GNATdoc User Guide

Release 2018

AdaCore

Dec 11, 2018

CONTENTS

1 Introduction **3**
- 1.1 Installation . . . 3
- 1.2 Launching GNATdoc . . . 3
- 1.3 Command line interface . . . 3

2 Annotating source files **7**
- 2.1 Documenting packages . . . 7
- 2.2 Documenting enumeration types . . . 7
- 2.3 Documenting record types . . . 8
- 2.4 Documenting subprograms . . . 9
- 2.5 Text markup . . . 10
- 2.6 Excluding entities . . . 11
- 2.7 Groups of packages . . . 11
- 2.8 Adding images . . . 11

3 Configuration **13**
- 3.1 Output directory . . . 13
- 3.2 Ignore subprojects . . . 13
- 3.3 Images directory . . . 13
- 3.4 Documentation pattern . . . 13
- 3.5 Custom tags definition . . . 14
- 3.6 HTML output customization . . . 14

4 Handling of custom tags **15**
- 4.1 HTML backend specific information . . . 15
- 4.2 Python API . . . 16

5 Search and indices **19**

Index **21**

Contents:

CHAPTER

ONE

INTRODUCTION

GNATdoc is a documentation tool for Ada which processes source files, extracts documentation directly from the sources, and generates annotated HTML files. It is based on the source cross-reference information (e.g. generated by GNAT for Ada files). This means that you should ensure that cross-reference information has been generated before generating the documentation. It also relies on standard comments that it extracts from the source code. The engine in charge of extracting them coupled with the cross-reference engine gives GNATdoc all the flexibility needed to generate accurate documentation, and report errors in case of wrong documentation.

1.1 Installation

GNATdoc is shipped as part of the GPS package. To install it, simply launch the GPS installer.

After the installation place `<gps_installation_prefix>/bin/` in your PATH environment variable.

1.2 Launching GNATdoc

GNATdoc is based on the source cross-reference information (e.g. generated by GNAT for Ada files). This means that you should ensure that cross-reference information has been generated before generating the documentation. For this purpose, before launching the tool compile your project.

GNATdoc requires your project hierarchy to be described via GNAT project files (.gpr).

To launch GNATdoc, execute:

```
gnatdoc -P<your_project>
```

where `<your_project>` is the .gpr file at the root of your project hierarchy (your root project).

GNATdoc generates an HTML report in the `gnatdoc` directory of the object directory of the main project.

1.3 Command line interface

A brief description of the supported switches is available through the switch –help:

```
$ gnatdoc --help
GNATdoc command line interface
Usage: gnatdoc [switches] [arguments]

-P, --project=ARG            Load the given project (mandatory)
-X ARG                       Specify an external reference in the project
-R, --regexp=ARG             Regular expression to select documentation comments
--preserve-source-formatting Preserve formatting of comments
```

(continues on next page)

(continued from previous page)

```
-e, --encoding=ARG           The character encoding used for source and ALI files
-b                           Process bodies to complete the spec documentation
-d                           Document bodies
--ignore-files=ARG           List of files ignored by GNATdoc
-l                           Leading documentation
--no-subprojects             Do not process subprojects
-p                           Process private part of packages
-q                           Be quiet/terse
--single-file=ARG            Single file processed by GNATdoc
-w                           Enable warnings for missing documentation
--enable-build               Rebuild the project before processing it
--version                    Shows GNATdoc's version
--output=ARG                 Format of generated documentation
--custom-tags-definition=ARG Load custom tag definition from the file
--symlinks                   Take additional time to resolve symbolic links
```

Project (-P)

Specify the path name of the main project file. The space between -P and the project file name is optional.

External reference (-X)

Specify an external reference in the project.

Preserve formatting of comments (–preserve-source-formatting)

When this switch is used, the line breaks and spaces present in the source comments will be preserved in the HTML output.

Regular expression (-R)

Regular expression used to select comments containing documentation. If not specified then all the comments found in the specification of the compilation units of the project are used to document the project; if specified then only those comments matching the specified regular expression are used to generate the documentation. The leading comment delimiters "–" are stripped before applying the regular expression.

For example, the regular expression "^-" can be used to select the documentation of the following subprogram and skip the internal comment:

```
function Set_Alarm
  (Message : String;
   Minutes : Natural) return Boolean;
--- Display a message after the given time.
--  TODO: what is the unit for Minutes?
--- @return True iff the alarm was successfully registered
```

Process bodies to complete the spec documentation(-b)

By default GNATdoc does not process the body of packages. This switch enables looking at subprograms in package bodies, as a fallback for finding documentation. When this switch is provided, GNATdoc first looks for the documentation in the package specification; if no documentation is found in the spec and then searches for documentation in the body of the subprogram.

Document bodies (-d)

When this switch is passed, GNATdoc processes bodies and extracts documentation for library-level entities. In the HTML output, GNATdoc emits separate pages for the documentaion extracted from bodies. This switch is incompatible with the -b switch.

Ignore files (–ignore-files)

This switch allows to specify a list of source files ignored by GNATdoc. The names of the files can be separated by spaces or commas. For example:

```
gnatdoc -P default.gpr --ignore-files="file_1.ads,file_2.ads"
gnatdoc -P default.gpr --ignore-files="file_1.ads file_2.ads"
```

Leading documentation (-l)

By default GNATdoc extracts the documentation by first looking at the comments located after the entity declaration and fallback to the comments located before the entity if not found. This switch reverts such behavior, thus extracting first leading comments.

Do not process subprojects (–no-subprojects)

By default GNATdoc generates the documentation of all the files of a root project and its subprojects. This switch restricts the generation of documentation to the root project.

Process private part of packages (-p)

By default GNATdoc does not generate documentation for declarations found in the private part of packages. This switch enables the generation of such documentation.

Be quiet / terse (-q)

Do not display anything except errors.

Single file (–single-files)

By default GNATdoc generates the documentation of all the files of a project. This switch restricts the generation of documentation to the specified file.

Enable warnings for missing documentation (-w)

Emit warnings for fields, parameters or subprograms which do not have documentation.

Rebuild the project before processing it (–enable-build)

GNATdoc will launch gprbuild on the project before building the documentation.

Output format (–output)

At current stage GNATdoc generates HTML files (*–output=html*).

Custom tags definition (–custom-tags-definition)

Load custom tag definitions from the given file. This switch overrides the value of the attribute Custom_Tag_Definition in the Documentation package of the project file.

Take additional time to resolve symbolic links (–symlinks)

Should be specified if your projet uses symbolic links for files. This will ensure that the links are fully resolved as stored in the database, and thus that when a file is visible through different links, the information is appropriately coalesced in the database for that file.

CHAPTER

TWO

ANNOTATING SOURCE FILES

GNATdoc extracts documentation directly from the comments present in source files for your project. Special tags present in the comments are interpreted by GNATdoc.

2.1 Documenting packages

The documentation attached to each package is the block of comment directly preceding the package declaration.

The following tags are supported in package comments:

@summary

a summary of the package

@description

a detailed description of the package

For example:

```
--  @summary
--  Drawing routines.
--
--  @description
--  This package provides routines for drawing basic shapes and Bézier curves.
--
package Drawing is
```

2.2 Documenting enumeration types

The documentation attached to each enumeration type is the block of comment directly following the record type declaration, or directly preceding it if the option *-l* was specified.

The following tag is supported when annotating enumeration literals:

@value

document an enumeration literal, with the following syntax:

@value <*enumeration_literal*> <*description*>

where:

<*enumeration_literal*>

is the value of the enumeration literal as it appears in the enumeration type declaration.

<description>

the documentation for the enumeration literal; all following text is considered for inclusion, until a blank comment line or another tag is encountered.

For example:

```
--  Colors supported by this drawing application
--  @value Black The black color is the default color of the pen
--  @value White The white color is the default color of the background
--  @value Green The green color is the default color of the border
type Colors is (Black, White, Green);
```

Enumeration literals can also be documented in line, with the documentation for each literal directly following its declaration (or directly preceding the component declaration, if the option *-l* was specified). In this case, the tag *@value* is not required:

```
--  Colors supported by this drawing application
type Colors is (
  Black,
  -- The black color is the default color of the pen
  White,
  -- The white color is the default color of the background
  Green
  -- The green color is the default color of the border
);
```

As shown above, a combined approach of documentation is also supported (see that the general description of the enumeration type *Colors* is located before its declaration and the documentation of its literals is located after their declaration).

2.3 Documenting record types

The documentation attached to each record type is the block of comment directly following the record type declaration, or directly preceding it if the option *-l* was specified.

The following tags are supported when annotating subprograms:

@field

document a record component, with the following syntax:

@field <component_name> <description>

where:

<component_name>

is the name of the component as it appears in the subprogram.

<description>

the documentation for the component; all following text is considered for inclusion, until a blank comment line or another tag is encountered.

For example:

```
--  A point representing a location in integer precision.
--  @field X Horizontal coordinate
--  @field Y Vertical coordinate
```

(continues on next page)

(continued from previous page)

```
type Point is
 record
    X : Integer;
    Y : Integer;
 end record;
```

Record components can also be documented in line, with the documentation for each component directly following its declaration (or directly preceding the component declaration, if the option *-l* was specified). In this case, the tag *@field* is not required:

```
--  A point representing a location in integer precision.
type Point is
 record
    X : Integer;
    --  Horizontal coordinate
    Y : Integer;
    --  Vertical coordinate
 end record;
```

As shown above, a combined approach of documentation is also supported (see that the general description of the record type *Point* is located before its declaration and the documentation of its components *X* and *Y* is located after their declaration).

2.4 Documenting subprograms

The documentation attached to each subprogram is the block of comment directly following the subprogram declaration, or directly preceding it if the option *-l* was specified.

The following tags are supported when annotating subprograms:

@param

document a subprogram parameter, with the following syntax:

@param <param_name> <description>

where:

<param_name>

is the name of the parameter as it appears in the subprogram.

<description>

the documentation for the parameter; all following text is considered for inclusion, until a blank comment line or another tag is encountered.

@return

document the return type of a function, with the following syntax:

@return <description>

where:

<description>

is the documentation for the return value; all following text is considered for inclusion, until a blank comment line or another tag is encountered.

@exception

document an exception, with the following syntax:

@*exception* <*exception_name*> <*description*>

where:

<*exception*>

is the name of the exception potentially raised by the subprogram

<*description*>

is the documentation for this exception; all following text is considered for inclusion, until a blank comment line or another tag is encountered.

For example:

```
function Set_Alarm
  (Message : String;
   Minutes : Natural) return Boolean;
--  Display a message after the given time.
--  @param Message The text to display
--  @param Minutes The number of minutes to wait
--  @exception System.Assertions.Assert_Failure raised
--     if Minutes = 0 or Minutes > 300
--  @return True iff the alarm was successfully registered
```

The parameters can also be documented in line, with the documentation for each parameter directly following the parameter type declaration (or directly preceding the parameter declaration, if the option *-l* was specified). In this case, the tag @*param* is not required:

```
function Set_Alarm
  (Message : String;
   --  The text to display

   Minutes : Natural
   --  The number of minutes to wait
  ) return Boolean;
--  Display a message after the given time.
--  @exception System.Assertions.Assert_Failure raised
--     if Minutes = 0 or Minutes > 300
--  @return True iff the alarm was successfully registered
```

2.5 Text markup

GNATdoc recognizes several markup constructs inside description text, that can be used to better control the format of the generated documentation.

All markup constructs are based on paragraphs. A paragraph is one or more consecutive lines of text, separated from the flow of comments by one blank line. All lines in a paragraph should have the same indentation. Sequential paragraphs with the same indentation level are processed as a list of paragraphs.

2.5.1 Bulled lists

Bulled lists start by a paragraph that begins with the characters '-' or '*'. List items of the same list must have the '-' or '*' character at same indentation. When the text of a list item occupies more than one line, it should be aligned with the first character on the first line of the list item. List items can have more than one paragraph, in which case all paragraphs should use the same indentation as the first paragraph:

```
--  - This is the first bullet list item.  The blank line above the
--    first list item is required; blank lines between list items
--    (such as below this paragraph) are optional.
--
--  - This is the first paragraph in the second item in the list.
--
--    This is the second paragraph in the second item in the list.
--    The blank line above this paragraph is required.  The left edge
--    of this paragraph lines up with the paragraph above, both
--    indented relative to the bullet.
--
--    - This is a sublist.  The bullet lines up with the left edge of
--      the text blocks above.  A sublist is a new list so requires a
--      blank line above and below.
--
--  - This is the third item of the main list.
--
--  This paragraph is not part of the list.
```

2.5.2 Code blocks

Code blocks can be used to include preformatted text into the documentation. A code block should have an indentation of three or more spaces. Contrary to other constructs, a code block doesn't end wth an empty line:

```
--    with Ada.Text_IO;
--
--    procedure Hello_World is
--    begin
--       Ada.Text_IO.Put_Line ("Hello, world!");
--    emd Hello_World;
```

2.6 Excluding entities

The tag *@private* notifies GNATdoc that no documentation must be generated on a given entity. For example:

```
type Calculator is tagged ...
procedure Add (Obj : Calculator; Value : Natural);
--  Addition of a value to the previus result
--  @param Obj The actual calculator
--  @param Value The added value
procedure Dump_State (Obj : Calculator);
--  @private No information is generated in the output about this
--  primitive because it is internally used for debugging.
```

2.7 Groups of packages

The tag *@group* is used by GNATdoc to generate an index of packages in the project grouped by categories.

2.8 Adding images

Documentation for packages and subprograms may include images.

This is done via the attribute:

@image

where the first parameter is the name of an image file. This file is expected in the images directory, as specified in the project file: see section Images directory below.

CHAPTER

THREE

CONFIGURATION

3.1 Output directory

The documentation is generated by default into a directory called `gnatdoc`, created under the object directory of the root project. This behavior can be modified by specifying the attribute Documentation_Dir in the package Documentation of your root project:

```
project Default is
   package Documentation is
      for Documentation_Dir use "html";
   end Documentation;
end P;
```

3.2 Ignore subprojects

By default GNATdoc recursively processes all the projects on which your root project depends. This behavior can be modified by specifying the attribute Ignored_Subprojects in the package Documentation of your root project:

```
with "prj_1";
with "prj_2";
with "prj_3";
project Default is
   package Documentation is
      for Ignored_Subprojects use ("prj_1", "prj_3");
   end Documentation;
end Default;
```

3.3 Images directory

The directory containing images is specified by the string attribute Image_Dir of the Documentation package:

```
package Documentation is
   for Image_Dir use "image_files";
end Documentation;
```

3.4 Documentation pattern

The regular expression for recognizing doc comments can be specified via the string attribute Doc_Pattern of the Documentation package:

```
package Documentation is
   for Doc_Pattern use "^<";
   --  This considers comments beginning with "--<" to be documentation
end Documentation;
```

If this attribute is not specified, all comments are considered to be doc.

This has the same semantics as the *-R* command-line switch. The command-line switch has precedence over the project attribute.

3.5 Custom tags definition

Set of tags processed by GNATdoc can be extended by providing custom tags handlers. String attribute Custom_Tags_Definition allows to specify Python file that contains implementation of custom tags handlers:

```
package Documentation is
   for Custom_Tags_Definition use "my_tags.py";
   --  GNATdoc loads my_tags.py file on startup to process custom tags
end Documentation;
```

If this attribute is not specified, there is no custom tags are processed.

It is possible to use the *–custom-tags-definition* command-line switch to provide file name of custom tags handlers file.

3.6 HTML output customization

GNATdoc uses a set of static resources and templates files to control the final rendering. Modifying these static resources and templates you can control the rendering of the generated documentation. The files used for generating the documentation can be found under `<install_dir>/share/gps/gnatdoc/html`. If you need a different layout from the proposed one, you can override those files and provides new files. The directory for user defined static resources and templates can be specified via the string attribute *HTTP_Custom_Dir* of the *Documentation* package in your project file:

```
package Documentation is
   for HTML_Custom_Dir use "docs/gnatdoc_html";
end Documentation;
```

All files in `static` subdirectory will be copied to result documentation directory. This can be used to provide additional files like CSS, images, etc.

Files in `templates` subdirectory are used as templates when documentation is generated. You can put modified versions of default files in this directory with same name as original file. Adding new files has no effect on generated documentation.

CHAPTER

FOUR

HANDLING OF CUSTOM TAGS

It is possible to extend the set of supported tags by providing custom tag handlers written in Python. Assuming you have written these in a file called *mytags.py*, you should add the switch *–custom-tags-definition=mytags.py* to the gnatdoc command line.

In your file, you will need to declare a Python class for each custom tag you wish to define. This class should be inherited from the *GPS.InlineTagHandler* class. Its constructor must call the constructor of the inherited class and pass it the name of the tag you want to handle. Each tag handler class must be registered by calling *GPS.register_tag_handler()*.

Custom tag handling class must implement the function `to_markup()`. Inside this function, you should use the provided *writer* to generate documentation. Most methods of the class *GPS.MarkupGenerator* have 'start'/'end' pairs. These functions generate elements that can be nested. The class for custom tag handling is responsible for generating one call to the 'end' function for each call 'start' function.

Here is an example of a custom tag handler that defines and registers a handler for the 'hello' tag, and outputs the text 'Hello, <parameter>!' to the generated documentation:

```
class HelloTagHandler(GPS.InlineTagHandler):
    def __init__(self):
        super(HelloTagHandler, self).__init__('hello')

    def has_parameter(self):
        return True

    def to_markup(self, writer, parameter):
        writer.text('Hello, %s!' % parameter)

GPS.register_tag_handler(HelloTagHandler())
```

4.1 HTML backend specific information

HTML backend allows to specify CSS class to be applied to generated elements. All functions of *GPS.MarkupGenerator* that have 'attributes' parameters use value of 'class' key to set CSS class name of generated element. See *HTML output customization* for more information how to provide your own CSS files and to modify default templates to link CSS files with generated documentation.

text method of *GPS.MarkupGenerator* accepts 'href' key to generate cross reference to URL specified by its value.

4.2 Python API

4.2.1 GPS

GPS.**register_tag_handler**(*handler*)
Registers instance of custom tag handler.

4.2.2 GPS.InlineTagHandler

class GPS.**InlineTagHandler**(*name*)
This class is intended to be used as base class for custom tag handlers of inline tags.

__init__(*name*)
Initialize base class

Parameters **name** (*string*) – Name of custom tag

has_parameter()
Should returns True when custom tag has parameter. Default implementation returns False.

Returns Should or should not additional parameter be parsed and passes to custom tag handler.

to_markup(*writer*, *parameter*)
Do custom tag processing and use writer to generate documentation.

Parameters

- **writer** (*GPS.MarkupGenerator*) – Writer to be used to generate documentation.
- **parameter** (*string*) – Additional parameter of custom tag provided in source code.

4.2.3 GPS.MarkupGenerator

class GPS.**MarkupGenerator**
Used by custom tag handler to generate structured output.

start_paragraph()
Opens paragraph in generated documentation. *GPS.MarkupGenerator.end_paragraph()* must be called to close paragraph.

Paragraphs can be nested and can contain lists and text too.

Parameters **attributes** (*dictionary*) – Additional attributes to be passed to backend.

end_paragraph()
Closes paragraph in generated documentation.

start_list()
Opens list in generated documentation. func:*GPS.MarkupGenerator.end_list* must be called to close list.

Lists can contains list items only.

Parameters **attributes** (*dictionary*) – Additional attributes to be passed to backend.

end_list()
Closes list in generated documentation.

start_list_item()
Opens item of the list in generated documentation. func:*GPS.MarkupGenerator.end_list_item* must be called to close list item.

List item can contain paragraph, lists and text.

Parameters **attributes** (*dictionary*) – Additional attributes to be passed to backend.

end_list_item()
Closes list item in generated documentation.

text(*attributes*)
Outputs test to generated documentation.

Parameters

- **text** (*string*) – Text to be output in generated documentation.
- **attributes** (*dict*) – Additional attributes to be passed to backend.

html()
Outputs HTML markup to generated documentation. This markup is processed by browser to display its content.

Note, this function is intended to be used for HTML backend only.

Parameters **html** (*string*) – HTML markup to be output in generated documentation.

generate_after_paragraph()
Switch MarkupGenerator to generate output after processing of paragraph where custom tag is used.

generate_inline()
Switch MarkupGenerator to generate output in place where custom tag is used.

CHAPTER

FIVE

SEARCH AND INDICES

- search
- genindex

INDEX

Symbols

__init__() (GPS.InlineTagHandler method), 16

E

end_list() (GPS.MarkupGenerator method), 16
end_list_item() (GPS.MarkupGenerator method), 16
end_paragraph() (GPS.MarkupGenerator method), 16

G

generate_after_paragraph() (GPS.MarkupGenerator method), 17
generate_inline() (GPS.MarkupGenerator method), 17

H

has_parameter() (GPS.InlineTagHandler method), 16
html() (GPS.MarkupGenerator method), 17

I

InlineTagHandler (class in GPS), 16

M

MarkupGenerator (class in GPS), 16

R

register_tag_handler() (in module GPS), 16

S

start_list() (GPS.MarkupGenerator method), 16
start_list_item() (GPS.MarkupGenerator method), 16
start_paragraph() (GPS.MarkupGenerator method), 16

T

text() (GPS.MarkupGenerator method), 17
to_markup() (GPS.InlineTagHandler method), 16