JSDoc Annotations

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

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 of 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 tot he 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 executes 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 ran 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 and suppresses warnings related to inconsistent return values when building in a fail-save to calling a constructor function without the new keyword	
@deprecated	@deprecated description	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. Use an (optional) description to provide reasoning and possible alternatives

@enum	@enum
	
	Example:
	`/**`
	` * @enum`
	` */`
	`var TEAM_COLORS = {`
	`  RED : 1,`
	`  GREEN : 2,`
	`  BLUE : 3`
	`}`
		variable	none in scripting, but Servoy Relations	Variables that contain a JavaScript Object with key/value pairs and are tagged as enumeration using the @enum tag have special meaning in Servoy relations: The relation editor allows selecting one of the keys of the object for the primary 'key' in a relation item.
				
				Only String and Number values are supported. They are treated as constants, meaning that changes to the values made through scripting are not supported: if the value is altered, already loaded relations will not be updated accordingly.

@example	@example	function, variable	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. To have more control over the formatting of the sample code, the entire sample code can be wrapped in pre-tags: <pre>samplecode</pre> Multiple @example tags can be defined for each function or variable
@inheritDoc	@inheritDoc	function	Inherit JS documentation from the super form's function	On mouse hover or autocomplete, the documentation of the super function is displayed. This can go through multiple levels, as long as this tag is present.
@override	@override	functions	none	Tag to describe that the function is overriding an equally named function on a super form

@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 match the specified types	Describe function parameters. The tag can be followed by a Type Expression 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 expressing can be omitted.
@parse	@parse	functions	Depending on how many functions have this tag, it might result in an increase of the workspace build time.	The @parse tag can be used to force parsing of those functions that normally are not parsed when the "Shallow parsing" preference is set.
@public	@public	function, variable	Explicitly marks a member as public API.	A member that is not marked as either public, private or protected is implicitly considered as public. Cannot be used in combination with @private or @protected
@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 Cannot be used in combination with @public or @protected
@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 Cannot be used in combination with @public or @private
@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 and offer Code Completion	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 Type Expression
@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], [unused], [nls])	function	Stop the generation of builder markers in Servoy Developer for the specified warnings • deprecated - markers related to referencing deprecated API • hides - markers related to declaring members that hide similar named members elsewhere in the scope • wrongparameters - markers related to calling functions with different type of parameters than specified by the function • undeclared - markers related to referencing members for which the build system cannot find a declaration • unused - marker related to a member that is not being used • nls - markers related to hardcoded text when builder markers for non externalized strings are enabled (disabled by default, see window > Preferences > JavaScripts > Errors/Warnings > Externalized Strings)	Custom Servoy JSDoc tag to suppress builder markers of a certain type within a function
@this	@this {Type}	function	The specified type is used by the build process for the "this" object available inside functions to determine the correctness of the code that uses the object and offer Code Completion	Tag to specify the type of the "this" object inside functions. The tag must be followed by a Type Expression
@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 and offer Code Completion	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
@typedef	@typedef {Type}	variables	Variables tagged using the @typedef JSDoc tag are considered definitions of types. These types can be used as type in other JSDoc tags by using the name of the variable	
@version	@version versionDescription	function, variable	none	Tag to provide information about the version of the code

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 = <html> <head> </head> <body> Hello World! </body> </html>

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	This can be used to suppress some builder markers related to apparent type inconsistencies.
OR type	{String|Number} Either a String or a Number	@param, @return, @type, @throws	
REST type	{...String} Indicates one or more String values	@param	Can only be used for the last declared parameter of a function
Array type	{String[]} {Array<String>} An array containing just string values {Array<String|Number>} An array containing string and/or number values {Array<Byte>} An array containing just bytes	@param, @return, @type, @throws	

Object type	{Object<String>} An object where the value for each key is a String value {Object<Array<String>>} An object where the value for each key contains arrays that in turn contains only string values { {name:String, age:Number}} An object with a "name" and "age" key, with resp. a string and number value	@param, @return, @type, @throws	
Object type with optional properties	{ {name:String, [age]:Number}} { {name:String, age:Number=}} An object with a "name" and optional "age" key, with resp. a string and number value	@param, @return, @type, @throws	
Function type	{function(String, Number, Array<Object>):Boolean} Between the bracket of the function the types of the function parameters can be sprecified. The functions returnType can be specified after the closing bracket, prefixed by a colon	@param, @return, @type	
JSFoundset type	{JSFoundset<db:/udm/contacts>}[1] A JSFoundSet from the contacts table of the udm database server {JSFoundset<{col1:String, col2:Number}>} A JSFoundSet with dataproviders "col1" and "col2" with resp. string and number types	@param, @return, @type	
JSRecord type	{JSRecord<db:/udm/contacts>}[1] A JSRecord from the contacts table of the udm database server {JSRecord<{col1:String, col2:Number}>} A JSFoundSet with dataproviders "col1" and "col2" with resp. string and number types	@param, @return, @type	
JSDataSet type	{JSDataSet<{name:String, age:Number}>} An JSDataSet with a "name" and "age" column, with resp. a string and number value	@param, @return, @type	
RuntimeForm type	{RuntimeForm<superFormName>} A RuntimeForm that extends superFormName	@param, @return, @type	
RuntimeWebComponent type	{RuntimeWebComponent<webComponentName>}	@param, @return, @type	The webComponentName can be found in the associated spec file; webComponentName may be for example: bootstrapcomponents-tabpanel (leading to the following type: {RuntimeWebComponent<bootstrapcomponents-tabpanel>})
CustomType	{CustomType<componentName.customTypeName>}	@param, @return, @type	componentName may be web component or web service. The customTypeName is defined in the spec file. Example: {CustomType<bootstrapcomponents-tabpanel.tab>}

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

Type Casting

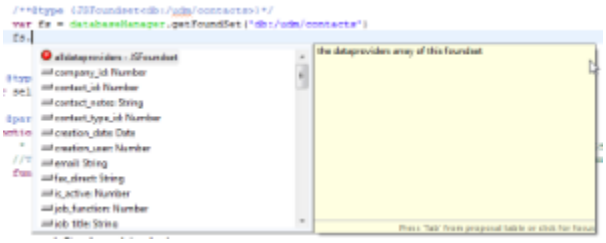
JSDoc can be used inside JavaScript code to specify the type of variables. This can be necessary if the correct type can't be automatically derived.

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 know on the generic JSFoundSet type. Through JSDoc casting however, it's possible to specify the type of the foundset object more specifically

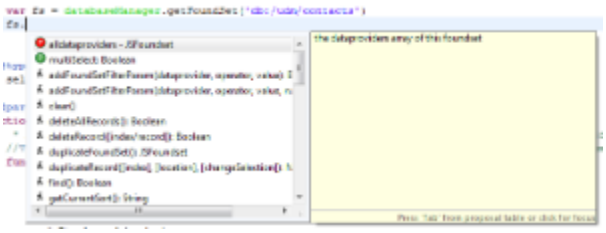
```
/**@type {JSFoundset<db:/udm/contacts>}*/
var fs = databaseManager.getFoundSet('db:/udm/contacts')
```

The difference between Code Completion with and without Type Casting can be seen in the two screenshots below. whent he 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 RuntimeXxxx 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

Shallow parsing property in the Servoy preferences

In the Preferences->ServoyJavascript Validation we have an option called shallow parsing. That property is enabled by default and if checked will make sure that when parsing over files the other files are not fully parsed but only functions that are marked with @parse or @constructor are parsed internally.

This greatly enhances performance because Servoy doesn't have to go over multiply files (deeply nested) to get the the type info of a object that is created in another file.

Problem is that you really need to document all the functions correctly that are not fully parsed. Because we can't extract on those the actual runtime return type from the function body. So @return is important to have for all the functions describing its object that it returns (together with the @param)

So to force Servoy to do parse the function because it is a constructor function like:

```
/**
 * @constructor
 */
function MyObject () {}
```

If you want to have a normal function (that for example does some prototyping) to be parsed you can use the @parse annotation:

```
/**
 * @parse
 */
function myInit() {
  MyObject.prototype = xxx
}
```

If those init methods are auto initializing variables like this:

```
var init = (
                /**
                 * @parse
                 */
                function(){
        SomeClass.prototype = Object.create(SomeClass.prototype);
        SomeClass.prototype.constructor = SomeClass;

        /**
         * @constructor
         */
        SomeClass.prototype.anotherFunction = function() {

        }
})()
```

make sure the @parse doc is on the function itself, not on the doc of the var init!