NAME

monodocer – ECMA Documentation Format Support

SYNOPSIS

monodocer [OPTIONS]*

OPTIONS

–assembly:ASSEMBLY

ASSEMBLY is a .NET assembly to generate documentation stubs for.

Specify a file path or the name of a GAC'd assembly.

–delete Allow monodocer to delete members from documentation files. The only members deleted are for members which are no longer present within the assembly.

If a type is no longer present, the documentation file is *not* deleted, but is instead *renamed* to have a **.remove** extension.

–?, –help

Show program argument information.

–ignoremembers

Do not update members.

This will add documentation stubs for added types, but will *not* add or remove documentation for any members of any type (including any added types).

–importslashdoc:FILE

FILE is an XML file generated with the **/doc:FILE** C# compiler flag (e.g. *mcs -doc:foo.xml foo.cs*). Import the member documentation contained within *FILE* into the documentation format used by monodoc.

–name:NAME

NAME is the name of the project this documentation is for.

This sets the */Overview/Title* element within the *index.xml* file created at the directory specified by *-path* . This is used by some programs for title information (e.g. *monodocs2html*).

–namespace:NAMESPACE

Only update the types within the namespace *NAMESPACE* .

–overrides

Include overridden methods in documentation.

This normally isn't necessary, as the Mono Documentation Browser will provide a link to the base type members anyway, as will *monodocs2html* if the base type is within the same assembly.

–path:OUTPUT_DIR

OUTPUT_DIR is the directory which will contain the new/updated documentation stubs.

–pretty Indent the XML files nicely.

–since:SINCE

Create a <since/> element for added types and members with the value *SINCE* .

For example, when given *-since:"Gtk# 2.4"* an element will be inserted into the *Docs* element for all added types and type members:

<since version="Gtk# 2.4" />

The Mono Documentation Browser and *monodocs2html* will use this element to specify in which version a member was added.

–type:TYPE

Only create/update documentation for the type *TYPE* .

−updateto:PATH
When updating documentation, write the updated documentation files into the directory *PATH* .

−V, −version
Display version and licensing information.

DESCRIPTION

monodocer has been obsoleted by **mdoc**(1). See the **mdoc-update**(1) man page.

monodocer is a program that creates XML documentation stubs in the ECMA Documentation Format. It does not rely on documentation found within the source code.

The advantages are:

* **Code readability.** Good documentation is frequently (a) verbose, and (b) filled with examples. (For comparison, compare Microsoft .NET Framework documentation, which is often a page or more of docs for each member, to JavaDoc documentation, which can often be a sentence for each member.)

  Inserting good documentation into the source code can frequently bloat the source file, as the documentation can be longer than the actual method that is being documented.

* **Localization.** In-source documentation formats (such as **/doc**) have no support for multiple human languages. If you need to support more than one human language for documentation purposes, *monodocer* is useful as it permits each language to get its own directory, and *monodocer* can add types/members for each separate documentation directory.

* **Administration.** It's not unusual to have separate documentation and development teams. It's also possible that the documentation team will have minimal experience with the programming language being used. In such circumstances, inline documentation is not desirable as the documentation team could inadvertantly insert an error into the source code while updating the documentation. Alternatively, you may not want the documentation team to have access to the source code for security reasons. *monodocer* allows the documentation to be kept *completely* separate and distinct from the source code used to create the assembly.

To turn the *monodocer* documentation into something that can be consumed by the Mono Documentation Browser (the desktop help browser, or the web interface for it) it is necessary to compile the documentation into a packed format. This is done with the mdassembler tool, for example, you could use this toolchain like this:

```
$ monodocer -assembly:MyWidgets -path:generated_docs
$ mdassembler --ecma generated_docs -out:MyWidgets
```

The above would generate a MyWidgets.zip and a MyWidgets.tree that can then be installed in the system. In addition to the two files (.zip and .tree) you must provide a .sources file which describes where in the help system the documentation should be hooked up, it is a very simple XML file, like this:

```
<?xml version="1.0"?>
<monodoc>
  <source provider="ecma" basefile="MyWidgets" path="classlib-gnome"/>
</monodoc>
```

The above configuration file describes that the documentation is in ECMA format (the compiled version) that the base file name is MyWidgets and that it should be hooked up in the "classlib-gnome" part of the tree. If you want to look at the various nodes defined in the documentation, you can look at monodoc.xml file which is typically installed in /usr/lib/monodoc/monodoc.xml.

Once you have all of your files (.zip, .tree and .sources) you can install them into the system with the following command:

$ cp MyWidgets.tree MyWidgets.zip MyWidgets.source ‘pkg-config monodoc --variable sourcesdir‘

The above will copy the files into the directory that Monodoc has registered (you might need root permissions to do this). The actual directory is returned by the *pkg-config* invocation.

STRING ID FORMAT

String IDs are used to refer to a type or member of a type. String IDs are documented in ECMA-334 3rd Edition, Annex E.3.1. They consist of a *member type prefix* , the full type name (namespace + name, separated by ’.’), possibly followed by the member name and other information.

Member type prefixes:

E: The String ID refers to an event. The event name follows the type name: *E:System.AppDomain.AssemblyLoad*

F: The String ID refers to a field. The field name follows the type name: *F:System.Runtime.InteropServices.DllImportAttribute.SetLastError*

M: Refers to a constructor or method. Constructors append *.ctor* to the type name, while methods append the method name (with an optional count of the number of generic parameters).

If the constructor or method take arguments, these are listed within parenthesis after the constructor/method name:

M:System.Object..ctor , *M:System.String..ctor(System.Char[])* , *M:System.String.Concat(System.Object)* , *M:System.Array.Sort“1(“0[])* , *M:System.Collections.Generic.List‘1..ctor* , *M:System.Collections.Generic.List‘1.Add(‘0)* .

N: Refers to a namespace, e.g. *N:System*

P: Refers to a property. If the property is an indexer or takes parameters, the parameter types are appended to the property name and enclosed with parenthesis: *P:System.String.Length* , *P:System.String.Chars(System.Int32)* .

T: The String ID refers to a type, with the number of generic types appended: *T:System.String* , *T:System.Collections.Generic.List‘1*

To make matters more interesting, generic types & members have two representations: the "unbound" representation (shown in examples above), in which class names have the count of generic parameters appended to their name. There is also a "bound" representation, in which the binding of generic parameters is listed within ’{’ and ’}’.

Unbound: *T:System.Collections.Generic.List‘1* , *T:System.Collections.Generic.Dictionary‘2* .

Bound: *T:System.Collections.Generic.List{System.Int32} T:System.Collections.Generic.Dictionary{System.String,System.Collections.Generic.List{System.Predicate{System.String}}}* .

As you can see, bound variants can be arbitrarily complex (just like generics).

Furthermore, if a generic parameter is bound to the generic parameter of a type or method, the "index" of the type/method’s generic parameter is used as the binding, so given

```
class FooType {
  public static void Foo<T> (System.Predicate<T> predicate) {}
}
```

The String ID for this method is *M:FooType.Foo“1(System.Predicate{“0})* , as *“0* is the 0th generic parameter index which is bound to *System.Predicate<T>* .

DOCUMENTATION FORMAT

monodocer generates documentation similar to the Ecma documentation format, as described in ECMA-335 3rd Edition, Partition IV, Chapter 7.

The principal difference from the ECMA format is that each type gets its own file, within a directory identical to the namespace of the type.

Most of the information within the documentation should *not* be edited. This includes the type name (

/Type/@FullName), implemented interfaces (*/Type/Interfaces*), member information (*/Type/Members/Member/@MemberName* , */Type/Members/Member/MemberSignature* , */Type/Members/Member/MemberType* , */Type/Members/Member/Parameters* , etc.).

What *should* be modified are all elements with the text *To be added.* , which are present under the *//Docs* elements (e.g. */Type/Docs* , */Type/Members/Member/Docs*). The contents of the *Docs* element is *identical* in semantics and structure to the inline C# documentation format, consisting of these elements (listed in ECMA-334 3rd Edition, Annex E, Section 2). The following are used within the element descriptions:

CREF Refers to a class (or member) reference, and is a string in the format described above in the *STRING ID FORMAT* section.

TEXT Non-XML text, and XML should not be nested.

XML Only XML elements should be nested (which indirectly may contain text), but non-whitespace text should not be an immediate child node.

XML_TEXT
Free-form text and XML, so that other XML elements may be nested.

The following elements are used in documentation:

<block subset="SUBSET" type="TYPE">XML_TEXT</block>
Create a block of text, similar in concept to a paragraph, but is used to create divisions within the text. To some extent, a <block/> is equivalent to the HTML <h2/> tag.

SUBSET should always be the value *none* .

TYPE specifies the heading and formatting to use. Recognized types are:

behaviors Creates a section with the heading *Operation* .

note Creates a section with the heading *Note:* .

overrides Creates a section with the heading *Note to Inheritors* .

usage Creates a section with the heading *Usage* .

<c>XML_TEXT</c>
Set text in a code-like font (similar to the HTML <tt/> element).

<code lang="LANGUAGE">TEXT</code>
Display multiple lines of text in a code-like font (similar to the HTML <pre/> element). *LANGUAGE* is the language this code block is for. For example, if *LANGUAGE* is **C#** , then *TEXT* will get syntax highlighting for the C# language within the Mono Documentation Browser.

<example>XML_TEXT</example>
Indicates an example that should be displayed specially. For example:

```
<example>
 <para>An introductory paragraph.</para>
 <code lang="C#">
  class Example {
   public static void Main ()
   {
    System.Console.WriteLine ("Hello, World!");
   }
  }
 </code>
</example>
```

<exception cref="CREF">XML_TEXT</exception>
Identifies an exception that can be thrown by the documented member.

<exception/> is a top-level element, and should be nested directly under the *<Docs/>* element.

CREF is the exception type that is thrown, while *XML_TEXT* contains the circumstances that would cause *CREF* to be thrown.

```
<exception cref="T:System.ArgumentNullException">
  <paramref name="foo" /> was <see langword="null" />.
</exception>
```

<list>XML</list>

Create a list or table of items. *<list/>* makes use of nested *<item>XML</item>* , *<listheader>XML</listheader>* , *<term>XML_TEXT</term>* , and *<description>XML_TEXT</description>* elements.

Lists have the syntax:

```
<list type="bullet"> <!-- or type="number" -->
  <item><term>Bullet 1</term></item>
  <item><term>Bullet 2</term></item>
  <item><term>Bullet 3</term></item>
</list>
```

Tables have the syntax:

```
<list type="table">
  <listheader> <!-- listheader bolds this row -->
    <term>Column 1</term>
    <description>Column 2</description>
    <description>Column 3</description>
  </listheader>
  <item>
    <term>Item 1-A</term>
    <description>Item 1-B</description>
    <description>Item 1-C</description>
  </item>
  <item>
    <term>Item 2-A</term>
    <description>Item 2-B</description>
    <description>Item 2-C</description>
  </item>
</list>
```

<para>XML_TEXT</para>

Insert a paragraph of *XML_TEXT* . This is for use within other tags, such as *<example/>* , *<remarks/>* , *<returns/>* , *<term/>* and *<description/>* (see *<list/>* , above), and most other elements.

For example,

```
<para>This is a paragraph of text.</para>
```

<param name="NAME">XML_TEXT</param>

<param/> is a top-level element, and should be nested directly under the *<Docs/>* element.

Describes the parameter *NAME* of the current constructor, method, or property:

```
<param name="count">
  A <see cref="T:System.Int32" /> containing the number
  of widgets to process.
</param>
```

<paramref name="NAME" />

Indicates that *NAME* is a parameter.

This usually renders *NAME* as italic text, so it is frequently (ab)used as an equivalent to the HTML <i/> element. See the *<exception/>* documentation (above) for an example.

<permission cref="CREF">XML_TEXT</permission>

Documentes the security accessibility requirements of the current member.

<permission/> is a top-level element, and should be nested directly under the *<Docs/>* element.

CREF is a type reference to the security permission required, while *XML_TEXT* is a description of why the permission is required.

```
<permission cref="T:System.Security.Permissions.FileIOPermission">
 Requires permission for reading and writing files. See
 <see cref="F:System.Security.Permissions.FileIOPermissionAccess.Read" />,
 <see cref="F:System.Security.Permissions.FileIOPermissionAccess.Write" />.
</permission>
```

<remarks>XML_TEXT</remarks>

Contains detailed information about a member.

<remarks/> is a top-level element, and should be nested directly under the *<Docs/>* element.

```
<remarks>Insert detailed information here.</remarks>
```

<returns>XML_TEXT</returns>

<remarks/> is a top-level element, and should be nested directly under the *<Docs/>* element.

Describes the return value of a method:

```
<returns>
 A <see cref="T:System.Boolean" /> specifying whether
 or not the process can access
 <see cref="P:Mono.Unix.UnixFileSystemInfo.FullName" />.
</returns>
```

<see cref="CREF" />

Creates a link to the specified member within the current text:

```
<see cref="M:Some.Namespace.With.Type.Method" />
```

<seealso cref="CREF" />

<seealso/> is a top-level element, and should be nested directly under the *<Docs/>* element.

Allows an entry to be generated for the *See Also* subclause. Use *<see/>* to specify a link from within text.

```
<seealso cref="P:System.Exception.Message" />
```

<since version="VERSION" />

<since/> is a top-level element, and should be nested directly under the *<Docs/>* element.

Permits specification of which version introduced the specified type or member.

```
<since version="Gtk# 2.4" />
```

<summary>DESCRIPTION</summary>

<summary/> is a top-level element, and should be nested directly under the *<Docs/>* element.

Provides a (brief!) overview about a type or type member.

This is usually displayed as part of a class declaration, and should be a reasonably short description of the type/member. Use *<remarks/>* for more detailed information.

<typeparam name="NAME">DESCRPITION</typeparam>

<typeparam/> is a top-level element, and should be nested directly under the *<Docs/>* element.

This is used to describe type parameter for a generic type or generic method.

NAME is the name of the type parameter, while *DESCRIPTION* contains a description of the parameter (what it's used for, what restrictions it must meet, etc.).

```
<typeparam name="T">The type of the underlying collection</typeparam>
```

<typeparamref>

Used to indicate that a word is a type parameter, for use within other text blocks (e.g. within *<para/>*).

<para>If <typeparamref name="T" /> is a struct, then...</para>

<value>DESCRIPTION</value>

<value/> is a top-level element, and should be nested directly under the *<Docs/>* element.

Allows a property to be described.

<value>
A <see cref="T:System.String" /> containing a widget name.
</value>

SEE ALSO

mdassembler(1), mdcs2ecma(1), mdnormalizer(1), mdvalidator(1), monodocs2html(1)

MAILING LISTS

Visit http://lists.ximian.com/mailman/listinfo/mono-docs-list for details.

WEB SITE

Visit http://www.mono-project.com for details