GNATdoc User Guide

Release 28.0w

AdaCore

Nov 26, 2026

CONTENTS

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

2 Annotating source files **5**
- 2.1 Documenting packages . . . 5
- 2.2 Documenting enumeration types . . . 5
- 2.3 Documenting record types . . . 6
- 2.4 Documenting subprograms . . . 8
- 2.5 Text markup . . . 9
- 2.6 Excluding entities . . . 10

3 Configuration **11**
- 3.1 Documentation pattern . . . 11
- 3.2 Excluded project files . . . 11
- 3.3 Output directory . . . 12
- 3.4 Output customization . . . 12
- 3.5 Images directories . . . 12

4 Backends **13**
- 4.1 HTML . . . 13
- 4.2 ODF . . . 14
- 4.3 RST . . . 14
- 4.4 XML . . . 14

5 Search and indices **15**

Contents:

CHAPTER

ONE

INTRODUCTION

GNATdoc is a documentation tool for Ada which processes source files, extracts documentation from the sources, and generates either annotated HTML files or Restructured Text (.rst) files.

It relies on documentation comments that it extracts from the source code. The engine in charge of extracting these comments, coupled with a cross-reference engine, gives GNATdoc all the flexibility needed to generate accurate documentation, and report errors in cases of missing documentation.

1.1 Installation

GNATdoc is shipped as part of the GNAT Studio package. To install it, launch the GNAT Studio installer.

After the installation, place <gnatstudio_installation_prefix>/bin/ in your PATH environment variable.

1.2 Launching GNATdoc

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 (referred to here as the root project).

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

1.3 Command line interface

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

```
$ gnatdoc --help
Usage: gnatdoc [options] project_file

Options:
  --backend <html|odf|rst|xml>
                            Backend to use to generate output
  --generate <public|private|body>
                            Part of code to generate documentation
  --style <leading|trailing|gnat>
```

(continues on next page)

(continued from previous page)

```
                              Use given style of documentation
  -O, --output-dir <output_dir>
                              Output directory for generated documentation
  -P, --project <project_file>
                              Project file to process
  --warnings                  Report warnings for undocumented entities
  -v, --verbose               Enable verbose output
  -X <name=value>             Specify an external reference for scenario
                                variables
  -h, --help                  Display help information

Arguments:
  project_file                Project file to process
```

Here is the list of supported switches:

- *-P, –project=<file>*: specify the project file

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

- *-X <NAME>=<VALUE>*: Project external references

  Specify an external reference in the project. This can be used multiple times.

- *–backend=<name>*: select the output format

  GNATdoc generates HTML files (*–backend=html*), OpenDocument Text document (*–backend-odf*), ReST files (*–backend=rst*), or XML file (*–backend=xml*). The default is HTML.

- *–generate=<part>*: Select the of entities to be included into the documentation

  public
  : Entities declared in the public part of the package specification. This is the default.

  private
  : Entities declared in the package specifications, in both public and private parts.

  body
  : All entities declared in the both package specifications and at library level in package bodies.

- *-O, –output-dir=<path>*: Output directory

  Directory to output generated documentation. This option overrides the value of the Documentation'Output_Dir attribute defined in the project file.

- *–style=<style>*: Documentation comments style

  leading
  : Documentation is extracted from the comment before the entity declaration.

  trailing
  : Documentation is extracted from the comment after the entity declaration.

  gnat
  : Documentation is extracted from the comment after the entity declaration and additional features of the GNAT style are enabled.

- *-v*, *–verbose*: Enable verbose output

- *–warnings*: Enable warnings for missing documentation

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

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.

Documentation is extracted from the comment blocks at different locations, depending of the entity kind and documentation style. The description below provides a list of all supported blocks for each entity kind.

2.1 Documenting packages

The documentation attached to each package is the block of comment from one of the following locations:

- directly preceeding context clauses
- directly preceeding the package declaration
- directly following 'is' keyword of the package declaration
- **directly following the block of 'pragma' directives and 'use' context** clauses at the beginning of the package declaration

For example:

```
--  Copyright (C) COPYRIGHT HOLDER

--  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 preceding, or directly following, the enumeration type declaration; it should have the same indentation as the enumeration type declaration, to distinguish it from the multiline in line documentation of the enumeration literal itself.

The following tag is supported when annotating enumeration literals:

@enum

document an enumeration literal, with the following syntax:

@enum <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 another tag or end of comment block is encountered.

For example:

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

Enumeration literals can also be documented inline, with the documentation for each literal directly following its declaration (or directly preceding the component declaration, if the *leading* style is used). In this case, the tag *@enum* 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 preceeding or directly following the record type declaration and has same indentation as the record type declaration (to distinguish it from the multiline in line documentation of the subprograms' parameter or subprogram's return value).

The following tags are supported when annotating record types:

@comp

document a record component, with the following syntax:

@comp <component_name> <description>

where:

<component_name>

is the name of the component as it appears in the record type.

<description>

the documentation for the component; all following text is considered for inclusion, until an another tag is encountered.

@disc

document a discriminant of a record/task/protected type, with the following syntax:

@disc <discriminant_name> <description>

where:

<discriminant_name>

is the name of the discriminant as it appears in the type.

<description>

the documentation for the discriminant; all following text is considered for inclusion, until an another tag is encountered.

@field

document a record component or discriminant, with the following syntax:

@field <name> <description>

where:

<name>

is the name of the component or discriminant as it appears in the record.

<description>

the documentation for the component or discriminant; all following text is considered for inclusion, until an another tag is encountered.

@member

alias of *@field* tag

For example:

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

Record components can also be documented inline, with the documentation for each component directly following its declaration (or directly preceding the component declaration, if the *leading* style of the documentation is specified). In this case, the *@field* tag 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 *leading* style 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 an another tag or end of the comment block 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 an another tag or end of the comment block 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 an another tag or end of the comment block 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 inline, with the documentation for each parameter directly following the parameter type declaration (or directly preceding the parameter declaration, if the *leading* style of the documentation was specified). In this case, the *@param* tag is not required:

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

2.5 Text markup

GNATdoc recognizes several markup constructs inside the description text, which can be used to better control the format of the generated documentation. GNATdoc's markup syntax is based on the MarkDown syntax (see Common Mark for a detailed description).

GNATdoc supports the following MarkDown features:

- paragraphs
- lists and list items
- indented code blocks (code blocks are indented by three or more spaces)
- fenced code blocks (code blocks are delimited by a pair of code fences indented by no more than two spaces, a code fence is a sequence of at least three backticks or tildes)

  The following restrictions apply:

  – The info string of the opening code fence, which is used by MarkDown to specify the language of the code, is ignored. Code blocks are always processed as Ada code.

  – Indentation rules of the CommonMark are shifted by one space, because an indented code block of GNATdoc starts at three spaces instead of four. Thus, a code fence is indented by no more than two spaces, and a code fence indented by three or more spaces starts an indented code block instead. Both fences and the text between them are the content of that indented code block, which makes it possible to document MarkDown markup itself.

    As a consequence, a code fence indented by three or more spaces doesn't interrupt a paragraph, because an indented code block never does.

    Note, indentation is computed after removal of the common indentation of the description text, and inside a list item it is relative to the content of that item, thus it may differ from the indentation of the comment in the source code.

  – When the closing code fence is absent, the code block is ended by the end of the enclosing section.

  – Tags are extracted before processing of the MarkDown markup, so a line that starts a new section (`@param`, `@return`, and so on) is processed as a tag even when it is inside a code block. This applies to indented code blocks too.

- emphasis/strong emphasis inlines
- code span inline

- images

2.6 Excluding entities

The @*exclude* tag indicates that no documentation should 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);
--  @exclude No information is generated in the output about this
--  primitive because it is internally used for debugging.
```

Note: specifing the @*exclude* tag for the packages removes the package and all its child packages from the generated documentation.

Note: the @*private* tag is now an alias of the @*exclude* tag and is no longer recommended.

The @*exclude-value* tag suppresses the generation of the documentation for a given expression value. For example:

```
C : constant Integer := 5;
--  The constant and its description will be included into the documentation, while
--  the actual value will not.
--  @exclude-value
```

CHAPTER

THREE

CONFIGURATION

GNATdoc can be configured via some attributes in the *Documentation* package of the root project file.

3.1 Documentation pattern

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

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

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

3.2 Excluded project files

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

```
package Documentation is
   for Excluded_Project_Files use
     ("vss_config.gpr",
      "vss_gnat.gpr");
end Documentation;
```

This list may include any project files directly or indirectly used by the root project.

3.3 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 *Output_Dir* in the *Documentation* package:

```
package Documentation is
   for Output_Dir ("html") use "html";
end Documentation;
```

3.4 Output customization

The GNATdoc backends can use a set of static resources and templates files to control the final rendering. By 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/gnatdoc/<backend>`. If you need a different layout from the proposed one, you can override those files and provides your own set of files. The directory for user defined static resources and templates can be specified via the string attribute *Resources_Dir* of the *Documentation* package in the project file:

```
package Documentation is
   for Resources_Dir ("html") use "docs/gnatdoc_html";
end Documentation;
```

Each backend has own conventions on the layout of the resources directory: see the documentation of the particular backend.

3.5 Images directories

List of directories to lookup for image files referenced in the documentation can be defined with *Image_Dirs* attribute of the root project file:

```
package Documentation is
   for Image_Dirs ("odf") use ("images");
end Documentation;
```

By default paths are resolved relatively to the project file's root directory.

CHAPTER

FOUR

BACKENDS

4.1 HTML

The HTML backend is default backend of the GNATdoc.

4.1.1 Command line options

The HTML backend can generate documentation structure in two formats.

- Group all entities by compilation units (the default)
- Group tagged types, its dispatching subprograms and subprograms that can be used by prefixed notation separately, and group other entities by compilation units. This format can be renabled by adding the *:oop* qualifier at the end of the *–backend* command line switch, as in:

  gnatdoc –backend=html:oop -P project.gpr

4.1.2 Layout of the resources directory

<resources_dir>/static
: The content of this directory is copied into the directory of the generated documentation. This can be used to provide additional files like CSS, images, etc.

<resources_dir>/templates
: Directory of the XHTML templates that are used to generate documentation. It contains the files described below.

<resources_dir>/templates/index.xhtml
: The template for the home page of the generated documentation.

<resources_dir>/templates/unit.xhtml
: The template for the documentation of the compilation unit.

<resources_dir>/templates/class.xhtml
: The template for the documentation of the class (when generation of the OOP style documentation is specified)

4.2 ODF

The ODF backend can be used to generate documentation in OpenDocument Format (ODF) format. The generated file can be opened by many office suites (Microsoft Word, LibreOffice Writer, etc) to apply different styles and print the documentation.

Output can be customized by providing a custom *<resources_dir>/template/documentation.fodt* file.

4.3 RST

The RST backend can be used to generate a set of *.rst* files to be used to generate documentation with *sphinx* configured with *ada-domain*.

4.4 XML

The XML backend can be used to generate documentation in XML format for processing with other tools.

CHAPTER

FIVE

SEARCH AND INDICES

- search
- genindex

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.