


Annotating JavaScript using JSDoc

The Script Editor in Servoy Developer offers full code completion (a.k.a. IntelliSense or autocomplete) and designtime code validation.

As JavaScript has no means to declare the type of a variable, the type of a function parameter or the return type of a function, it is not possible to get 100% correct results by just analyzing the JavaScript code itself.

In order to improve the quality of the code completion and code validation, functions and variables can be annotated with JSDoc to provide the missing information.

Besides the benefits for code completion and validation, adding JSDoc to JavaScript code also improves the readability of the code for other developers, as JSDoc allows for adding more info than just the typing info.

 Using the [JSDoc plugin](#) hosted on [ServoyForge](#) it is also possible to generate HTML documentation of the JavaScript code in a Solution, based on the JSDoc supplied.

In This Chapter

- [What does JSDoc consist of?](#)
- [Where does JSDoc come from and which syntax is supported](#)
- [Working with JSDoc in the Script Editor](#)
- [JSDoc Tags](#)
- [Type Expressions](#)
- [Type Casting](#)

What does JSDoc consist of?

The JSDoc syntax consists of a set of JSDoc tags, contained in JSDoc comments.

JSDoc comments are like multi-line JavaScript comments, but the opening tag is `/**` **instead of just** `/'`

Some of the JSDoc tags require a Type Expression as one of the parameters and most allow for an extra description behind the tag and it's parameters.

Example

```
/**
 * A simple demo function that outputs some text
 * @author Tom
 * @private
 *
 * @param {String} text The text that will be written to the output
 * @throws (String)
 * returns Boolean
 *
 * @example try {
 *     saySomething('Hello world!');
 * } catch(e) {
 *
 * }
 *
 * @see application.output
 * @since 1.0
 * @version 1.0.1<br>
 * - Added some more JSDoc tags for the demo
 */
function saySomething(text) {
    if (text == null || text.length == 0) {
        throw "Invalid input!"
    }
    application.output(text);
    return true;
}
```

Where does JSDoc come from and which syntax is supported

JSDoc is not a official standard, but the defacto standard is is defined by the [JSDoc Toolkit](#) project. The other major definer of JSDoc is [Google Closure Compiler's support for JavaScript annotation](#).

The JSDoc syntax supported by the Servoy Developer IDE is derived from the [JSDoc Toolkit](#) and [Google Closure Compiler's support for JavaScript annotation](#), plus some custom Servoy extensions.

See [JSDoc Tags](#) and [Type Expressions](#) below for the supported tags and their syntax.


Working with JSDoc in the Script Editor

As mentioned in the intro, the Script Editor in Servoy Developer utilizes JSDoc to improve the quality of code completion and validation.

The Script Editor and Servoy Developer in general also facilitates the creation of JSDoc comments:

- When creating functions and variables through the wizards in the Solution Explorer or the Properties pane linked to the Form Editor, Servoy will automatically generate the variable or function with JSDoc comments.
- When manually creating variables and functions inside the Script Editor, using code completion it is possible to select Script Templates for new variables or functions that include the JSDoc comments
- When working with existing variables and functions, the Script Editor has a function to automatically generate the JSDoc comments for the selected variable or function. This function can be accessed through:
 - Alt-Shift-J
 - Context Menu > Source > Generate Element Comments
- Inside the JSDoc comment, the Script Editor offers code completion for the available JSDoc tags if the "@" sign is entered and then code completion is requested (Control-Space)

When hovering over a reference to the variable or function somewhere in the Solution, the tooltip will show the JSDoc for the variable/function.

 Note that the Script Editor will always generate a JSDoc comment block with a @properties tag when saving the Script editor, if no JSDoc comments have been defined. The @properties tag is a tag containing information for Servoy to provide proper linking and versioning.

JSDoc Tags

The following JSDoc tags are supported in the Script Editor. This means that the JSDoc tags will be rendered without the "@" sign when hovering over a reference to the function or variable.

The developer can add any custom tag to the JSDoc comment, but besides being shown in the tooltip when hovering over references it will not do anything.

Tag	Syntax & Examples	Context	Impact	Description
@AllowToRunInFind	@AllowToRunInFind	function	Determines if the function will be executed in FindMode when used as an event handler	Custom Servoy JSDoc tag to annotate a function that it can be run if the Form on which the function is run is in FindMode
@author	@author userName	function, variable	none	Indicates the author of the code
@constructor	@constructor	function	This will show a different icon on the Script Outline view. besides that no further impact	
@deprecated	@deprecated	function, variable	Accessing a deprecated variable or calling a deprecated function will produce a builder marker in Servoy Developer	Indicates that the function or variable is obsolete or has been replaced and should be used anymore.
@example	@example	function, variable	none	<p>Tag allowing to provide some sample code how to use the function or variable. Multiline content is possible by including " " as line-breaks behind each line of content.</p> <p>To have more control over the formatting of the sample code, the entire sample code can be wrapped in pre-tags:</p> <pre>samplecode</pre> <p>Multiple @example tags can be defined for each function or variable</p>
@param	@param {Type} name parameterDescription	function	Builder markers will be generated in Servoy Developer if the function is called with values for the parameters that do not	<p>Describe function parameters.</p> <p>The tag can be followed by a Type Expression between {} and must have a name.</p> <p>The "name" must match the name of one of the parameters in the function declaration.</p> <p>When the parameter is an unknown Java object (so not a JavaScript object) or there should be any type information assigned to the parameter, the type expressing can be omitted.</p>
@private	@private	function, variable	Accessing a private variable/function from outside the scope in which it is declared will generate a builder marker in Servoy Developer	Annotates a variable or function as accessible only from within the file in which it is declared
@protected	@protected	function, variable	Accessing a protected variable/function from outside the scope in which it is declared or a child scope will generate a builder marker in Servoy Developer	Annotates a variable or function as accessible from within the same file in which it is declared and all files that extend this file
@return	@return {Type}	function	The specified type is used by the build process to determine the correctness of the code that uses the returned value	<p>Annotates the type of the returned value.</p> <p>If the function does not return any value, omit the @return tag.</p> <p>The tag must be followed by a Type Expression</p>
@returns	@returns {Type}	function	see @return	alias for @return
@see	@see seeDescription	function, variable	none	Tag to provide pointers to other parts of the code that are related

@since	@since versionDescription	function, variable	none	Tag to provide information about in which version of the code the variable or function was introduced
@SuppressWarnings	@SuppressWarnings ([deprecated], [hides], [wrongparameters], [undeclared])	function	Stop the generation of builder markers in Servoy Developer for the specified warnings	Custom Servoy JSDoc tag to suppress builder markers of a certain type within a function.
@throws	@throws {Type}	function	none	Tag to describe the type of Exceptions that can be raised when the function is called. Multiple @throws tags are allowed. The tag must be followed by a Type Expression
@type	@type {Type}	variable, inline variable, (function*)	The specified type is used by the build process to determine the correctness of the code that uses the variable	Tag to specify the type of the value that a variable can hold. The tag must be followed by a Type Expression On functions the @type tag is an alternative for @returns, but only one of the two can be used
@version	@version versionDescription	function, variable	none	Tag to provide information about the version of the code

i A file can be either a Form JavaScript file or the globals JavaScript file. Only Form can be extended, thus the @protected tag is not relevant for annotating variables and functions within the globals JavaScript file

Type Expressions

Type Expressions are used to describe the type and/or structure of data in the following cases:

Use case	Tag	Example
function parameters	@param	/** * @param {String} value Just some string value */ function demo(value) {...}
function return type	@return @returns	/** * @param {String} value Just some string value * @return { {x:Number, y:Number} } */ function demo(value) { ... return {x: 10, y: 20} }
functions exceptions	@throws	/** * @throws {Number} */ function demo(value) { ... throw -1; }
variables	@type	/** * @type {XML} */ var html = Hello World!

A Type Expression is to always be surrounded by curly braces: {typeExpression}. Note that when using the Object Type expression variation that start and stops with curly braces as well, this results in double opening and closing braces.

Expression name	Syntax example	Context	Comments
Named type	{String} {Boolean} {Number} {XML} {XMLList} {RuntimeForm} {RuntimeLabel} {JSButton} {JSForm}	@param, @return, @type, @throws	The complete list of available types can be seen by triggering Code Completion inside the curly braces in the Script Editor
Any type	{*} Any type of value	@param, @return, @type, @throws	
OR type	{String Number} Either a String or a Number	@param, @return, @type, @throws	

REST type	<code>{...String}</code> Indicates one or more String values	<code>@param</code>	
Array type	<code>{String[]}</code> <code>{Array}</code> An array containing just string values <code>{Array}</code> An array containing just bytes	<code>@param</code> , <code>@return</code> , <code>@type</code> , <code>@throws</code>	
Object type	<code>{Object}</code> An object where the value for each key is a String value <code>{Object>}</code> An object where the value for each key contains arrays that in turn contains only string values <code>{ (name:String, age:Number)}</code> An object with a "name" and "age" key, with resp. a string and number value	<code>@param</code> , <code>@return</code> , <code>@type</code> , <code>@throws</code>	
Object type with optional properties	<code>{ (name:String, [age]:Number)}</code> <code>{ (name:String, age:Number=)}</code> An object with a "name" and optional "age" key, with resp. a string and number value	<code>@param</code> , <code>@return</code> , <code>@type</code> , <code>@throws</code>	
JSFoundset type	<code>{JSFoundset}</code> ¹ A JSFoundSet from the contacts table of the udm database server <code>{JSFoundset<{col1:String, col2:Number}>}</code> A JSFoundSet with dataproviders "col1" and "col2" with resp. string and number types	<code>@param</code> , <code>@return</code> , <code>@type</code>	
JSRecord type	<code>{JSRecord}</code> ¹ A JSRecord from the contacts table of the udm database server <code>{JSRecord<{col1:String, col2:Number}>}</code> A JSFoundSet with dataproviders "col1" and "col2" with resp. string and number types	<code>@param</code> , <code>@return</code> , <code>@type</code>	
JSDataSet type	<code>{JSDataSet<{name:String, age:Number}>}</code> An JSDataSet with a "name" and "age" column, with resp. a string and number value	<code>@param</code> , <code>@return</code> , <code>@type</code>	
RuntimeForm type	<code>{RuntimeForm}</code> A RuntimeForm that extends superFormName	<code>@param</code> , <code>@return</code> , <code>@type</code>	

¹ the value in between <...> is the datasource notation that is built up of the database server and tablename: db:{serverName}/{tableName}

Type Casting

JSDoc can be used inside JavaScript code to specify the type of variables. This can be necessary if the correct type can't be automatically derived.

An example of such scenario is for example the `databaseManager.getFoundSet()` function. This function returns an object of the generic type `JSFoundSet`. In most if not all scenario's however, it is known for which specific datasource the `JSFoundSet` was instantiated and the foundset object will be used as such in code, accessing dataproviders on the foundset object that are specific to the datasource. This will result in builder markers, because those dataproviders are not known on the generic `JSFoundSet` type. Through JSDoc casting however, it's possible to specify the type of the foundset object more specifically

```
/**@type {JSFoundset<db:/udm/contacts>}*/
var fs = databaseManager.getFoundSet('db:/udm/contacts')
```

The difference between Code Completion with and without Type Casting can be seen in the two screenshots below. When the Type casting is omitted, the offered Code Completion related only to the generic `JSFoundset` type. With the Type Casting in place, all the dataproviders of the specific datasource are also available in Code Completion:

With Type Casting:



```

1  // TODO: Add your implementation details here.
2
3  // Example: Add a new contact
4  // 1. Create a new Contact object
5  // 2. Add it to the database
6  // 3. Save the database
7
8  // Example: Delete a contact
9  // 1. Find the contact by name or phone number
10 // 2. Delete it from the database
11 // 3. Save the database
12
13 // Example: Update a contact
14 // 1. Find the contact by name or phone number
15 // 2. Update its details
16 // 3. Save the database
17
18 // Example: Get all contacts
19 // 1. Retrieve the database
20 // 2. Return the list of contacts
21
22 // Example: Get contact by name
23 // 1. Retrieve the database
24 // 2. Find the contact by name
25 // 3. Return the contact
26
27 // Example: Get contact by phone number
28 // 1. Retrieve the database
29 // 2. Find the contact by phone number
30 // 3. Return the contact
31
32 // Example: Save the database
33 // 1. Serialize the database to a file
34 // 2. Return the file path
35
36 // Example: Load the database
37 // 1. Deserialize the database from a file
38 // 2. Return the database
39
40 // Example: Initialize the database
41 // 1. Create a new database
42 // 2. Add some default contacts
43 // 3. Save the database
44
45 // Example: Test the database
46 // 1. Create a new database
47 // 2. Add some default contacts
48 // 3. Save the database
49 // 4. Load the database
50 // 5. Print the contacts
51
52 // Example: Main method
53 // 1. Create a new database
54 // 2. Add some default contacts
55 // 3. Save the database
56 // 4. Load the database
57 // 5. Print the contacts
58
59 // Example: Main method
60 // 1. Create a new database
61 // 2. Add some default contacts
62 // 3. Save the database
63 // 4. Load the database
64 // 5. Print the contacts
65
66 // Example: Main method
67 // 1. Create a new database
68 // 2. Add some default contacts
69 // 3. Save the database
70 // 4. Load the database
71 // 5. Print the contacts
72
73 // Example: Main method
74 // 1. Create a new database
75 // 2. Add some default contacts
76 // 3. Save the database
77 // 4. Load the database
78 // 5. Print the contacts
79
80 // Example: Main method
81 // 1. Create a new database
82 // 2. Add some default contacts
83 // 3. Save the database
84 // 4. Load the database
85 // 5. Print the contacts
86
87 // Example: Main method
88 // 1. Create a new database
89 // 2. Add some default contacts
90 // 3. Save the database
91 // 4. Load the database
92 // 5. Print the contacts
93
94 // Example: Main method
95 // 1. Create a new database
96 // 2. Add some default contacts
97 // 3. Save the database
98 // 4. Load the database
99 // 5. Print the contacts
100
101 // Example: Main method
102 // 1. Create a new database
103 // 2. Add some default contacts
104 // 3. Save the database
105 // 4. Load the database
106 // 5. Print the contacts
107
108 // Example: Main method
109 // 1. Create a new database
110 // 2. Add some default contacts
111 // 3. Save the database
112 // 4. Load the database
113 // 5. Print the contacts
114
115 // Example: Main method
116 // 1. Create a new database
117 // 2. Add some default contacts
118 // 3. Save the database
119 // 4. Load the database
120 // 5. Print the contacts
121
122 // Example: Main method
123 // 1. Create a new database
124 // 2. Add some default contacts
125 // 3. Save the database
126 // 4. Load the database
127 // 5. Print the contacts
128
129 // Example: Main method
130 // 1. Create a new database
131 // 2. Add some default contacts
132 // 3. Save the database
133 // 4. Load the database
134 // 5. Print the contacts
135
136 // Example: Main method
137 // 1. Create a new database
138 // 2. Add some default contacts
139 // 3. Save the database
140 // 4. Load the database
141 // 5. Print the contacts
142
143 // Example: Main method
144 // 1. Create a new database
145 // 2. Add some default contacts
146 // 3. Save the database
147 // 4. Load the database
148 // 5. Print the contacts
149
150 // Example: Main method
151 // 1. Create a new database
152 // 2. Add some default contacts
153 // 3. Save the database
154 // 4. Load the database
155 // 5. Print the contacts
156
157 // Example: Main method
158 // 1. Create a new database
159 // 2. Add some default contacts
160 // 3. Save the database
161 // 4. Load the database
162 // 5. Print the contacts
163
164 // Example: Main method
165 // 1. Create a new database
166 // 2. Add some default contacts
167 // 3. Save the database
168 // 4. Load the database
169 // 5. Print the contacts
170
171 // Example: Main method
172 // 1. Create a new database
173 // 2. Add some default contacts
174 // 3. Save the database
175 // 4. Load the database
176 // 5. Print the contacts
177
178 // Example: Main method
179 // 1. Create a new database
180 // 2. Add some default contacts
181 // 3. Save the database
182 // 4. Load the database
183 // 5. Print the contacts
184
185 // Example: Main method
186 // 1. Create a new database
187 // 2. Add some default contacts
188 // 3. Save the database
189 // 4. Load the database
190 // 5. Print the contacts
191
192 // Example: Main method
193 // 1. Create a new database
194 // 2. Add some default contacts
195 // 3. Save the database
196 // 4. Load the database
197 // 5. Print the contacts
198
199 // Example: Main method
200 // 1. Create a new database
201 // 2. Add some default contacts
202 // 3. Save the database
203 // 4. Load the database
204 // 5. Print the contacts
205
206 // Example: Main method
207 // 1. Create a new database
208 // 2. Add some default contacts
209 // 3. Save the database
210 // 4. Load the database
211 // 5. Print the contacts
212
213 // Example: Main method
214 // 1. Create a new database
215 // 2. Add some default contacts
216 // 3. Save the database
217 // 4. Load the database
218 // 5. Print the contacts
219
220 // Example: Main method
221 // 1. Create a new database
222 // 2. Add some default contacts
223 // 3. Save the database
224 // 4. Load the database
225 // 5. Print the contacts
226
227 // Example: Main method
228 // 1. Create a new database
229 // 2. Add some default contacts
230 // 3. Save the database
231 // 4. Load the database
232 // 5. Print the contacts
233
234 // Example: Main method
235 // 1. Create a new database
236 // 2. Add some default contacts
237 // 3. Save the database
238 // 4. Load the database
239 // 5. Print the contacts
240
241 // Example: Main method
242 // 1. Create a new database
243 // 2. Add some default contacts
244 // 3. Save the database
245 // 4. Load the database
246 // 5. Print the contacts
247
248 // Example: Main method
249 // 1. Create a new database
250 // 2. Add some default contacts
251 // 3. Save the database
252 // 4. Load the database
253 // 5. Print the contacts
254
255 // Example: Main method
256 // 1. Create a new database
257 // 2. Add some default contacts
258 // 3. Save the database
259 // 4. Load the database
260 // 5. Print the contacts
261
262 // Example: Main method
263 // 1. Create a new database
264 // 2. Add some default contacts
265 // 3. Save the database
266 // 4. Load the database
267 // 5. Print the contacts
268
269 // Example: Main method
270 // 1. Create a new database
271 // 2. Add some default contacts
272 // 3. Save the database
273 // 4. Load the database
274 // 5. Print the contacts
275
276 // Example: Main method
277 // 1. Create a new database
278 // 2. Add some default contacts
279 // 3. Save the database
280 // 4. Load the database
281 // 5. Print the contacts
282
283 // Example: Main method
284 // 1. Create a new database
285 // 2. Add some default contacts
286 // 3. Save the database
287 // 4. Load the database
288 // 5. Print the contacts
289
290 // Example: Main method
291 // 1. Create a new database
292 // 2. Add some default contacts
293 // 3. Save the database
294 // 4. Load the database
295 // 5. Print the contacts
296
297 // Example: Main method
298 // 1. Create a new database
299 // 2. Add some default contacts
300 // 3. Save the database
301 // 4. Load the database
302 // 5. Print the contacts
303
304 // Example: Main method
305 // 1. Create a new database
306 // 2. Add some default contacts
307 // 3. Save the database
308 // 4. Load the database
309 // 5. Print the contacts
310
311 // Example: Main method
312 // 1. Create a new database
313 // 2. Add some default contacts
314 // 3. Save the database
315 // 4. Load the database
316 // 5. Print the contacts
317
318 // Example: Main method
319 // 1. Create a new database
320 // 2. Add some default contacts
321 // 3. Save the database
322 // 4. Load the database
323 // 5. Print the contacts
324
325 // Example: Main method
326 // 1. Create a new database
327 // 2. Add some default contacts
328 // 3. Save the database
329 // 4. Load the database
330 // 5. Print the contacts
331
332 // Example: Main method
333 // 1. Create a new database
334 // 2. Add some default contacts
335 // 3. Save the database
336 // 4. Load the database
337 // 5. Print the contacts
338
339 // Example: Main method
340 // 1. Create a new database
341 // 2. Add some default contacts
342 // 3. Save the database
343 // 4. Load the database
344 // 5. Print the contacts
345
346 // Example: Main method
347 // 1. Create a new database
348 // 2. Add some default contacts
349 // 3. Save the database
350 // 4. Load the database
351 // 5. Print the contacts
352
353 // Example: Main method
354 // 1. Create a new database
355 // 2. Add some default contacts
356 // 3. Save the database
357 // 4. Load the database
358 // 5. Print the contacts
359
360 // Example: Main method
361 // 1. Create a new database
362 // 2. Add some default contacts
363 // 3. Save the database
364 // 4. Load the database
365 // 5. Print the contacts
366
367 // Example: Main method
368 // 1. Create a new database
369 // 2. Add some default contacts
370 // 3. Save the database
371 // 4. Load the database
372 // 5. Print the contacts
373
374 // Example: Main method
375 // 1. Create a new database
376 // 2. Add some default contacts
377 // 3. Save the database
378 // 4. Load the database
379 // 5. Print the contacts
380
381 // Example: Main method
382 // 1. Create a new database
383 // 2. Add some default contacts
384 // 3. Save the database
385 // 4. Load the database
386 // 5. Print the contacts
387
388 // Example: Main method
389 // 1. Create a new database
390 // 2. Add some default contacts
391 // 3. Save the database
392 // 4. Load the database
393 // 5. Print the contacts
394
395 // Example: Main method
396 // 1. Create a new database
397 // 2. Add some default contacts
398 // 3. Save the database
399 // 4. Load the database
400 // 5. Print the contacts
401
402 // Example: Main method
403 // 1. Create a new database
404 // 2. Add some default contacts
405 // 3. Save the database
406 // 4. Load the database
407 // 5. Print the contacts
408
409 // Example: Main method
410 // 1. Create a new database
411 // 2. Add some default contacts
412 // 3. Save the database

```

```
/**@type {Array<String>}*/  
var myStringArray = \[\]
```

An example of a generic type would be `RuntimeComponent`, which is the super type for `RuntimeLabel`, `RuntimeField` etc. `RuntimeComponent` defines all the properties and methods that all the other `RuntimeXxx` types have in common. When the need arises to call methods or set properties that are specific to a specific `RuntimeXxx` type, the generic type can be casted:

```
if (elements\[1\] instanceof RuntimeLabel) {
    /**@type{RuntimeLabel}*/
    var myLabel = elements\[1\]
    var elementNames = myLabel.getLabelForElementName() //Calling method specific for labels
}
```

Type Casting can only be performed on variable declarations. It is not possible switch the type of an already declared variable later in code