BrightScriptDoc

The BrightScript Language Documentation Generator

Overview

Basic Commenting

Like many other languages, BrightScript source code can be documented in-line using special character sequences to indicate a comment. For BrightScript, a basic comment starts with either the string "REM" or the single quote "'" character.

Beyond basic commenting – The BrightScriptDoc processor and BrightScriptDoc Tags

You can optionally choose to enhance your BrightScript code commenting with a special set of tags within those comment lines that, when processed by the BrightScriptDoc processor, will translate into formatted documentation that follows a pre-defined structure and presentation style.

The BrightScriptDoc processor is embedded in the BrightScript Eclipse plugin and is used automatically by the plugin to process code element documentation for use in the IDE's hover-docs and auto-completion features.

The BrightScriptDoc tags are a short-hand that translate into generated HTML on the fly.

BrightScriptDoc tags look and behave very similar to other language documentation tagging systems, (such as JavaDoc, PHPDoc, or AsDoc), but are specific to the needs of the BrightScript language.

Depending on the specific tag type, a tag instance may include both required and/or optional additional info after the tagname. Typically the last chunk of text after the tagname is the descriptive text argument used in the display of that specific tag's processed output.

BrightScriptDoc tags fall into two broad categories: Block tags or Inline tags.

Block Tags

Block tags are in the general format: @tagname <tag specific arguments>

Block tags must be the only thing on the comment line (nothing but REM or single quote and whitespace can come before it on the line).

Block tags types are categorized as either singletons or mult-instance,

Singleton block tag instances may only appear once per code element. If more than one instance of a singleton tag is used for a given code element, the first instance is used and all subsequent instances are logged as errors and ignored. Examples tag types include the @deprecated and @return tags.

Multi-instance block tags may be either named or unnamed , and may appear 0 or more times per code element. Multi-instance block tag's processed output is grouped together in a single output section. Example tag types include the @see and @param tags.

The @deprecated tag

The @deprecated tag is a singleton block tag.
@deprecated is used to indicate that the associated code element should be considered deprecated.
@deprecated takes a single optional argument (i.e. everything after the tag itself) - the descriptive text, to be displayed in the code element's documentation section marked “Deprecated:”. The “Deprecated” documentation section is special in that it is the only block tag/section to display **before** the code element's main description section in the HTML output.

The @param tag

The @param tag is a named, multi-instance block tag.
@param is used to describe one of a function/sub's parameter values.

@param tags have 2 arguments – 1 required and 1 optional.
The first sequence of characters not including a space (i.e. the first word) after the tag itself is considered the name of the parameter, and everything after that is considered the descriptive text argument, to be displayed in the code element's documentation section marked “Parameters:”.

The @return tag

The @return tag is a singleton block tag.
The associated code element must be a function (including anonymous functions) or sub.
@return is used to describe the function/sub's return value.
@return takes a single required descriptive text argument, to be displayed in the code element's documentation section marked “Returns:”.

The @since tag

The @since tag is a singleton block tag.
@since is used to indicate when the code element first came into existence.
@since takes a single descriptive text argument, to be displayed in the code element's documentation section marked “Since:”. Typically the descriptive text should center on datetime and/or version information.

The @see tag

The @see tag is an unnamed mult-instance block tag.
@see is used to reference other items that are related to the associated code element.

The @see tag comes in 3 different possible forms:

@see "text"

In this form, the @see tag simply outputs the quoted text as is (without the quotes).

@see label

In this form, the @see tag outputs the entire referenced link, as-is.

@see type.method#field label

In this form, the @see tag outputs a link that points to another code element.
When the BrightScriptDoc processor is running in the context of the BrightScript Eclipse plugin, this form of @see tag generates urls in the BrightScriptDocs special url protocol format (see below for a description of the BrightScriptDocs special url protocol format). The label argument, if missing, is automatically mirrors the unprocessed link value (i.e. without the "bsdocs:" url prefix).

Inline Tags

Inline tags are in the format: {@tagname <tag specific arguments>}

Inline tags can be used anywhere display text can be used, including in the descriptive text arguments of block tags and the main comment section. Example tag types include the {@link}, {@literal}, and {@code} tags.

The {@link} tag

The @link tag is an inline tag.
@link is an inline version of the 3rd form of the @see tag (i.e. the type.method#field form). Note that the <a href... form of @see is not needed as a separate inline tag, since most HTML markup is permitted inline in text as-is.

The {@literal} tag

The @literal tag is an inline tag.
@literal is used to display the first and only tag argument by applying HTML4 escape sequences to it. The resulting escaped text is wrapped in an HTML span tag.

The {@code} tag

The @code tag is an inline tag.
@code is very similar to the @literal tag is that it is also used to display the first and only tag argument by applying HTML4 escape sequences to it.
However, the resulting escaped text is wrapped in an HTML code tag instead of a span tag.

Placement of comments

BrightScriptDoc tags can be used in comments associated with specific functions, subs or variables. For functions and subs, the BrightScriptDoc processor looks for the contiguous group of comment lines immediately above the function/sub declaration. For variables, associated comments may be either grouped immediately above the variable's first assignment, or on the same line as the variable's first assignment. If both same-line and above comments exist, the comment line group above is used as the code element's associated documentation.

General format of a BrightScriptDoc tag-enhanced comment

The general format of a comment that is processed by the BrightScriptDoc processor should begin with a paragraph describing the code element (function/sub or variable), followed by an empty comment

line and then 0 or more BrightScriptDoc block tags, each on their own line. No non-comment lines should be present in this line grouping – if they are, the processor may not include the entire set of lines. The description paragraph text may contain a mix of HTML and BrightScriptDoc inline tags. In some cases, a given block tag may contain argument text used for presentation - that text argument itself may also contain inline tags. Note that use of HTML entities and escaping in presentation text may be required in order to get the desired result in some cases.

General format and sequence of the processed output

As a general rule for processing, all main and tag argument "descriptive text" is scanned for inline tags, which are converted to their respective HTML ouput "in-place".

All instances of multi-instance block tags are grouped together in the processed output under a single paragraph with a title indicative of the specific tag type.

The output of a processed tag-enhanced comment in general is in HTML and follows this specific sequence order and format:

> The singleton @deprecated tag, if it was used, is translated into a formatted paragraph titled "Deprecated:", with the processed descriptive comment argument as the text of the paragraph.
>
> The main descriptive text, processed for inline tags, is displayed without a paragraph title.
>
> All of the multi-instance @param tags, if they were used, are translated into a formatted paragraph titled "Parameters:". See the @param tag description in the Tags section below for info on the output format of these tags.
>
> The singleton @return tag, if it was used, is translated into a formatted paragraph titled "Returns:", with the processed descriptive comment argument as the text of the paragraph.
>
> The singleton @since tag, if it was used, is translated into a formatted paragraph titled "Since:", with the processed descriptive comment argument as the text of the paragraph.
>
> All of the multi-instance @see tags, if they were used, are translated into a formatted paragraph titled "See Also:". See the @see tag description in the Tags section below for info on the output format of these tags.

BrightScriptDocs special URL protocol format

The BrightScriptDocs special url protocol format allows the plugin to automatically handle references to BrightScript code elements defined by the user-developer in their channel code, the BrightScript global functions, and the BrightScript built-in collection of components, interfaces, and events.

The form of a bsdocs protocol url is: bsdocs://type.method#field

Where:

The type portion is optional, and refers to a BrightScript type or the special case type "global". BrightScript types are the names of the built-in BrightScript components, interfaces, and events. The special case "global" type refers to the BrightScript language global functions. Functions/subs and variables defined in the developer's BrightScript channel code can be referenced by omitting the type portion of the url and starting with the .method portion (with the #field portion only being required if the code element is a variable within the function/sub).

The method portion is optional, refers to a function or sub, and is always prefixed with a "." (dot character).

The field portion is optional, refers to variables within the specified function or sub, and is always prefixed with a “#” character.

Example

```
' Gets a false value in a convoluted way
' @deprecated Replaced by {@link .Foo2 Foo2} as of version 11.42
Function Foo() as Boolean
    x = false
    y = x
    return y
End Function

' Replaces the old Foo function with a more efficient algorithm
' @since version 11.42
Function Foo2() as Boolean
    return false
End Function
'
' Embodies the creation and running of a single custom screen for XYZ-hosted content
'
' @param utils an instance of the AA utils object created from calling the {@link
.CreateUtils Utils Contructor}
' @param site the site url string
' @param titlePrefix a static prefix for the video title display line
' @param contentAAArray an Array of content meta-data Aas
' @return False if there was an error during creating or running this screen.
' @see <a href="http://sdkdocs.roku.com/display/sdkdoc/Content+Meta-Data">Content Meta-
data</a>
' @see .CreateUtils CreateUtils()
Function ShowCustomScreen(utils as Object, site as String, titlePrefix as String,
contentAAArray as Object) as Boolean

    ' This is the z variable.  It does nothing useful...
    ' except as the target of a &#x0040;see or &#x0040;&#x007B;link&#x007D;
BrightScriptDoc tag
    z = "something"

    ' This is an example of top comments winning out over same-line.
    ' BrightScriptDocs prefers above comments over same-line comments.
    ' This is the set of lines that'll be used in hover-docs and auto-completion.
    x = 1 ' This comment won't be used by BrightScriptDoc

    y = {
        ' Anonymous functions comments are best placed above the field assignment
        ' So that all the relevent tags can be used
        '
        ' @param bar an integer that's apparently useful in calculating foo
        ' @param baz who knows what kind of object this is
        ' @return A string of some sort
        getFoo : Function(bar as Integer, baz as Object) as String
        End Function
    }
```

```
    z = CreateObject("roFoo")

End Function

' Creates an {@link roAssociativeArray associative array} of useful general utility
functions.
' {@literal <bold> and </bold> are presented as is in this literal tag, rather than
causing "and" to be bolded}
' {@code of course, the same thing is true in this code tag, but the font face is
different - <bold> and the matching </bold> tags are shown rather than cause the "and the
matching" to be bolded}
' <br/>
' And any old HTML you want to throw in is also valid, although equivalent BrightScriptDoc
tags are more efficient...
' <table border=1>
' <th>col 1</th><th>col 2</th><th>col 3</th>
' <tr><td>row 1 col 1</td><td>row 1 col 2</td><td>row 1 col 3</td></tr>
' </table>
' And so on...
' @see .ShowCustomScreen#z The z variable in the ShowCustomScreen method is rather
uninteresting
' @return an AA containing useful utility functions
Function CreateUtils() as Object
    aa = CreateObject("roAssociativeArray")

    aa.showcustom = ShowCustomScreen
    aa.foo = foo2

    return aa
End Function
```