
An Odin domain and Odin API documentation

Design, rationale, and review

01 / SOURCE

Write with meaning.

02 / STRUCTURE

Make relations clear.

03 / PUBLICATION

Read on screen and paper.

Contents

Abstract	1
Motivation and scope	1
Specification	1
Rationale and alternatives	4
Security and operational considerations	4
Backwards compatibility	4
Acceptance criteria and implementation status	5
Open questions	5
Review history	5
References	5

Number: 0005
Title: An Odin domain and Odin API documentation
State: committed
Type: Standards Track
Authors: Vikrant Rathore
Created: 2026-09-30
Updated: 2026-10-11
Discussion: No external discussion URL assigned
Labels: architecture, build

Abstract

Guidedog documents Odin APIs as Sphinx documents Python ones. An `odin` domain, always on, describes packages, procedures, procedure groups, structs, unions, enums, bit sets, bit fields, other types, constants, variables, foreign imports, fields, and enumerators, with signatures written and shown as Odin writes declarations; roles link to them; each object has an id, an index entry, a table of contents entry, and an `objects.inv` entry. Generating directives (`odin:autopackage` and five others) and API pages (`odin_autoapi_dirs`, as `sphinx-autoapi`'s `autoapi_dirs`) write that domain's reStructuredText from Odin source, parsed with Odin's own `core:odin/parser`. The compiler never runs. Guidedog's own reference, `docs/api`, is built this way.

Motivation and scope

Guidedog is written in Odin, and Odin has no documentation tool that produces a cross-referenced site and book with prose around the reference. Sphinx has no Odin domain. The Python pieces Guidedog already has (the Python domain, `sphinx.ext.autodoc` from static analysis, the project's object tables and inventories) give the design; this record applies it to Odin.

In scope: the domain in the Sphinx reader (inside the allocation boundary of GDS 0003); `lib/odindoc`, a host library that analyses Odin packages and writes the domain's text; the project's generating directives, API pages, cross-references, inventory, and incremental rebuilds. Out of scope: type checking, evaluating constants, and running anything.

Specification

The domain

`odin:package` sets the current package, as `py:module` sets the module, with `:synopsis:`, `:platform:`, `:deprecated:`, `:no-index:`, and `:imports:`, the import aliases the package's signatures use (`gd=core rst=readers/rst`). A package is named by its import path: `core:fmt` for a collection, or its folder below a project root (`readers/sphinx`). `odin:currentpackage` changes the context without a target.

Directive	Signature, as Odin writes it
<code>odin:procedure, odin:proc</code>	<code>[@(attrs)] name :: [#force_inline]proc["c"](params) [-> results][tags][where ...] or the short form name(params) -> results</code>
<code>odin:progroup</code>	<code>name :: proc{a, b};</code> each member links to its procedure
<code>odin:struct, odin:union</code>	<code>Name :: struct(\$T: typeid) #packed, Name :: union #no_nil {A, B}</code>
<code>odin:enum, odin:bitset, odin:bitfield</code>	<code>Name :: enum u8, Name :: bit_set[E; u8], Name :: bit_field u32</code>
<code>odin:type</code>	distinct types and aliases: <code>Handle :: distinct uintptr</code>
<code>odin:field, odin:enumerator</code>	inside a type: <code>name: T</code> with a tag or a bit size; <code>Name = 3</code>
<code>odin:const, odin:var</code>	<code>NAME :: 64, NAME : int : 64; name: T = value</code>
<code>odin:foreign</code>	a foreign import: <code>libc "system:c"</code>

Every object takes `:no-index:`, `:no-index-entry:`, `:no-contents-entry:`, `:no-typesetting:`, and the domain's own `:package:`, `:private:`, `:deprecated:`, `:availability:` (a first paragraph "Availability: Windows, Linux."), and `:foreign:`. Members nested in a type are named `Type.member`. Later signature lines of one directive are the same object on other targets: they get no id and no entry.

Signatures become `Desc_*` nodes: attributes and keywords as `Desc_Annotation`, the package's short name as `Desc_Addname` (with `add_module_names`), the name, parameters as a `Desc_Parameterlist`, named results as a second one after the text `" -> "`. Names in type positions become `Xref{odin:type}` with the context attributes `odin:package` and `odin:type`; an alias of `:imports:` is followed to its package in the reader. Built-in types, keywords, polymorphic names (`$T` and `T` after it, including a nesting type's), array lengths, enumeration names, defaults, constants, tags, and where clauses stay text, so nothing that cannot be an object produces a broken link, the way the Python domain leaves built-ins to its builtin resolver.

Ids, references, and the inventory

An object's id is Sphinx's `make_id("odin-" + package + "." + name):` `odin-core-fmt.println`, `odin-readers-sphinx.Config.docname`; a package's is `odin-package-core-fmt`. `make_id` keeps case, dots, and underscores, as for Python ids, and turns the import path's `:` and `/` into hyphens: ids are readable in URLs, the same in every build, and apart for packages that share a last segment. The alternative, `<package>.<name>` as for Python, would put `:` and `/` into ids.

Roles `pkg`, `proc`, `type`, `const`, `var`, `field`, `enumerator`, and `obj` take the Python domain's `~` and leading `.`. A target is looked up as `package.Type.name`, `package.name`, the name as written, then by the end of an object's name after a `:` or `/`, so `fmt.println` finds `core:fmt.println` as Odin code names a package by its last segment; a leading `.` searches any suffix, filtered by the role's object types.

Index entries read “println (procedure in core:fmt)”. `objects.inv` lists every object as `odin:<objtype>`, packages with priority 0.

Generating from source

`lib/odindoc` (a host library; it imports only Odin’s `core:` and `base:` packages) reads every `.odin` file of a package folder except `*_test.odin` and `#+ignore` files through the host’s confined reader, parses each with `core:odin/parser`, and records declarations in source order: kind, signature as written (the source span normalized to one line, never a procedure’s body), attributes, doc comment (the comment group directly above, and a field’s or enumerator’s comment at the end of its line), privacy (`@(private)`, `@(private="file")`, `#+private`), members, and targets.

`odin:autopackage`, `odin:autoproc`, `odin:autoprocgroupp`, `odin:autotype`, `odin:autoconst`, and `odin:autovar` hand the request to the host, as `autodoc`’s directives do (`Config.odin_autodoc`), and parse the answer where they stand. Options: `:members:`, `:undoc-members:`, `:private-members:`, `:exclude-members:`, `:member-order:` (source, alphabetical, groupwise), `:no-index:`.

Doc comments are plain text and become `reStructuredText` that reads as written: paragraphs and lists stay; an indented block is `code-block:: odin` (text after “Output:”); “Inputs:” and “Returns:” items become `:param name:` and `:result name:` fields (a new `result` field of the domain) or `:returns:`; “NOTE:” and “WARNING:” lines are admonitions; backquoted code is literal; every other markup character is escaped.

Platforms

Files are read as Odin builds them for every target at once. File-name suffixes (`_windows`, `_linux_arm64`, `_amd64`) and `#+build` tags (negations, the older `//+build`) give a file’s targets, a set of (OS, architecture) pairs Odin supports; file-scope when blocks comparing `ODIN_OS` and `ODIN_ARCH` narrow them. Declarations of one name and signature merge, with the union of their targets. One made for every target of the package is documented once; one made for some says where (“Availability: Windows”); one whose signature differs per target is documented once per signature, the first indexed. A package built only for some targets says so with `:platform:`. No setting is needed.

API pages

`odin_autoapi_dirs` lists folders, relative to the source folder; each package below them gets a page `<odin_autoapi_root>/<path>` holding `odin:autopackage` with `odin_autoapi_options` and `odin_autoapi_member_order`, and `<root>/index` lists the packages with their synopses in a hidden toctree. The pages are virtual documents: `discover` adds them to the project’s sources with their text, the build reads them as it reads files, and they never appear in the source folder. They take part in every builder, search, the general index, and the book; `odin_autoapi_add_toctree_entry` adds the index to the root document’s first toctree. `odin_collections` names folders for import-style arguments (`core:strings`) and gets no pages. All settings are checked when `conf.toml` loads, with Elm-style diagnostics.

Setting	Default	Meaning
<code>odin_autoapi_dirs</code>	<code>[]</code>	folders whose packages get pages
<code>odin_collections</code>	<code>{}</code>	collection names to folders
<code>odin_autoapi_root</code>	<code>"api"</code>	the pages' folder
<code>odin_autoapi_options</code>	<code>["members", "undoc-members"]</code>	options of each page
<code>odin_autoapi_member_order</code>	<code>"source"</code>	source, alphabetical, groupwise
<code>odin_autoapi_add_toctree_entry</code>	<code>true</code>	the index joins the root toctree
<code>odin_autoapi_generate_api_doc</code>	<code>true</code>	<code>false</code> : directives only

Every Odin file read is a dependency of the page that read it (`Doc_Info.deps`), so editing a package reads only its page again; the index's text holds the synopses, so its digest changes when one does. A page whose package was not found is read again on every build.

Rationale and alternatives

The Python domain's design was kept wherever Odin allows, so a Sphinx user needs to learn nothing new: the same options, the same info fields, the same `~` and `.`, the same nesting of members, and the same inventory. Signatures are Odin's own syntax rather than a C-like one, since Odin readers read declarations that way.

Parsing with `core:odin/parser` keeps the reader of the source the one the language ships, so every syntax Odin accepts is read. Running `odin doc` would give types checked by the compiler, but would run a compiler at build time, for one target at a time, and could not see the other targets' files; static analysis documents the whole cross-platform API.

Virtual documents, rather than files written into the source folder as `sphinx-autoapi` writes them, keep the source folder the author's and let incremental builds work on dependencies as `autodoc`'s pages do.

Security and operational considerations

Building never compiles or runs Odin code. Source is read only below `odin_autoapi_dirs` and `odin_collections`, through `host.confined`, as includes are, and within the document size limit. A package that does not parse is reported with a warning, and what could be read is documented.

Backwards compatibility

The domain adds directives and roles under `odin:`, a name Sphinx does not use; projects that do not use it are unaffected, and their documents read as before. The environment's version changed because documents may now be virtual.

Acceptance criteria and implementation status

Implemented: the domain (`lib/readers/sphinx/odin.odin`, tests in `odin_test.odin`, allocation-free reading in `reading_makes_no_allocator_calls`), `lib/odindoc` with its tests, the project's generating directives, API pages, resolution, inventory, and dependencies (`tests/odin_test.odin`), and `docs/api`, which builds Guidedog's own reference with strict HTML and PDF builds. API page counts are build observations, not a fixed part of the domain contract. The beta review reruns the package and integration suites with allocation leak checks and `AddressSanitizer`.

Open questions

- Private types that public declarations mention cannot be linked; `docs/api` lists them in `nitpick_ignore`. Should the generator show them without a link, or should `lib/` make such declarations private?
- Whether `A :: B` declares a type or a constant is told from the names; the compiler would know.
- when conditions other than `ODIN_OS` and `ODIN_ARCH` comparisons are not evaluated.
- A package module index (`odin-modindex`), as the Python module index, is not written.

Review history

2026-09-30: drafted with the implementation.

References

- GDS 0003, the Guidedoc engine and its allocation boundary.
- GDS 0006, Sphinx-style projects and recoverable publication: domains, autodoc, and the native project model.
- Sphinx 9.1, `sphinx.domains.python` and `sphinx.ext.autodoc`; `sphinx-autoapi`.
- Odin's `core:odin/parser`, `core:odin/ast`, and its file-name and `#+build` rules.

Lifecycle event

2026-10-11: prediscussion → discussion. Assigned a permanent number and opened for discussion.

Lifecycle event

2026-10-11: discussion → accepted. Maintainer-requested implementation-status reconciliation; current supported contract reviewed and deferred scope stated explicitly.

Lifecycle event

2026-10-11: accepted → committed. Current supported design implemented; beta review fixes and limitations are recorded. Native library and integration suites with leak checks; sanitizer and embedded-backend validation; `docs/manual/evidence/beta-review-20261011.md`. Deferred features are not claimed as implemented.