        "removeTabAt": {
            "returns": "boolean",
            "parameters": [{"name": "index", "type": "int"}]
        },
        "setTabEnabledAt": {
            "parameters": [{ "name": "i", "type": "int" }, { "name": "value", "type": "boolean" }]
        },
        "setTabTextAt": {
            "parameters": [{"name": "index","type": "int"},
                { "name": "text", "type": "string" }]
        },
        "requestFocus": {
            "parameters": [{ "name": "mustExecuteOnFocusGainedMethod", "type": "boolean", "optional": true }],
            "delayUntilFormLoad": true,
            "discardPreviouslyQueuedSimilarCalls": true
        }
    },

    "types": {
        "tab": {
            "name": "string",
            "containsFormId": "form",
            "text": "tagstring",
            "relationName": "relation",
            "active": "boolean",
            "foreground": "color",
            "disabled": "boolean",
            "imageMediaID": "media",
            "mnemonic": "string"
        }
    }
}
```