


# Annotating JavaScript using JSDoc

The Script Editor in Servoy Developer offers full code completion (a.k.a. IntelliSense or autocomplete) and design-time 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 its 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 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	none	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
@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	Describe function parameters. The tag can be followed by a <a href="#">Type Expression</a> between {} and must have a name. The "name" must match the name of one of the parameters in the function declaration. 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 expression can be omitted.
@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	Annotates the type of the returned value. If the function does not return any value, omit the @return tag. The tag must be followed by a <a href="#">Type Expression</a>
@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 <a href="#">Type Expression</a>
@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 <a href="#">Type Expression</a> 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
AnyType	* Any type of value	@param, @return, @type, @throws	
OR Type	{String Number} Either a String or a Number	@param, @return, @type, @throws	
REST Type	{...String} Indicates one or more String values	@param	

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

<sup>1</sup> 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

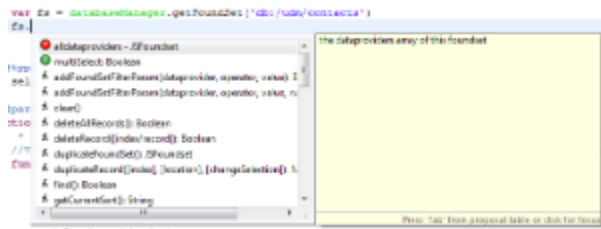
```
/**@type {JSFoundset<db:/udm/contacts>}*/
var fs = databaseManager.getFoundSet('db:/udm/contacts')
```

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:



Without Type Casting:



Another example is entries in Objects and/or Arrays: if every entry is of the same type, this can be specified on the Object/Array declaration using JSDoc, for example:

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

If the Object/Array contains entries of different types, the type of the entries cannot be specified when declaring the Object/Array, or only a more generic type can be specified.

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