

EDoc

Copyright © 2006-2024 Ericsson AB. All Rights Reserved.
EDoc 1.2.1
May 2, 2024

Copyright © 2006-2024 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

May 2, 2024

1 EDoc User's Guide

EDoc is the Erlang program documentation generator. Inspired by the Javadoc (TM) tool for the Java (TM) programming language, EDoc is adapted to the conventions of the Erlang world, and has several features not found in Javadoc.

1.1 Welcome to EDoc

EDoc is the Erlang program documentation generator. Inspired by the Javadoc(TM) tool for the Java(TM) programming language, EDoc is adapted to the conventions of the Erlang world, and has several features not found in Javadoc.

EDoc can generate static HTML documentation accessible with any web browser or **EEP-48** doc chunks with erlang +html(3) to provide documentation for other tools like shell_docs(3).

1.1.1 Contents

- Introduction
- Running EDoc
- The overview page
- Generic tags
- Overview tags
- Module tags
- Function tags
- References
- Notes on XHTML
- Wiki notation
- Macro expansion
- Type specifications
- Doc chunks
- Acknowledgements

1.1.2 Introduction

EDoc lets you write the documentation of an Erlang program as comments in the source code itself, using **tags** on the form "`@Name ...`". A source file does not have to contain tags for EDoc to generate its documentation, but without tags the result will only contain the basic available information that can be extracted from the module.

A tag must be the first thing on a comment line, except for leading '%' characters and whitespace. The comment must be between program declarations, and not on the same line as any program text. All the following text - including consecutive comment lines - up until the end of the comment or the next tagged line, is taken as the **content** of the tag.

Tags are associated with the nearest following program construct "of significance" (the module name declaration and function definitions). Other constructs are ignored; e.g., in:

```
%% @doc Prints the value X.

-record(foo, {x, y, z}).

print(X) -> ...
```

the `@doc` tag is associated with the function `print/1`.

Note that in a comment such as:

```
% % @doc ...
```

the tag is ignored, because only the first '%' character is considered "leading". This allows tags to be "commented out".

Some tags, such as `@type`, do not need to be associated with any program construct. These may be placed at the end of the file, in the "footer".

1.1.3 Running EDoc

The following are the main functions for running EDoc:

- edoc:application/2: Creates documentation for a typical Erlang application.
- edoc:files/2: Creates documentation for a specified set of source files.
- edoc:run/2: General interface function; the common back-end for the above functions. Options are documented here.

Note that the function edoc:file/2 belongs to the old, deprecated interface (from EDoc version 0.1), and should not be used.

It's also possible to run EDoc directly from the command line, using the bin/edoc EScript. The script acts as a command line entry point to both edoc:application/2 and edoc:files/2 functions. It also allows to generate just the EEP-48 doc chunks (using the `-chunks` flag) instead of the complete HTML documentation.

1.1.4 The overview page

When documentation is generated for an entire application, an overview page, or "front page", is generated. (The page you are now reading is an overview page.) This should contain the high-level description or user manual for the application, leaving the finer details to the documentation for individual modules. By default, the overview page is generated from the file `overview.edoc` in the target directory (typically, this is the `doc` subdirectory of the application directory); see edoc_doclet for details.

The format of the overview file is the same as for EDoc documentation comments (see Introduction), except that the lines do not have leading '%' characters. Furthermore, all lines before the first tag line are ignored, and can be used as a comment. All tags in the overview file, such as `@doc`, `@version`, etc., refer to the application as a whole; see Overview tags for details.

Here is an example of the contents of an overview file:

```
** this is the overview.doc file for the application 'frob' **

@author R. J. Hacker <rjh@acme.com>
@copyright 2007 R. J. Hacker
@version 1.0.0
@title Welcome to the `frob' application!
@doc `frob' is a highly advanced frobnicator with low latency,
...
```

1.1.5 Generic tags

The following tags can be used anywhere within a module:

`@clear`

This tag causes all tags above it (up to the previous program construct), to be discarded, including the `@clear` tag itself. The text following the tag is also ignored. **This is typically only useful in code containing conditional compilation, when preprocessing is turned on.** (Preprocessing is turned off by default.) E.g., in

```
-ifdef(DEBUG).
%% @doc ...
foo(...) -> ...
-endif.
%% @clear

%% @doc ...
bar(...) -> ...
```

the `@clear` tag makes sure that EDoc does not see two `@doc` tags before the function `bar`, even if the code for function `foo` is removed by preprocessing. (There is no way for EDoc to see what the first `@doc` tag "really" belongs to, since preprocessing strips away all such information.)

`@docfile`

Reads a plain documentation file (on the same format as an overview file - see The overview page for details), and uses the tags in that file as if they had been written in place of the `@docfile` tag. The content is the name of the file to be read; leading and trailing whitespace is ignored. See also @headerfile.

`@end`

The text following this tag is always ignored. Use this to mark the end of the previous tag, when necessary, as e.g. in:

```
%% ---------------------------------
%% ...
%% @doc ...
%% ...
%% @end
%% ---------------------------------
```

to avoid including the last "ruler" line in the `@doc` tag.

Note: using some other "dummy" @-tag for the same purpose might work in a particular implementation of EDoc, but is not guaranteed to. Always use @end to ensure future compatibility.

`@headerfile`

Similar to the @docfile tag, but reads a file containing Erlang source code - generally this should be a header file (with the extension `.hrl`). If the file turns out to contain one or more function definitions or a module declaration, all tags that occur above the last such definition or module declaration are ignored, and EDoc will print a warning. This tag allows you to write documentation in a header file and insert it at a specific place in the documentation, even if the header file is used (i.e., included) by several modules. The `includes` option can be used to specify a search path (see edoc:read_source/2).

`@todo` (or `@TODO`)

Attaches a To-Do note to a function, module or overview-page. The content can be any XHTML text describing the issue, e.g.:

```
%% @TODO Finish writing the documentation.
```

or

```
%% @todo Implement <a href="http://www.ietf.org/rfc/rfc2549.txt">RFC 2549</a>.
```

These tags can also be written as "`TODO:`", e.g.:

```
%% TODO: call your mother
```

see Wiki notation for more information. To-Do notes are normally not shown unless the `todo` option is turned on (see edoc:get_doc/2).

`@type`

Documents an abstract data type or type alias. The content consists of a type declaration or definition, optionally followed by a period ('`.`') separator and XHTML text describing the type (i.e., its purpose, use, etc.). There must be at least one whitespace character between the '`.`' and the text. See Type specifications for syntax and examples. All data type descriptions are placed in a separate section of the documentation, regardless of where the tags occur.

Instead of specifying the complete type alias in an EDoc documentation comment, type definitions from the actual Erlang code can be re-used for documentation. See Type specifications for examples.

1.1.6 Overview tags

The following tags can be used in an overview file.

`@author`

See the @author module tag for details.

`@copyright`

See the @copyright module tag for details.

`@doc`

See the @doc module tag for details.

`@reference`

See the @reference module tag for details.

`@see`

See the @see module tag for details.

`@since`

See the @since module tag for details.

`@title`

Specifies a title for the overview page. This tag can **only** be used in an overview file. The content can be arbitrary text.

`@version`

See the @version module tag for details.

1.1.7 Module tags

The following tags can be used before a module declaration:

`@author`

Specifies the name of an author, along with contact information. An e-mail address can be given within `<...>` delimiters, and a URI within `[...]` delimiters. Both e-mail and URI are optional, and any surrounding whitespace is stripped from all strings.

The name is the first nonempty string that is not within `<...>` or `[...]`, and does not contain only whitespace. (In other words, the name can come before, between, or after the e-mail and URI, but cannot be split up; any sections after the first are ignored.) If an e-mail address is given, but no name, the e-mail string will be used also for the name. If no `<...>` section is present, but the name string contains an '@' character, it is assumed to be an e-mail address. Not both name and e-mail may be left out.

Examples:

```
%% @author Richard Carlsson
```

```
%% @author Richard Carlsson <carlsson.richard@gmail.com>
%%   [http://example.net/richardc/]
```

```
%% @author <carlsson.richard@gmail.com>
```

```
%% @author carlsson.richard@gmail.com [http://example.net/richardc/]
```

`@copyright`

Specifies the module copyrights. The content can be arbitrary text; for example:

```
%% @copyright 2001-2003 Richard Carlsson
```

`@deprecated`

Mark the module as deprecated, indicating that it should no longer be used. The content must be well-formed XHTML, and should preferably include a `{@link}` reference to a replacement; as in:

```
%% @deprecated Please use the module {@link foo} instead.
```

`@doc`

Describes the module, using well-formed XHTML text. The first sentence is used as a summary (see the @doc function tag for details). For example.:

```
%% @doc This is a <em>very</em> useful module. It is ...
```

`@hidden`

Marks the module so that it will not appear in the documentation (even if "private" documentation is generated). Useful for sample code, test modules, etc. The content can be used as a comment; it is ignored by EDoc.

`@private`

Marks the module as private (i.e., not part of the public interface), so that it will not appear in the normal documentation. (If "private" documentation is generated, the module will be included.) The content can be used as a comment; it is ignored by EDoc.

`@reference`

Specifies a reference to some arbitrary external resource, such as an article, book, or web site. The content must be well-formed XHTML text. Examples:

```
%% @reference Pratchett, T., <em>Interesting Times</em>,
%% Victor Gollancz Ltd, 1994.
```

```
%% @reference See <a href="www.google.com">Google</a> for
%% more information.
```

`@see`

See the @see function tag for details.

`@since`

Specifies when the module was introduced, with respect to the application, release or distribution it is part of. The content can be arbitrary text.

`@version`

Specifies the module version. The content can be arbitrary text.

1.1.8 Function tags

The following tags can be used before a function definition:

`@deprecated`

See the @deprecated module tag for details.

`@doc`

XHTML text describing the function. The first sentence of the text is used as a quick summary; this ends at the first period character (`'.'`) or exclamation mark (`'!'`) that is followed by a whitespace character, a line break, or the end of the tag text, and is not within XML markup. (As an exception, the first sentence may be within an initial paragraph element)

`@equiv`

Specify equivalence to another function call/expression. The content must be a proper Erlang expression. If the expression is a function call, a cross-reference to the called function is created automatically. Typically, this tag is used instead of `@doc`.

`@hidden`

Marks the function so that it will not appear in the documentation (even if "private" documentation is generated). Useful for debug/test functions, etc. The content can be used as a comment; it is ignored by EDoc.

`@param`

Provide more information on a single parameter of the enclosing function. The content consists of a parameter name, followed by one or more whitespace characters, and XHTML text.

`@private`

Marks the function as private (i.e., not part of the public interface), so that it will not appear in the normal documentation. (If "private" documentation is generated, the function will be included.) Only useful for exported functions, e.g. entry points for `spawn`. (Non-exported functions are always "private".) The content can be used as a comment; it is ignored by EDoc.

`@returns`

Specify additional information about the value returned by the function. Content consists of XHTML text.

`@see`

Make a reference to a module, function, datatype, or application. (See References.) The content consists of a reference, optionally followed by a period (`'.'`), one or more whitespace characters, and XHTML text to be used for the label; for example "`@see edoc`" or "`@see edoc. <b>EDoc</b>`". If no label text is specified, the reference itself is used as the label.

`@since`

Specifies in what version of the module the function was introduced; cf. the @version module tag. The content can be arbitrary text.

`@spec`

Used to specify the function type; see Type specifications for syntax details. If the function name is included in the specification, it must match the name in the actual code. When parameter names are not given in the specification, suitable names will be taken from the source code if possible, and otherwise synthesized.

Instead of specifying the complete function type in an EDoc documentation comment, specifications from the actual Erlang code can be re-used for documentation. See Type specifications for examples.

`@throws`

Specifies which types of terms may be thrown by the function, if its execution terminates abruptly due to a call to `erlang:throw(Term)`. The content is a type expression (see Type specifications), and can be a union type.

Note that exceptions of type `exit` (as caused by calls to `erlang:exit(Term)`) and `error` (run-time errors such as `badarg` or `badarith`) are not viewed as part of the normal interface of the function, and cannot be documented with the `@throws` tag.

`@type`

See the @type generic tag for details. Placing a `@type` tag by a function definition may be convenient, but does not affect where the description is placed in the generated documentation.

1.1.9 References

In several contexts (`@see` tags, `@link` macros, etc.), EDoc lets you refer to the generated documentation for modules, functions, datatypes, and applications, using a simple and compact syntax. The possible formats for references are:

Reference syntax	**Example**	**Scope**
`Module`	edoc_run, `erl.lang.list`	Global
`Function/Arity`	`file/2`	Within module
`Module:Function/Arity`	edoc:application/2	Global
`Type()`	`filename()`	Within module
`Module:Type()`	edoc:edoc_module()	Global
`//Application`	edoc	Global
`//Application/Module`	edoc_doclet(3)	Global
`//Application/ Module:Function/Arity`	edoc_run:file/1	Global
`//Application/ Module:Type()`	edoc:edoc_module()	Global

Table 1.1: reference syntax

EDoc will resolve references using the information it finds in `edoc-info`-files at the locations specified with the `doc_path` option. EDoc will automatically (and somewhat intelligently) try to find any local `edoc-info`-files using the current code path, and add them to the end of the `doc_path` list. The target doc-directory is also searched for an existing info file; this allows documentation to be built incrementally. (Use the `new` option to ignore any old info file.)

Note that if the name of a module, function or datatype is explicitly qualified with an application (as in "`//edoc/edoc_run`"), this overrides any other information about that name, and the reference will be made relative to the location of the application (if it can be found). This makes it possible to refer to e.g. a module "`fred`" as "`//foo/fred`" without accidentally getting a reference to "`//bar/fred`". You should not use this form of explicit references for names that are local to the application you are currently creating - they will always be resolved correctly.

Note that module-local references such as `file/2` only work properly within a module. In an overview-page like this (i.e., the one you are currently reading), no module context is available.

1.1.10 Notes on XHTML

In several places, XHTML markup can be used in the documentation text, in particular in `@doc` tags. The main differences from HTML are the following:

- All elements must have explicit start and end tags, and be correctly nested. This means that you cannot e.g. write a `<li>` tag without also writing a corresponding `</li>` tag in the right place. This could be an annoyance at times, but has the great advantage that EDoc can report all malformed XHTML in your source code, rather than propagate the errors to the generated documentation.
- XHTML tag and attribute names should always be lower-case.
- Attributes must be quoted, as in e.g. `<a name="top">`.

To write an element like the HTML `<br>`, which has no actual content, you can write either the full `<br></br>`, or better, use the XHTML abbreviated form `<br/>`.

Since the purpose of EDoc is to document programs, there is also a limited form of "wiki"-syntax available for making program code easier to write inline (and to make the doc-comments easier to read). See Wiki notation for details.

The HTML heading tags `h1` and `h2` are reserved for use by EDoc. Headings in documentation source code should start at `h3`. There is however a special syntax for writing headings which avoids using specific level numbers altogether; see Headings for details.

EDoc uses XMerL to parse and export XML markup.

1.1.11 Wiki notation

When EDoc parses XHTML, it does additional pre- and post-processing of the text in order to expand certain notation specific to EDoc into proper XHTML markup. This "wiki" (**http://en.wikipedia.org/wiki/Wiki**) notation is intended to make it easier to write source code documentation.

Empty lines separate paragraphs

Leaving an empty line in XHTML text (i.e., a line which except for any leading start-of-comment '%' characters contains only whitespace), will make EDoc split the text before and after the empty line into separate paragraphs. For example:

```
%% @doc This will all be part of the first paragraph.
%% It can stretch over several lines and contain <em>any
%% XHTML markup</em>.
%%
%% This is the second paragraph. The above line is
%% regarded as "empty" by EDoc, even though it ends with
%% a space.
```

will generate the following text:

*This will all be part of the first paragraph. It can stretch over several lines and contain **any XHTML markup**.*

This is the second paragraph. The above line is regarded as "empty" by EDoc, even though it ends with a space.

Paragraph splitting takes place after the actual XHTML parsing. It only affects block-level text, and not e.g., text within `<pre>` markup, or text that is already within `<p>` markup.

Headings

Section headings, sub-headings, and sub-sub-headings, can be written using the following notation:

```
== Heading ==
=== Sub-heading ===
==== Sub-sub-heading ====
```

Such a heading must be alone on a line, except for whitespace, and cannot be split over several lines. A link target is automatically created for the heading, by replacing any whitespace within the text by a single underscore character. E.g.,

```
== Concerning Hobbits ==
```

is equivalent to

```
<h3><a name="Concerning_Hobbits">Concerning Hobbits</a></h3>
```

Thus, headings using this notation should not contain characters that may not be part of URL labels, except for whitespace. If you need to create such headings, you have to use the explicit XHTML markup.

A hypertext link to a heading written this way can be created using the `@section` macro, which transforms the argument text into a label as described above. E.g.,

```
{@section Concerning Hobbits}
```

is equivalent to writing

```
<a href="#Concerning_Hobbits">Concerning Hobbits</a>
```

The above expansions take place before XML parsing.

External links

Writing a URL within brackets, as in "`[http://www.w3c.org/]`", will generate a hyperlink such as **http://www.w3c.org/**, using the URL both for the destination and the label of the reference, equivalent to writing "`<a href="http://www.w3c.org/"><code>http://www.w3c.org/</code></a>`". This short-hand keeps external URL references short and readable. The recognized protocols are `http`, `ftp`, and `file`. This expansion takes place before XML parsing.

TODO-notes

Lines that begin with the text "`TODO:`" (the colon is required) are recognized as tags, as if they had been written as "`@todo ...`" (see @todo tags for further details).

Verbatim quoting

In XHTML text, the '`' character (Unicode `000060`, known as "grave accent" or "back-quote") can be used for verbatim quoting. This expansion takes place before XML parsing.

- A character sequence "`` `...' ``" or "`` ``...'' ``" will be expanded to "`<code>...</code>`", where all occurrences of the special XML characters '<' and '&' (and for completeness, also '>') in the quoted text have been escaped to "`<`", "`&`", and "`>`", respectively. All whitespace is stripped from the beginning and end of the quoted text.

Double back-quotes "``...''" can be used to quote text containing single '' characters. The automatic stripping of any surrounding whitespace makes it possible to write things like "`` 'foo@bar' ''".

To quote text containing "''" verbatim, explicit <code> markup or similar must be used.

- A character sequence "```...'''" will be expanded to "<pre><![CDATA[...]]></pre>", which disables all XML markup within the quoted text, and displays the result in fixed-font with preserved indentation. Whitespace is stripped from the end of the quoted text, but not from the beginning, except for whole leading lines of whitespace. This is useful for multi-line code examples, or displayed one-liners.
- To produce a single '`'-character in XML without beginning a new quote, you can write "`'" (no space between the '`' and the ''). You can of course also use the XML character entity "`".

Examples:

```
%% @doc ...where the variable `Foo' refers to...
```

```
%% @doc ...returns the atom `` 'foo@erlang.org' ''...
```

```
%% @doc ...use the command ```erl -name foo''' to...
```

```
%% @doc ...as in the following code:
%% ```f(X) ->
%%       case X of
%%          ...
%%       end'''
```

```
%% @doc ...or in the following:
%% ```
%%     g(X) ->
%%       fun () -> ... end
%% '''
```

1.1.12 Macro expansion

Before the content of a tag is parsed, the text undergoes **macro expansion**. The syntax for macro calls is:

```
{@name}
```

or

```
{@name argument}
```

where **name** and **argument** are separated by one or more whitespace characters. The argument can be any text, which may contain other macro calls. The number of non-escaped "{@" and "}" delimiters must be balanced.

The argument text is first expanded in the current environment, and the result is bound to the **macro parameter**, written {@?}. (If no argument is given, {@?} is bound to the empty string.) The macro definition is then substituted for the call, and expansion continues over the resulting text. Recursive macro expansions are not allowed.

User-defined macros

Users can define their own macros by using the def EDoc option; see edoc:file/2 and edoc:get_doc/2 for more information. User-defined macros override predefined macros.

Predefined macros

`{@date}`

Expands to the current date, as "`Month Day Year`", e.g. "May 2 2024".

`{@link reference. description}`

This creates a hypertext link; cf. the @see function tag above for details. The description text (including the period separator) is optional; if no text is given, the reference itself is used. For example, `{@link edoc:file/2}` creates the link edoc:file/2, and `{@link edoc:file/2. <em>this link</em>}` creates this link.

`{@module}`

Expands to the name of the current module. Only defined when a module is being processed.

`{@section heading}`

Expands to a hypertext link to the specified section heading; see Headings for more information.

`{@time}`

Expands to the current time, as "`Hr:Min:Sec`", e.g. "17:18:15".

`{@type type-expression}`

Formats a type expression within `<code>...</code>` markup and with hypertext links for data types. For example, `{@type {options, List::edoc:option_list()@}}` generates "`{options, List::edoc:option_list()}`". (Cf. Escape sequences.)

`{@version}`

Intended for use in @version tags. Defaults to a timestamp using `{@date}` and `{@time}`. Typically, this macro is redefined by the user when an official release of the application is generated.

Escape sequences

To prevent certain characters from being interpreted as delimiters, for example to produce the text "`{@`" in the output, or use a '`}`' character in the argument text of a macro call, the following escape sequences may be used:

`@{`

Expands to "`{`". Example:

```
%% @doc A macro call starts with the sequence "@{@".
```

`@}`

Expands to "`}`". Example:

```
%% @doc ...{@foo ...{Key, Value@}...}...
```

`@@`

Expands to "@". Example:

```
%% @doc Contact us at support@@{@hostname}
```

Will generate the text "Contact us at support@vaporware.acme.com" if the macro `hostname` is bound to "`vaporware.acme.com`". Also:

```
%% @doc You might want to write something like
%% @@foo that will expand to @foo and does not start
%% a new tag even if it appears first in a line.
```

1.1.13 Type specifications

Function specifications

Note that although the syntax described in the following can still be used for specifying functions we recommend that Erlang specifications (that is the `-spec` and `-type` attributes) as described in Types and Function Specification should be added to the source code instead. This way the analyses of Dialyzer's can be utilized in the process of keeping the documentation consistent and up-to-date.

Erlang specifications (`-spec` and `-type`) are required to properly generate doc chunks. Redundant `-spec` attributes and `@spec` tags will cause warnings to be emitted and the specifications to be skipped in chunks. Old-style `@spec` and `@type` specifications can still be used to generate static HTML documentation.

The following grammar describes the form of the specifications following a `@spec` tag. A '?' suffix implies that the element is optional. Function types have higher precedence than union types; e.g., "`(atom()) -> atom() | integer()`" is parsed as `((atom()) -> atom()) | integer()`, not as `(atom()) -> (atom() | integer())`.

`Spec`	::=	`FunType "where"? DefList? | FunctionName FunType "where"? DefList?`
`FunctionName`	::=	`Atom`
`FunType`	::=	`"(" UnionTypes? ")" "->" UnionType`
`UnionTypes`	::=	`UnionType | UnionType "," UnionTypes`
`UnionType`	::=	`UnionList | Name "::" UnionList`
`Name`	::=	`Variable`
`UnionList`	::=	`Type | Type "+" UnionList | Type "|" UnionList`
`Type`	::=	`TypeVariable | Atom | Integer | Float | Integer ".." Integer | FunType | "fun(" FunType ")" | "fun(...)" | "{" UnionTypes? "}" | "#" Atom "{" Fields? "}" | "[" "]" | "[" UnionType "]" | "[" UnionType "," "..." "]" | "(" UnionType ")" | BinType | TypeName "(" UnionTypes? ")" |`

		ModuleName ":" TypeName "(" UnionTypes? ")" | "//" AppName "/" ModuleName ":" TypeName "(" UnionTypes? ")"
Fields	::=	Field | Fields "," Fields
Field	::=	Atom "=" UnionList
BinType	::=	"<<>>" | "<<" BaseType ">>" | "<<" UnitType ">>" | "<<" BaseType "," UnitType ">>"
BaseType	::=	"_" ":" Integer
UnitType	::=	"_" ":" "_" "*" Integer
TypeVariable	::=	Variable
TypeName	::=	Atom
ModuleName	::=	Atom | ModuleName "." Atom
AppName	::=	Atom
DefList	::=	Def | DefList Def | DefList "," Def
Def	::=	TypeVariable "=" UnionList | TypeName "(" TypeVariables? ")" "=" UnionType
TypeVariables	::=	TypeVariable | TypeVariable "," TypeVariables

Table 1.2: specification syntax grammar

Examples:

```
-spec my_function(X :: integer()) -> integer().
%% @doc Creates ...
```

```
%% @spec my_function(X::integer()) -> integer()
```

```
%% @spec (X::integer()) -> integer()
```

```
%% @spec sqrt(float()) -> float()
```

```
%% @spec pair(S, T) -> {S, T}
```

```
%% @spec append(List, List) -> List
%%       List = [term()]
```

```
%% @spec append(A::List, B::List) -> List
%%       List = [Item]
%%       Item = term()
```

```
%% @spec open(File::filename()) -> FileDescriptor
%% where
%%       filename() = string() + atom(),
%%       FileDescriptor = term()
```

```
%% @spec close(graphics:window()) -> ok
```

The first example shows the recommended way of specifying functions.

In the above examples, `X`, `A`, `B`, and `File` are parameter names, used for referring to the parameters from the documentation text. The **type variables** `S`, `T` and `List` are used to simplify the type specifications, and may be supplied with definitions. It is also possible to give definitions for named types, which means that the name is simply an alias. (Use the `@type` tag to document abstract data types.) If a named type is defined in another module, it can be referred to as `Module:TypeName(...)`. Note that the keyword '`where`' is optional before a list of definitions, and that the definitions in the list may optionally be separated by '`,`'.

Both the '`|`' and the '`+`' character may be used to separate alternatives in union types; there is no semantic difference. Note that the notation `[Type]` means "proper (nil-terminated) list whose elements all belong to `Type`"; For example, `[atom()|integer()]` means the same thing as `[atom()+integer()]`, i.e., a proper list of atoms and/or integers.

If only a type variable is given for a parameter, as in "`pair(S, T) -> ...`", the same variable name may implicitly be used as the parameter name; there is no need to write "`pair(S::S, T::T) -> ...`".

EDoc automatically extracts possible parameter names from the source code, to be used if no parameter name is given in the specification (or if the specification is missing altogether). If this fails, EDoc will generate a dummy parameter name, such as `X1`. This way, EDoc can often produce helpful documentation even for code that does not contain any annotations at all.

Type definitions

Note that although the syntax described in the following can still be used for specifying types we recommend that Erlang types as described in Types and Function Specification should be added to the source code instead.

Old-style `@type` tags will result in just type names, but no definitions, in doc chunks - please use `-type` attributes to ensure all available type information is available in the chunks. For static HTML, Erlang types will be used unless there is a type alias with the same name.

The following grammar (see above for auxiliary definitions) describes the form of the definitions that may follow a `@type` tag:

`Typedef`	::=	`TypeName "(" TypeVariables? ")" DefList? | TypeName "(" TypeVariables? ")" "=" UnionList DefList?`

Table 1.3: type definition grammar

(For a truly abstract data type, no equivalence is specified.) The main definition may be followed by additional local definitions. Examples:

```
-type my_list(X) :: [X]. %% A special kind of lists ...
```

```
-opaque another_list(X) :: [X].
%% another_list() is a kind of list...
```

```
%% @type myList(X). A special kind of lists ...
```

```
%% @type filename() = string(). Atoms not allowed!
```

```
%% @type thing(A) = {thong, A}
%%           A = term().
%%   A kind of wrapper type thingy.
```

The first two examples show the recommended way of specifying types.

Pre-defined data types

The following data types are predefined by EDoc, and may not be redefined:

```
any()
arity()
atom()
binary()
bitstring()
bool()        (allowed, but use boolean() instead)
boolean()
byte()
char()
cons()
deep_string()
float()
function()
integer()
iodata()
iolist()
list()
maybe_improper_list()
mfa()
module()
nil()
neg_integer()
node()
non_neg_integer()
nonempty_improper_list()
nonempty_list()
nonempty_maybe_improper_list()
nonempty_string()
none()
number()
pid()
port()
pos_integer()
reference()
string()
term()
timeout()
tuple()
```

Details:

- `any()` means "any Erlang data type". `term()` is simply an alias for `any()`.
- `atom()`, `binary()`, `float()`, `function()`, `integer()`, `pid()`, `port()` and `reference()` are primitive data types of the Erlang programming language.
- `boolean()` is the subset of `atom()` consisting of the atoms `true` and `false`.
- `char()` is the subset of `integer()` representing Unicode character codes: hex 000000-10FFFF.
- `tuple()` is the set of all tuples `{...}`.
- `list(T)` is just an alias for `[T]`; list() is an alias for `list(any())`, i.e., `[any()]`.
- `nil()` is an alias for the empty list `[]`.
- `cons(H,T)` is the list constructor. This is usually not used directly. It is possible to recursively define `list(T) := nil()+cons(T,list(T))`.
- `string()` is an alias for `[char()]`.
- `deep_string()` is recursively defined as `[char()+deep_string()]`.
- `none()` means "no data type". E.g., a function that never returns has type `(...) -> none()`

1.1.14 Doc chunks

EDoc implements **EEP-48** and allows to output doc chunks for Erlang projects which use the EDoc language for source code documentation.

There are two ways to generate the doc chunks: either by using bin/edoc or by using the **Rebar3 edoc command**. Ultimately, they're both just entry points to EDoc, the application, so deciding which to use is just a matter of preference.

Using edoc EScript

In order to generate doc chunks using bin/edoc, make sure the compiled Erlang app can be found in the Erlang code path:

```
$ pwd
/tmp/recon
$ rebar3 compile
$ export PATH="$(dirname `which erl`)/../lib/erlang/lib/edoc-0.12/bin:$PATH"
$ edoc -app recon -chunks -pa _build/default/lib/recon/ebin
Running with opts:
#{app => recon,
  code_paths => ["_build/default/lib/recon/ebin"],
  files => [],mode => chunks,run => app}
$ ls _build/default/lib/recon/doc/chunks/
recon.chunk        recon_lib.chunk    recon_rec.chunk
recon_alloc.chunk  recon_map.chunk    recon_trace.chunk
```

The project does not have to be built with Rebar3 - the above is just an example. From now on the chunks will be available as the source of documentation for the Erlang shell:

```
$ erl -pa _build/default/lib/recon/ebin
...
1> h(recon).

   recon

  Recon, as a module, provides access to the high-level functionality contained in the
  Recon application.
  ...

2> h(recon_alloc, allocators, 0).

  -spec allocators() -> [allocdata(term())].

  returns a dump of all allocator settings and values
```

Using Rebar3 edoc command

Doc chunks can also be built using `rebar3 edoc`, given `edoc_opts` is properly set up:

```
$ pwd
/tmp/recon
$ cat >> rebar.config <<EOF
> {edoc_opts, [{doclet, edoc_doclet_chunks},
>              {layout, edoc_layout_chunks},
>              {preprocess, true},
>              {dir, "_build/docs/lib/recon/doc"}]}.
> EOF
$ rebar3 edoc
$ ls _build/docs/lib/recon/doc/chunks/
recon.chunk        recon_lib.chunk    recon_rec.chunk
recon_alloc.chunk  recon_map.chunk    recon_trace.chunk
```

`rebar3 shell` will set up the paths for us if we use the `docs` profile:

```
$ rebar3 as docs shell
...
1> h(recon).

   recon

  Recon, as a module, provides access to the high-level functionality contained in the
  Recon application.
  ...
```

Using the EDoc API

EDoc comes with two sets of doclet/layout pairs:

- edoc_doclet and edoc_layout - the default pair which is used to generate static HTML documentation
- edoc_doclet_chunks and edoc_layout_chunks - the **EEP-48** compliant pair which generates doc chunks with erlang+html(3)

In order to generate doc chunks using edoc:application/2 or edoc:files/2 we have to specify which doclet and layout we want to use:

```
Opts = [{doclet, edoc_doclet_chunks},
        {layout, edoc_layout_chunks}].
```

Then, it's just a matter of deciding whether we want to generate documentation for the whole application (edoc:application/2) or only selected source files (edoc:files/2):

```
App = my_app.
edoc:application(App, Opts).
%% or
Files = ["src/my_app_mod1.erl", "src/my_app_mod2.erl"].
edoc:files(Files, Opts).
```

See `src/edoc_cli.erl` source code for an example of using this interface.

1.1.15 Acknowledgements

Since the first version of EDoc, several people have come up with suggestions (Luke Gorrie, Joe Armstrong, Erik Stenman, Sean Hinde, Ulf Wiger, ...), and some have even submitted code to demonstrate their ideas (Vlad Dumitrescu, Johan Blom, Vijay Hirani, ...). None of that code was actually included in the Great Rewriting that followed the initial public release (EDoc version 0.1), but most of the central points were addressed in the new system, such as better modularization and possibility to plug in different layout engines, and making EDoc understand the application directory layout.

It is now getting too hard to keep track of all the people who have made further suggestions or submitted bug reports, but your input is always appreciated. Thank you.

2 Reference Manual

EDoc is the Erlang program documentation generator. Inspired by the Javadoc (TM) tool for the Java (TM) programming language, EDoc is adapted to the conventions of the Erlang world, and has several features not found in Javadoc.

edoc

Erlang module

EDoc - the Erlang program documentation generator.

This module provides the main user interface to EDoc.

- EDoc User Manual
- Running EDoc

DATA TYPES

comment() = erl_comment_scan:comment()

edoc_module() = xmerl_scan:xmlElement()

The EDoc documentation data for a module, expressed as an XML document in XMerL format. See the file **edoc.dtd** for details.

entry() = #entry{name=function_name() | atom(), args=[atom() | list()], line=integer(), export=boolean(), data=entry_data()}

Module Entries (one per function, plus module header and footer).

entry_data() = term()

env() = #env{}

Environment information needed by EDoc for generating references.

filename() = file:filename()

function_name() = {atom(), integer()}

module_meta() = #module{name=[] | atom(), parameters=none | [atom()], functions=ordset(function_name()), exports=ordset(function_name()), attributes=ordset({atom(), term()}), records=[{atom(), [{atom(), term()}]}], encoding=epp:source_encoding(), file=file:filename()}

Module information.

ordset(T) = ordsets:ordset(T)

proplist() = proplists:proplist()

syntaxTree() = erl_syntax:syntaxTree()

tag() = #tag{name=atom(), line=integer(), origin=comment | code, data=term(), form=undefined | erl_parse:abstract_form()}

Generic tag information. `#tag.form` is only defined if `#tag.origin` is `code`, that is the `#tag{}` represents a code fragment, not a doc comment tag.

Exports

`application(App::atom()) -> ok`

Equivalent to application(Application, []).

`application(App, Options) -> ok`

Types:

`App = atom()`
`Options = proplist()`

Run EDoc on an application in its default app-directory. See application/3 for details.

See also: application/1.

`application(App, Dir, Options) -> ok`

Types:

`App = atom()`
`Dir = filename()`
`Options = proplist()`

Run EDoc on an application located in the specified directory. Tries to automatically set up good defaults. Unless the user specifies otherwise:

- The `doc` subdirectory will be used as the target directory, if it exists; otherwise the application directory is used.
- The source code is assumed to be located in the `src` subdirectory, if it exists, or otherwise in the application directory itself.
- The subpackages option is turned on. All found source files will be processed.
- The `include` subdirectory is automatically added to the include path. (Only important if preprocessing is turned on.)

See run/2 for details, including options.

See also: application/2.

`file(Name::filename()) -> ok`

This function is deprecated: See file/2 for details.

Equivalent to file(Name, []).

`file(Name, Options) -> ok`

Types:

`Name = filename()`
`Options = proplist()`

This function is deprecated: This is part of the old interface to EDoc and is mainly kept for backwards compatibility. The preferred way of generating documentation is through one of the functions application/2 and files/2.

Reads a source code file and outputs formatted documentation to a corresponding file.

Options:

`{dir, filename()}`

Specifies the output directory for the created file. (By default, the output is written to the directory of the source file.)

`{source_suffix, string()}`

Specifies the expected suffix of the input file. The default value is `".erl"`.

`{file_suffix, string()}`

Specifies the suffix for the created file. The default value is `".html"`.

See get_doc/2 and layout/2 for further options.

For running EDoc from a Makefile or similar, see edoc_run:file/1.

See also: read/2.

```
files(Files::[filename()]) -> ok
```

```
files(Files, Options) -> ok
```

Types:

`Files = [filename()]`
`Options = proplist()`

Runs EDoc on a given set of source files. See run/2 for details, including options.

```
get_doc(File::filename()) -> {module(), edoc_module()}
```

Equivalent to get_doc(File, []).

```
get_doc(File, Options) -> R
```

Types:

`File = filename()`
`Options = proplist()`
`R = {module(), edoc_module()} | {module(), edoc_module(), [entry()]}`

Reads a source code file and extracts EDoc documentation data. Note that without an environment parameter (see get_doc/3), hypertext links may not be correct.

Options:

`{def, Macros}`

- `Macros` = `Macro | [Macro]`
- `Macro` = `{Name::atom(), Text::string() | MacroFun}`
- `MacroFun = fun((MacroArgument::string(), Line::integer(), edoc_lib:edoc_env()) -> (Text::string()))`

Specifies a set of user-defined EDoc macros. The text substituted for macro calls is specified as either a `string()` or a `function()`. The function is called with the macro argument text, the current line number, and the current environment. The fun is to return a `string()`. See Macro expansion for details.

`{hidden, boolean()}`

If the value is `true`, documentation of hidden functions will also be included. The default value is `false`.

`{private, boolean()}`

If the value is `true`, documentation of private functions will also be included. The default value is `false`.

`{todo, boolean()}`

If the value is `true`, To-Do notes written using `@todo` or `@TODO` tags will be included in the documentation. The default value is `false`.

See read_source/2, read_comments/2 and edoc_lib:get_doc_env/3 for further options.

See also: get_doc/3, layout/2, read/2, run/2, edoc_extract:source/5.

`get_doc(File, Env, Options) -> R`

Types:

`File = filename()`
`Env = env()`
`Options = proplist()`
`R = {module(), edoc_module()} | {module(), edoc_module(), [entry()]}`

Like get_doc/2, but for a given environment parameter. `Env` is an environment created by edoc_lib:get_doc_env/3.

`layout(Doc::edoc_module()) -> string()`

Equivalent to layout(Doc, []).

`layout(Doc, Opts) -> string()`

Types:

`Doc = edoc_module()`
`Opts = proplist()`

Transforms EDoc module documentation data to text. The default layout creates an HTML document.

Options:

`{layout, Module::atom()}`

Specifies a callback module to be used for formatting. The module must export a function `module(Doc, Options)`. The default callback module is edoc_layout; see edoc_layout:module/2 for layout-specific options.

See also: file/2, layout/1, read/2, run/2.

`read(File::filename()) -> string()`

Equivalent to read(File, []).

`read(File, Opts) -> string()`

Types:

`File = filename()`
`Opts = proplist()`

Reads and processes a source file and returns the resulting EDoc-text as a string. See get_doc/2 and layout/2 for options.

See also: file/2.

`read_comments(File::filename()) -> [comment()]`

Equivalent to read_comments(File, []).

`read_comments(File, Opts) -> [comment()]`

Types:

`File = filename()`
`Opts = proplist()`

Extracts comments from an Erlang source code file. See the module erl_comment_scan(3) for details on the representation of comments. Currently, no options are available.

`read_source(Name::filename()) -> [syntaxTree()]`

Equivalent to read_source(File, []).

`read_source(File, Opts) -> [syntaxTree()]`

Types:

`File = filename()`
`Opts = proplist()`

Reads an Erlang source file and returns the list of "source code form" syntax trees.

Options:

`{preprocess, boolean()}`

If the value is `true`, the source file will be read via the Erlang preprocessor (`epp`). The default value is `false`. `no_preprocess` is an alias for `{preprocess, false}`.

Normally, preprocessing is not necessary for EDoc to work, but if a file contains too exotic definitions or uses of macros, it will not be possible to read it without preprocessing. **Note: comments in included files will not be available to EDoc, even with this option enabled.**

`{includes, Path::[string()]}`

Specifies a list of directory names to be searched for include files, if the `preprocess` option is turned on. Also used with the `@headerfile` tag. The default value is the empty list. The directory of the source file is always automatically appended to the search path.

`{macros, [{atom(), term()}]}`

Specifies a list of pre-defined Erlang preprocessor (`epp`) macro definitions, used if the `preprocess` option is turned on. The default value is the empty list.

`{report_missing_types, boolean()}`

If the value is `true`, warnings are issued for missing types. The default value is `false`. `no_report_missing_types` is an alias for `{report_missing_types, false}`.

`{link_predefined_types, boolean()}`

If the value is `true`, all predefined data types will have a link to the erlang module. This option is to be used when generating documentation for the Erlang/OTP docs. The default value is `false`. `no_link_predefined_types` is an alias for `{link_predefined_types, false}`.

See also: erl_syntax(3), get_doc/2.

`run(Files, Opts) -> ok`

Types:

`Files = [filename()]`

`Opts = proplist()`

Runs EDoc on a given set of source files. Note that the doclet plugin module has its own particular options; see the `doclet` option below.

Also see layout/2 for layout-related options, and get_doc/2 for options related to reading source files.

Options:

`{app_default, string()}`

Specifies the default base URI for unknown applications.

`{application, App::atom()}`

Specifies that the generated documentation describes the application `App`. This mainly affects generated references.

`{dir, filename()}`

Specifies the target directory for the generated documentation.

`{doc_path, [string()]}`

Specifies a list of file system paths pointing to directories that contain EDoc-generated documentation. All paths for applications in the code path are automatically added.

`{doclet, Module::atom()}`

Specifies a callback module to be used for creating the documentation. The module must export a function `run(Cmd, Ctxt)`. The default doclet module is edoc_doclet; see edoc_doclet:run/2 for doclet-specific options.

`{file_suffix, string()}`

Specifies the suffix used for output files. The default value is `".html"`. Note that this also affects generated references.

`{new, boolean()}`

If the value is `true`, any existing `edoc-info` file in the target directory will be ignored and overwritten. The default value is `false`.

`{source_path, [filename()]}`

Specifies a list of file system paths used to locate the source code for packages.

`{source_suffix, string()}`

Specifies the expected suffix of input files. The default value is `".erl"`.

`{subpackages, boolean()}`

If the value is `true`, all subpackages of specified packages will also be included in the documentation. The default value is `false`. `no_subpackages` is an alias for `{subpackages, false}`.

Subpackage source files are found by recursively searching for source code files in subdirectories of the known source code root directories. (Also see the `source_path` option.) Directory names must begin with a lowercase letter and contain only alphanumeric characters and underscore, or they will be ignored. (For example, a subdirectory named `test-files` will not be searched.)

See also: application/2, files/2.

edoc_cmd

Command

This script is a command line entry point to both edoc:application/2 and edoc:files/2 functions. It also allows to generate just the EEP-48 doc chunks (using the `-chunks` flag) instead of the complete HTML documentation.

Exports

`edoc -app <app> [-chunks]`

EDoc is invoked via `edoc:application/2`, with the default set of options. If `-chunks` is given, then only doc chunks will be generated, not the full HTML documentation.

Run the script with no arguments to get the full list of options.

`edoc -files <erl_file>... [-chunks]`

EDoc is invoked via `edoc:files/2`, with the default set of options. If `-chunks` is given, then only doc chunks will be generated, not the full HTML documentation.

Run the script with no arguments to get the full list of options.

edoc_doclet

Erlang module

Standard doclet module for EDoc.

DATA TYPES

command() = doclet_gen() | doclet_toc()

Doclet commands.

context() = #doclet_context{dir=string(), env=edoc:env(), opts=[term()]}

Context for doclets.

doclet_gen() = #doclet_gen{sources=[string()], app=no_app | atom(), modules=[module()]}

Doclet command.

doclet_toc() = #doclet_toc{paths=[string()], indir=string()}

Doclet command.

Exports

`run(Doclet_gen::edoc_doclet:command(), Ctxt::edoc_doclet:context()) -> ok`

Main doclet entry point. See the file **edoc_doclet.hrl** for the data structures used for passing parameters.

Also see edoc:layout/2 for layout-related options, and edoc:get_doc/2 for options related to reading source files.

Options:

`{file_suffix, string()}`

Specifies the suffix used for output files. The default value is `".html"`.

`{hidden, boolean()}`

If the value is `true`, documentation of hidden modules and functions will also be included. The default value is `false`.

`{overview, edoc:filename()}`

Specifies the name of the overview-file. By default, this doclet looks for a file `"overview.edoc"` in the target directory.

`{private, boolean()}`

If the value is `true`, documentation of private modules and functions will also be included. The default value is `false`.

`{stylesheet, string()}`

Specifies the URI used for referencing the stylesheet. The default value is `"stylesheet.css"`. If an empty string is specified, no stylesheet reference will be generated.

`{stylesheet_file, edoc:filename()}`

Specifies the name of the stylesheet file. By default, this doclet uses the file `"stylesheet.css"` in the `priv` subdirectory of the EDoc installation directory. The named file will be copied to the target directory.

`{title, string()}`

Specifies the title of the overview-page.

See also

edoc

edoc_doclet_chunks

Erlang module

Doclet generating standalone **EEP-48** doc chunk files.

Section **Using the EDoc API** in the EDoc User's Guide shows an example of using this module.

Exports

`run(Doclet_gen::edoc_doclet:command(), Ctxt::edoc_doclet:context()) -> ok`

Main doclet entry point.

This doclet is tightly coupled with edoc_layout_chunks and should be used together with it.

The only option this doclet accepts is `dir`, which controls where the `chunks` subdirectory and the EEP-48 chunk files will be placed.

See also

edoc_layout_chunks, shell_docs(3)

edoc_extract

Erlang module

EDoc documentation extraction.

DATA TYPES

context() = module | footer | function | overview | single

filename() = file:filename()

proplist() = proplists:proplist()

Exports

`file(File, Context, Env, Opts) -> {ok, Tags} | {error, Reason}`

Types:

File = filename()
Context = context()
Env = edoc:env()
Opts = proplist()
Tags = [term()]
Reason = term()

Reads a text file and returns the list of tags in the file. Any lines of text before the first tag are ignored. `Env` is an environment created by edoc_lib:get_doc_env/3. Upon error, `Reason` is an atom returned from the call to file:read_file/1 or the atom 'invalid_unicode'.

See text/4 for options.

`header(File, Env, Opts) -> edoc:entry_data()`

Types:

File = filename()
Env = edoc:env()
Opts = proplist()

Similar to header/5, but reads the syntax tree and the comments from the specified file.

See also: header/4, edoc:read_comments/2, edoc:read_source/2.

`header(Forms, File, Env, Opts) -> edoc:entry_data()`

Types:

Forms = erl_syntax:forms()
File = filename()
Env = edoc:env()
Opts = proplist()

Extracts EDoc documentation from commented header file syntax trees. Similar to source/5, but ignores any documentation that occurs before a module declaration or a function definition. (Warning messages are printed if content may be ignored.) `Env` is assumed to already be set up with a suitable module context.

See also: erl_recomment(3), header/5.

`header(Forms, Comments, File, Env, Opts) -> edoc:entry_data()`

Types:

`Forms = erl_syntax:forms()`
`Comments = [edoc:comment()]`
`File = filename()`
`Env = edoc:env()`
`Opts = proplist()`

Similar to header/4, but first inserts the given comments in the syntax trees. The syntax trees must contain valid position information. (Cf. edoc:read_comments/2.)

See also: erl_recomment(3), header/3, header/4.

`source(File, Env, Opts) -> R`

Types:

`File = filename()`
`Env = edoc:env()`
`Opts = proplist()`
`R = {module(), edoc:edoc_module()} | {module(), edoc:edoc_module(), [edoc:entry()]}`

Like source/5, but reads the syntax tree and the comments from the specified file.

See also: source/4, edoc:read_comments/2, edoc:read_source/2.

`source(Forms, File, Env, Opts) -> R`

Types:

`Forms = erl_syntax:forms()`
`File = filename()`
`Env = edoc:env()`
`Opts = proplist()`
`R = {module(), edoc:edoc_module()} | {module(), edoc:edoc_module(), [edoc:entry()]}`

Extracts EDoc documentation from commented source code syntax trees. The given `Forms` must be a single syntax tree of type `form_list`, or a list of syntax trees representing "program forms" (cf. edoc:read_source/2. `Env` is an environment created by edoc_lib:get_doc_env/3. The `File` argument is used for error reporting and output file name generation only.

See edoc:get_doc/2 for descriptions of the `def`, `hidden`, `private`, and `todo` options.

See also: erl_recomment(3), source/5, edoc:read_comments/2, edoc:read_source/2.

`source(Forms, Comments, File, Env, Opts) -> R`

Types:

`Forms = erl_syntax:forms()`

```
Comments = [edoc:comment()]
File = filename()
Env = edoc:env()
Opts = proplist()
R = {module(), edoc:edoc_module()} | {module(), edoc:edoc_module(),
[edoc:entry()]}
```

Like source/4, but first inserts the given comments in the syntax trees. The syntax trees must contain valid position information. (Cf. edoc:read_comments/2.)

See also: erl_recomment(3), source/3, source/4, edoc:read_comments/2, edoc:read_source/2.

`text(Text, Context, Env, Opts) -> Tags`

Types:

```
Text = string()
Context = context()
Env = edoc:env()
Opts = proplist()
Tags = [term()]
```

Returns the list of tags in the text. Any lines of text before the first tag are ignored. `Env` is an environment created by edoc_lib:get_doc_env/3.

See source/4 for a description of the `def` option.

See also

edoc

edoc_layout

Erlang module

The standard HTML layout module for EDoc. See the edoc module for details on usage.

Exports

`module(Element, Options) -> term()`

The layout function.

Options to the standard layout:

`{index_columns, integer()}`

Specifies the number of column pairs used for the function index tables. The default value is 1.

`{pretty_printer, atom()}`

Specifies how types and specifications are pretty printed. If the value `erl_pp` is specified the Erlang pretty printer (the module `erl_pp`) will be used. The default is to do no pretty printing which implies that lines can be very long.

`{stylesheet, string()}`

Specifies the URI used for referencing the stylesheet. The default value is `"stylesheet.css"`. If an empty string is specified, no stylesheet reference will be generated.

`{sort_functions, boolean()}`

If `true`, the detailed function descriptions are listed by name, otherwise they are listed in the order of occurrence in the source file. The default value is `true`.

`{xml_export, Module::atom()}`

Specifies an xmerl callback module to be used for exporting the documentation. See xmerl:export_simple/3 for details.

See also: edoc:layout/2.

`overview(E, Options) -> term()`

`type(E) -> term()`

See also

edoc

edoc_layout_chunks

Erlang module

Convert EDoc module documentation to an **EEP-48** `docs_v1` chunk.

This layout is only expected to work with edoc_doclet_chunks. Section **Using the EDoc API** in the EDoc User's Guide shows an example of using this module.

This module breaks the convention stated in `edoc_doclet` to not rely on `edoc.hrl` in doclets and layouts. It uses `#entry{}` records directly to recover information that is not otherwise available to layouts.

DATA TYPES

beam_language() = atom()

doc() = #{doc_language() => doc_string()} | none | hidden

doc_language() = binary()

doc_string() = binary()

docs_v1() = #docs_v1{anno=erl_anno:anno(), beam_language=beam_language(), format=mime_type(), module_doc=doc(), metadata=metadata(), docs=[docs_v1_entry()]}

The Docs v1 chunk according to EEP 48.

docs_v1_entry() = {_KindNameArity::{atom(), atom(), arity()}, _Anno::erl_anno:anno(), _Signature::signature(), _Doc::doc(), _Metadata::metadata()}

A tuple equivalent to the `#docs_v1_entry{}` record, but with the record name field skipped.

metadata() = map()

mime_type() = binary()

signature() = [binary()]

Exports

`module(Doc::edoc:edoc_module(), Options::proplists:proplist()) -> binary()`

Convert EDoc module documentation to an EEP-48 style doc chunk.

See also

edoc_doclet_chunks, shell_docs(3)

edoc_lib

Erlang module

Utility functions for EDoc.

DATA TYPES

proplist() = proplists:proplist()

Exports

`get_doc_env(App, Modules, Options) -> edoc:env()`

Types:

`App = atom() | no_app`
`Modules = [module()]`
`Options = proplist()`

Creates an environment data structure used by parts of EDoc for generating references, etc. See edoc:run/2 for a description of the options `file_suffix`, `app_default` and `doc_path`.

See also: edoc:get_doc/3, edoc_extract:source/4.

`infer_module_app(Mod::module()) -> no_app | {app, atom()}`

Infer application containing the given module.

It's expected that modules which are not preloaded and don't match the `<app>/ebin/<mod>.beam` path pattern will NOT have an app name inferred properly. `no_app` is returned in such cases.

`write_file(Text, Dir, Name, Options) -> term()`

See also

edoc

edoc_run

Erlang module

Interface for calling EDoc from Erlang startup options.

The following is an example of typical usage in a Makefile:

```
    docs:
            erl -noshell -run edoc_run application "'$(APP_NAME)'" \
              '"."' '[{def,{vsn,"$(VSN)"}}]'
```

(note the single-quotes to avoid shell expansion, and the double-quotes enclosing the strings).

New feature in version 0.6.9: It is no longer necessary to write `-s init stop` last on the command line in order to make the execution terminate. The termination (signalling success or failure to the operating system) is now built into these functions.

DATA TYPES

args() = [string()]

Exports

`application(Args::args()) -> no_return()`

Calls edoc:application/3 with the corresponding arguments. The strings in the list are parsed as Erlang constant terms. The list can be either `[App]`, `[App, Options]` or `[App, Dir, Options]`. In the first case edoc:application/1 is called instead; in the second case, edoc:application/2 is called.

The function call never returns; instead, the emulator is automatically terminated when the call has completed, signalling success or failure to the operating system.

`file(Args::args()) -> no_return()`

This function is deprecated: This is part of the old interface to EDoc and is mainly kept for backwards compatibility. The preferred way of generating documentation is through one of the functions application/1 and files/1.

Calls edoc:file/2 with the corresponding arguments. The strings in the list are parsed as Erlang constant terms. The list can be either `[File]` or `[File, Options]`. In the first case, an empty list of options is passed to edoc:file/2.

The following is an example of typical usage in a Makefile:

```
    $(DOCDIR)/%.html:%.erl
            erl -noshell -run edoc_run file '"$<"' '[{dir,"$(DOCDIR)"}]' \
              -s init stop
```

The function call never returns; instead, the emulator is automatically terminated when the call has completed, signalling success or failure to the operating system.

`files(Args::args()) -> no_return()`

Calls edoc:files/2 with the corresponding arguments. The strings in the list are parsed as Erlang constant terms. The list can be either `[Files]` or `[Files, Options]`. In the first case, edoc:files/1 is called instead.

The function call never returns; instead, the emulator is automatically terminated when the call has completed, signalling success or failure to the operating system.

See also

edoc