Server Side Scripting

Defining the server side scripting file in .spec

A component or service can have a serverside part, so that some of it's logic is executed on the server; in the spec file this is configured like:

.spec file

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

Server-side code is build up the same as the client-side so you have **$scope** object with **model/api/**(since 8.3) **handler/**(since 8.3.1) **servoyApi** objects.

An example is the default tabpanel.

API methods (private, public) and calling them in client-side js, server-side js and solution code

What API methods can solution code, server side js and client side js call

Server side scripting can contain API methods defined in the spec of the component / service. These methods execute then directly server-side. When a certain API is called from Servoy solution scripting, Servoy will first check if that API is defined server side. If it is, it will just call the server side API method.

If the API method is not defined in the server side file (or component doesn't have server side scripting at all) , Servoy will call that method from the client side js file of the service / component. Any such message or call that is send to client will send all the outstanding server-side model changes as well to the client - so that client will be in sync with server before the API method gets called.

As this server side file can also have the implementation of the public API that is defined in the .spec file, you can handle complex things server side; this can be used as a performance enhancement so that the api call doesn't have to go to the client to execute at the moment the call is made from Servoy solution code.

The API call that is executed server-side could for example just update some model properties of the component/service; these will be sent automatically as one thing to the client when the server-side code is done. And then client can detect those model changes. Or **server-side code could call directly a client side API method** after doing some pre-processing for example. More complex components like tabpanel, grouping girds based on foundsets or services like form popup service heavily rely on server-side implementation both for **public APIs (defined in "api" section in the .spec file and present in server-side scripting as $scope.api.xyz())** and for private implementations details from **internalAPIs (defined in "internalApi" section of the .spec file and present in server-side scripting as $scope.xyz() and in client side scripting in the usual api object)**.

On the **$scope.api object you can define the public api methods** that you want to execute server side. The difference - compared to client side structure is that "internalApi"s from .spec file are defined directly on $scope compared to client - where "internalApi"s are defined in the same place as public api methods.

Private and public API methods can be defined either in server side js code or in client side js code. Solution code can call public API methods from either of the two. Server-side js code can call client side js (and of course server side js) public and private API methods. Client side code js can call server side private API methods (and of course any client side API methods).

Calling client side API methods from server side js

Example of calling client side api from server side js

```
$scope.api.closePopup = function() // we define a server side public api method
{
    if ($scope.model.popupIsShown)
    {
        // only calls client side api (which generates network traffic) if it is needed
        $scope.api.doClosePopup(); // "doClosePopup" is defined either in "api" (public) or "internalApi"
(private) section of the spec file; and it's implemented in client side js
    }
}
```

Calling server side internal API methods from client side js

What can also be done (Servoy 8.0.2+) is that client-side scripting can call a server side component/service private/internal scripting method. This function has to be defined on the $scope object in server-side js. **Since Servoy 8.2** you must define these methods in a special api category of the .spec, ***internalApi*** . In prior versions define these in ***api*** section instead. As an example:

serverside script

```
$scope.loadRelatedFoundsetForNewTreeNode = function(parentRecord, relationIdx) {
    (...) // related foundset is prepared here
        return newRelatedFoundset;
}
```

.spec file

```
"internalApi": {
    "loadRelatedFoundsetForNewTreeNode": {
        "returns": "foundsetRef",
        "parameters": [
                { "name": "parentRecord", "type": "record" },
            { "name": "relationIdx", "type": "int" }
        ]
    }
}
```

Then a **component** can use the **svyServoyapi** to call this server side internal/private function from client side js:

client javascript

```
// assign to the scope the svy-servoyapi
scope: {
    model: '=svyModel',
    servoyApi: '=svyServoyapi'
},

        // in the controller or link functions you can use that then
    (...)
    $scope.servoyApi.callServerSideApi("loadRelatedFoundsetForNewTreeNode",[rows[i], 2]).then(function
(returnedFoundsetRef) {
        console.log("Related foundset successfully returned...");
    }, (...));
    (...)
```

callServerSideApi above needs to have the function name and an array of arguments; it returns a $q promise of angular where the then function will give you the return value of the callback.

In case of a service you can use the **$services** service to do the same thing, but you have to also give the name of the service itself (of course usually services would not work with foundsets and relations, but it is just an example):

clientside service script

```
$services.callServerSideApi("myservicename","loadRelatedFoundsetForNewTreeNode",[rows[i], 2]).then(function
(returnedFoundsetRef) {
   console.log("Related foundset successfully returned...");
 });
```

Model changes to server

Beware that callServerSideApi **does not send outstanding client side model changes to server**; this should be handled by sending new values as parameters and modifying model server-side.

Internal API can also be used for API that is defined on client, but can only be called from server side scripting. This API shouldn't be called from Servoy Developer scripting and won't show in code completion.

Calling handlers from server-side js

Since Servoy 8.3 server side scripting also has access to the **handlers** a developer has assigned in the designer - through **$scope.handlers**. Together with the new **"private": true** configuration on a handler definition **in the spec file**, you can make handlers that are not directly callable from the client but only through server side scripting.

```
$scope.myServerSideInternalAPIMethod = function(name, type) {
    // call a handlder directly with the arguments are return the value the handler gives
        return $scope.handlers.onAction(name, type);
}
```

Private handlers which are callable only from server-side js can be used to implement secure complex web components; for instance a Navigation Menu component which has a collection of MenuItems custom type. When the whole menu is disabled you can use the protecting properties (see Protecting Properties at Specification (.spec file)) to block certain handlers to be triggered; however when a single MenuItem is disabled and the navigation menu is enabled meanwhile, to securely prevent the handler to be executed for the disabled MenuItem (e.g. onMenuItemClick should be executed only if the menu item is enabled), you should call a server side script which verifies the MenuItem is actually enabled and triggers the private handler from the server side script.

For information about documenting API functions see the documenting api functions example. For information about documenting model properties and handlers please have a look at Documenting what properties do / Documenting handlers and handler templates.

Calling special servoyApi functions from server-side js

Starting with Servoy 8.3.1, inside the server side scripting file of a component/service you can call specific Servoy provided functions using the " ***servoyApi*** " object.

hideForm api is used to mark a form as not visible anymore immediately because cannot wait for client to mark it as hidden(for example in removeTab of a tabpanel).

servoy API example

```
servoyApi.hideForm('myform')
```

copyObject is used to create a new javascript object from existing one. This is a shortcut for creating the object from scratch and copying all properties.

servoy API example

```
var tabCopy = servoyApi.copyObject($scope.model.tabs[0])
```

getViewFoundSet is used to creates a view (read-only) foundset from a query builder select (QBSelect) (**available starting with Servoy 8.4.0**)

servoy API example

```
var query = parentFoundset.getQuery();
var myViewFoundset = servoyApi.getViewFoundSet("myViewFoundset", query);
```

getQuerySelect is used to get select query for a dataSource (**available starting with Servoy 2019.03**)

servoy API example

```
var query = servoyApi.getQuerySelect(foundset.getDataSource());
```

getMediaUrl is used to generate a url from a byte array so that the client can get the bytes from that url (**available starting with Servoy 2019.09**)

servoy API example

```
var query = servoyApi.getMediaUrl(bytes);
```

getDatasourcePKs is used to generate a list of primary keys names for the given data source (**available starting with Servoy 2021.06**)

servoy API example

```
var query = servoyApi.getDatasourcePKs(datasource);
```

Logging

Inside the server side scripting file of a component/service you can log messages using " ***console*** ", not application,output. The output will appear in developer's console view as well as in the application server log file (depending on configured logging levels). For example:

Logging in component/service server side scripting file

```
console.log(message);
console.warn(message);
console.error(message);
```

Hooking up into component lifecycle

Support has been added in Servoy 2022.06 for two new handler functions. These can be defined for hooking up into the component's client side destroy / create lifecycle.
These two methods will be called directly in server-side scripting when the component is shown/hidden on client. They are useful because - for example - some components want to do cleanup actions when they get hidden - but it is too late to call a server-side api when the component detects that it is destroyed on client, because server will normally block that call - the form is already known to be hidden (sometimes, in rare cases, at that point it may even be destroyed).

Any code the component wants to execute server-side for show/hide can be added to these two functions by:

- defining them in the server side scripting file of the component
  onShow: which will be called every time the component gets shown

  ```
  $scope.onShow = function() {
    // code to execute when the component is shown/created in the client.
  }
  ```

  onHide: which will be called every time the component gets hidden

  ```
  $scope.onHide = function() {
    // code to execute when the component is hidden/destroyed in the client.
  }
  ```

- defining them in the internalApi section of the component's .spec file:

```
"internalApi": {
    "onHide": {},
        "onShow": {}
}
```