OpenD Author Guide

OpenD Documentation

Release 0.2

Generated 2026-11-01

Contents

OpenD overview

Get started

- Install the CLI from a checkout
- Validate a docs project
- Preview as different readers
- CLI reference

Writing docs

- Docs project layout
- opend.config.yaml reference
- Topics, maps and bookmaps
- Use keys for names, links and images
- Reuse content with conref and conkeyref
- Add images and diagrams
- Links, and links to hidden topics
- Document several releases with release branches
- preview-contexts.yaml reference
- Validate every pull request

Web editor

- Edit docs in the browser
- Edit and save a topic
- Publish your changes
- Resolve conflicts with the main branch

Applicability

- Applicability attributes
- Release range grammar
- Fail-closed filtering
- Conditional keys
- Filter at build time with DITAVAL

Context-sensitive help

- Check context help in the help lab
- Tag topics with context keys
- help-contexts.yaml reference
- How a context resolves, and its fallback rules

PDF publications

- PDF publications
- Define a publication with a bookmap
- Per-reader filtering in PDFs
- Theme the site, the panel and PDFs
- Choose a PDF template

Supported DITA subset

Validation diagnostics

OpenD overview

A docs platform where "which release applies" and "who may see this" are one rule, written once in the content and applied everywhere it is shown.

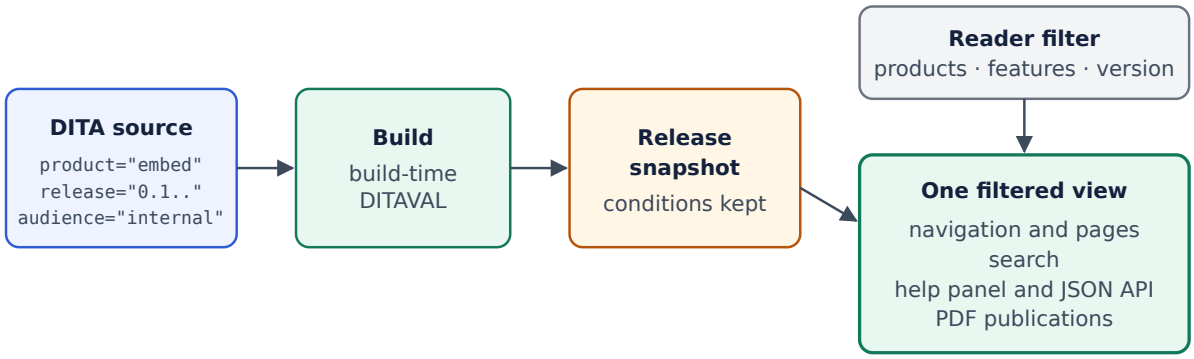
Authors write DITA in a Git repository or in a visual editor on the web. The build compiles it into a release snapshot that keeps every element's applicability, and each reader is served only what their release and entitlements allow: on the docs site, in the help panel inside your apps, in search, and in PDF manuals.

One rule for release and entitlement

Most documentation tools filter by release when they build, then handle access in a separate layer, if at all. In OpenD, any map entry, topic or element can carry standard DITA filtering attributes, and one filter decides both questions:

- `product` and `feature` are matched against what the reader's organization is entitled to.
- `release` takes version ranges, such as `0.1..`, and is matched against the release the reader runs.
- `audience`, `platform` and the other DITA attributes are filtered once, at build time, with a DITAVAL file.

Every output goes through the same filtered view



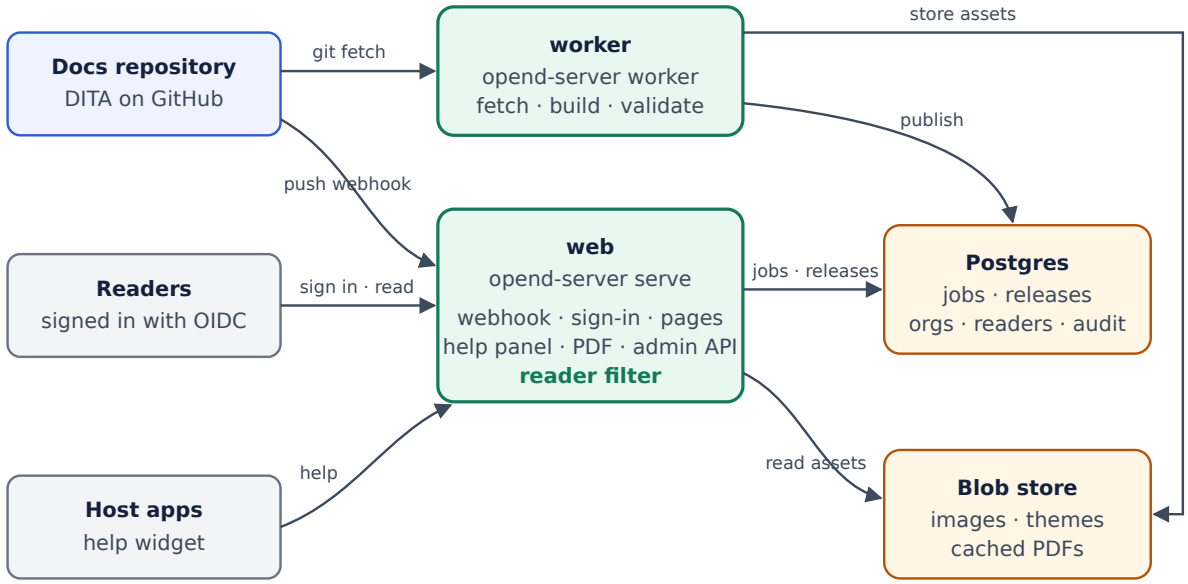
The filter is fail-closed and never leaks through links, titles or search: a link to a page the reader cannot see renders as plain text, and a hidden page answers with the same 404 as a missing one. See Applicability attributes.

Architecture

The same packages run on an author's laptop and on the server, so what an author previews is what readers get.

- **Locally**, the `opend` command validates a docs project, previews it as any reader, and prints PDF publications.
- **Hosted**, a push to the docs repository fires a GitHub webhook. A worker builds and validates the commit, and the web service serves signed-in readers their filtered pages. A push that fails validation never goes live.
- **In the browser**, the web editor lets authors who do not use Git change the same repository. They save to their own branch and publish through a pull request, so a change still goes live only by a merge and a passing build.
- **In your apps**, a small widget opens a help panel on the topic for the current screen, filtered for the signed-in user.

The hosted platform: web, worker, Postgres and a blob store



What release 0.2 includes

- The build and filter core, the `opend` CLI and a local previewer.
- The hosted path: push to publish, OIDC reader sign-in, filtered delivery and an admin API.
- Context-sensitive help embedded in your own web apps.
- PDF publications from bookmaps, filtered per reader.
- A web editor for authors who do not use Git, with live preview and publishing through pull requests.
- A version picker, so readers can choose which version of the docs they read.

Multi-tenancy and an operator console come later and reuse the same packages. See and Release notes for 0.1.

Where to go next

- Install the CLI from a checkout
- Applicability attributes
- Context-sensitive help in your apps
- Edit docs in the browser
- Hosted architecture: push to publish

Questions? Ask the operator of your docs site.

Get started

Install the CLI from a checkout

Build the `opend` command from the OpenD repository and check that it runs.

Before you begin

You need Node 22.12 or later, pnpm 10, Git, and a checkout of the OpenD repository.

About this task

The CLI is not published to npm yet, so you run it from a checkout. Everything it does happens on your machine: it never contacts a server.

1. Clone the repository and install its dependencies.

   ```
   git clone https://github.com/desktopmachineshop/opend.git
   cd opend
   pnpm install
   ```

2. Build the packages: `pnpm build`.
   The CLI runs the compiled output in `packages/cli/dist`, so build again after you pull changes.
3. Check the CLI runs: `pnpm opend --help`.
   It lists the `validate`, `build` and `preview` commands.
4. Optional: try it on the starter template.

   ```
   pnpm opend validate templates/docs-starter
   pnpm opend preview templates/docs-starter
   ```

Results

`pnpm opend` is a shortcut for `node packages/cli/dist/bin.js`. To use the CLI from your own docs repository, run that file with its full path, for example `node ~/src/opend/packages/cli/dist/bin.js validate .`, or pass the docs folder to `pnpm opend` from the checkout.

What to do next

Next: Validate a docs project, then Preview as different readers.

Validate a docs project

Run the same checks as CI and the hosted build: broken links, missing chunks, undeclared values, bad release ranges and more.

About this task

`opend validate` builds the project in memory and reports every problem with its file, line and column. The previewer and the hosted worker run exactly the same checks, so a project that validates locally publishes.

1. From the OpenD checkout, validate your docs folder.

   ```
   pnpm opend validate path/to/docs
   ```

   Each problem prints on its own line, followed by a summary:

   ```
   topics/install.dita:14:9: error [broken-link] link target not
   found: setup.dita
   topics/install.dita:20:7: warning [unresolved-key] key "edition" is
   not defined; using fallback content
   1 error(s), 1 warning(s)
   ```

2. Fix each problem.
   The code in brackets identifies the problem. Validation diagnostics explains every code and how to fix it.

3. Optional: validate a release branch or tag without checking it out.

   ```
   pnpm opend validate path/to/docs --ref v1
   ```

Results

The command exits with status 1 when there is at least one error, and 0 otherwise. Warnings do not fail it unless you pass `--strict`.

What to do next

To validate every pull request, see Validate every pull request. For every option, see CLI reference.

Preview as different readers

Serve the docs on your machine, rebuilding on save, and see exactly what each kind of reader gets.

About this task

The previewer uses the same build, filter and renderer as hosted delivery. "Preview as" is the real reader filter, not an approximation, so if a paragraph shows for a reader here, it shows for that reader in production.

1. Start the previewer on your docs folder.

   ```
   pnpm opend preview path/to/docs
   ```

   It prints any problems, then `Serving http://127.0.0.1:4000/ (watching for changes).` Open that address. Pages reload when you save a file.

2. Look at the staff view.
   With no reader options the previewer shows everything that survives the build-time filter, and marks each conditional element with its attributes, such as `product=embed`. Every alternative of a conditional key is shown, separated by a bar. Readers never see these markers.

3. Switch reader with **Preview as** in the banner at the top of each page.
   The list holds the readers in `preview-contexts.yaml`. The choice is kept in a cookie, so it sticks while you browse. The banner also shows the reader's products, features and version, and a summary of any validation problems.

4. Or start the previewer as a particular reader.
   - `--as admin` picks a named reader from `preview-contexts.yaml`.
   - `--product core,embed --feature early-access --version 0.1` describes a reader on the command line. `--version` defaults to the project's release.
   - `--all` is the staff view, the default.

5. Optional: preview a release branch or tag without checking it out.

   ```
   pnpm opend preview path/to/docs --ref v1
   ```

   A git ref is read once and not watched.

Results

Besides the pages, the previewer serves a few helper pages:

`/search?q=…`

The search box in the page header: the current reader's pages that match, searched in memory. See Search the docs.

`/_opend/publications`
Every PDF publication, and whether the current reader can see anything in it.

`/pdf/<name>.pdf` **and** `/pdf/<name>.html`
A publication printed for the current reader, and its print HTML.

`/_opend/help-lab`
What every context-sensitive help key and route resolves to. See Check context help in the help lab.

`/_opend/diagnostics.json`
The current validation problems as JSON.

What to do next

To add your own sample readers, see preview-contexts.yaml reference.

CLI reference

Every `opend` command and option.

Each command takes the docs root, the folder that holds `opend.config.yaml`, as its argument. It defaults to the current folder. Run `pnpm opend help <command>` for the same information.

opend validate [dir]

Runs every check and prints the problems. Exits with status 1 on any error, or on any problem with `--strict`.

Option	Default	Meaning
`--ref <ref>`	working tree	Validate a git branch or tag instead of the files on disk.
`--strict`	off	Treat warnings as failures.
`--format <format>`	`text`	`text` prints `file:line:column: severity [code] message` and a summary. `json` prints the problems as a JSON array. `github` prints GitHub Actions workflow annotations, so problems show inline on pull requests.

opend build [dir]

Compiles the docs into the release snapshot the hosted build stores: JSON holding every topic with its conditions, the navigation, publications, context-sensitive help keys and the reuse index. Fails without writing anything when there is an error.

Option	Default	Meaning
`--ref <ref>`	working tree	Build a git branch or tag.
`--out <file>`	`.opend/build/<release>.json`	Where to write the snapshot. The default is inside the docs root.

opend preview [dir]

Serves the docs locally and rebuilds on save. See Preview as different readers.

Option	Default	Meaning
`--as <name>`		Preview as a named reader from `preview-contexts.yaml`. An unknown name is an error that lists the known ones.
`--product <list>`		Preview as a reader holding these products, comma-separated. Can be repeated.
`--feature <list>`		Preview as a reader holding these features, comma-separated. Can be repeated.
`--version <version>`	the project's release	The reader's release, such as `0.1`.
`--all`	on	The staff view: everything, with conditions marked. Used when no reader option is given.
`--ref <ref>`	working tree	Preview a git branch or tag. Not watched.
`-p, --port <port>`	`4000`	Port to listen on. `0` picks a free port.
`--host <host>`	`127.0.0.1`	Interface to bind. Only change it on a trusted network: the staff view shows everything.
`--no-watch`	watching	Do not rebuild when files change.

Writing docs

Docs project layout

A docs project is a folder, usually a whole Git repository, with `opend.config.yaml` at its root. Copy `templates/docs-starter` to start one.

Only `opend.config.yaml` and the root map it names are required. The rest is convention, and the starter template follows it:

Path	What it holds
`opend.config.yaml`	Release, root map, publications, declared attribute values, build filter, theme and logo. See opend.config.yaml reference.
`maps/`	The site map, bookmaps for PDF publications, and submaps such as shared key definitions.
`topics/`	One topic file per page. The page URL is the file path without its extension, for example `/topics/install.html`.
`reuse/`	Reusable chunks pulled in with `conref` or `conkeyref`. Kept out of every map, so they are never published on their own.
`images/`	SVG, PNG, JPEG, GIF or WebP files that topics reference.
`filters/publish.ditaval`	The build-time filter, for example to drop `audience="internal"` for everyone.
`theme/site.css`	Brand colours and fonts for the site, the help panel and PDFs.
`preview-contexts.yaml`	Sample readers for the previewer. See preview-contexts.yaml reference.
`help-contexts.yaml`	Maps host-app routes to context-sensitive help keys. See help-contexts.yaml reference.
`.github/workflows/validate.yml`	Validates every pull request. See Validate every pull request.

What the build reads

- Files ending in `.dita`, `.ditamap`, `.bookmap`, `.ditaval`, `.xml`, `.yaml` and `.yml` are read as text. Other files are only checked for existence until something references them.
- Folders named `.git`, `node_modules`, `.opend` and `dist` are ignored.

- `.xml` files whose root is not a topic are left alone, so other XML can live in the project.
- Paths in maps, references and the config must stay inside the docs root.

opend.config.yaml reference

Every field of the project configuration file, with its default and rules.

Field	Default	Meaning and rules
`release`	required	The version this branch documents, such as `0.1` or `2.1`. The hosted build also requires it to be a version number: its major part is the release line (its major and minor parts on a repository with minor lines), so `1.4` readers keep getting the 1.x docs after 2.0 is published.
`map`	required	The root map for the site navigation and keys, such as `maps/docs.ditamap`.
`publications`	`[]`	Bookmaps (or maps) published as PDFs. Their file names must be unique, because each is served at `/pdf/<file name>.pdf`.
`attributes`	`{}`	The allowed values of each filtering attribute, such as `product: [core, hosted, embed]`. Keys must be filtering attributes: `product`, `feature`, `audience`, `platform`, `otherprops`, `props` or `deliveryTarget`. `release` must not be listed: it takes version ranges. Each value is a single token of letters, digits, `_`, `.`, `:` and `-`.
`buildFilter`	none	A DITAVAL file applied to every reader at build time. See Filter at build time with DITAVAL.
`theme`	none	A stylesheet loaded after the built-in styles on every page, in the help panel and in PDFs. See Theme the site, the panel and PDFs.
`logo`	none	An image shown before the site title in the page header: an `.svg`, `.png`, `.jpg`, `.jpeg`, `.gif` or `.webp` file. It is drawn 28 pixels high and is decorative, because the title follows it. An SVG logo can carry its own `prefers-color-scheme` rule for dark mode, as this site's `theme/logo.svg` does.

Field	Default	Meaning and rules
`pdf`	the `standard` template	How publications print: a built-in `template` (`standard`, `report`, `compact` or `handbook`) and any of its settings to change, such as `pageSize`, `cover` or `footer`. See Choose a PDF template.
`site`	none	How the hosted site is set up, applied by the hosted build: `releaseRefs`, `releaseLines`, `access`, `publicEntitlements` and `versionPicker`, each optional and with the admin API's values. The CLI only validates it. See Set the site up from opend.config.yaml.
`title`	`Documentation`	The site title in the page header and the PDF cover. Set it to an empty string to use the root map's title instead.

Example

```
title: OpenD Documentation
release: '0.1'
map: maps/docs.ditamap
publications:
  - maps/admin-guide.bookmap
  - maps/author-guide.bookmap
attributes:
  product: [core, hosted, embed]
  feature: [early-access]
  audience: [author, admin, developer, internal]
buildFilter: filters/publish.ditaval
theme: theme/site.css
logo: theme/logo.svg
pdf:
  template: standard
  footer: OpenD {release}
site:
  releaseRefs: [main, 'v*']
  access: public
  publicEntitlements: { products: [core], version: latest }
  versionPicker: { mode: live }
```

Quote the release. Unquoted, YAML reads `0.1` as a number, and the config is rejected.

Every path must stay inside the docs root: no leading `/` and no `..`. Mistakes are reported as `invalid-config`.

Topics, maps and bookmaps

Each page is a topic in its own file. A map arranges topics into the site navigation and defines keys; a bookmap arranges them into a PDF publication.

Topics

A topic file holds one topic: `topic`, `concept`, `task`, `reference`, `glossentry` or `troubleshooting`. Its root element needs an `id`, and every `id` inside one topic must be unique.

```
<task id="install-cli">
  <title>Install the CLI from a checkout</title>
  <shortdesc>Build the opend command and check that it runs.
</shortdesc>
  <taskbody>
    <steps>
      <step><cmd>Clone the repository.</cmd></step>
    </steps>
  </taskbody>
</task>
```

The page URL is the file's path without its extension, so `topics/start/install-cli.dita` is served at `/topics/start/install-cli.html`. Keep file names stable: links from outside the site depend on them.

A topic may contain nested topics, which render as sections of the same page and can be linked with `file.dita#nested-id`. A `<dita>` wrapper may hold several topics; the first is the page.

Maps

The root map named by `map` in `opend.config.yaml` defines the navigation, in document order:

```
<map>
  <title>OpenD Documentation</title>
  <mapref href="keys.ditamap"/>
  <topicref href="../topics/overview.dita"/>
  <topichead navtitle="Get started">
    <topicref href="../topics/start/install-cli.dita"/>
  </topichead>
  <topichead navtitle="Embedded help" product="embed">
    <topicref href="../topics/help/help-overview.dita"/>
  </topichead>
</map>
```

- `topicref` adds a topic. Without a `navtitle`, the navigation shows the topic's title as the reader sees it, so a conditional phrase in a title stays filtered.
- `topichead` adds a heading with a `navtitle` and no page. A heading with nothing visible under it is left out of that reader's navigation.
- `topicgroup`, `frontmatter` and `backmatter` group entries without adding a level.
- `mapref` (or a `topicref` to a `.ditamap`) includes a submap at that point. Its entries and keys are merged in; reference cycles are errors.
- `toc="no"` publishes a topic without listing it in the navigation.

- `keydef` and `processing-role="resource-only"` define keys without adding navigation. See Use keys for names, links and images.

A topic is published to a reader only when a navigation entry they can see points at it and the topic's own root passes their filter. Filtering attributes on a map entry apply to everything under it.

Bookmaps

A bookmap listed under `publications` becomes a PDF. Its `booktitle/mainbooktitle` is the publication title, and `chapter`, `part` and `appendix` entries work like `topicref`. A bookmap resolves keys in its own keyspace, so pull shared keys in with a `mapref`. See Define a publication with a bookmap.

Use keys for names, links and images

Define a value once in a map, with `keydef`, and refer to it everywhere with `keyref`. Changing the map changes every page.

About this task

Keys suit product names, version numbers, external URLs, and topics that many pages link to. These docs keep theirs in `maps/keys.ditamap` and pull it into every map with a `mapref`.

1. Define the key in a map.

```
<!-- Variable text -->
<keydef keys="product-name">
  <topicmeta><keywords><keyword>OpenD</keyword></keywords>
</topicmeta>
</keydef>
<!-- An external link with default link text -->
<keydef keys="repo"
href="https://github.com/desktopmachineshop/opend" scope="external"
format="html">
  <topicmeta><linktext>the OpenD repository</linktext></topicmeta>
</keydef>
<!-- A topic -->
<keydef keys="env-vars" href="../topics/hosting/env-vars.dita"/>
```

2. Refer to it from a topic.

```
<title><keyword keyref="product-name"/> overview</title>
<p>Clone <xref keyref="repo"/>.</p>
<p>See <xref keyref="env-vars"/>.</p>
<image keyref="logo"><alt>Logo</alt></image>
```

An empty element takes the key's text: the `keyword` or `linktext` in its `topicmeta`, or else the target topic's title, or else the URL. An element with its own content keeps that content and only takes the key's target.

3. Validate.
   An undefined key in `keyref` is a warning, `unresolved-key`, and the element's own content is used as a fallback. An undefined key in `conkeyref` is an error.

Results

When a key is defined more than once, this definition wins:

- Within one map, the first definition in document order.
- A map's own definitions win over those of the submaps it references.
- Each map and bookmap has its own keyspace. A topic used by both the site map and a bookmap is compiled once, with the site map's keys.
- A key may have several definitions with different conditions. Each reader gets the first one their filter keeps: see Conditional keys.

Key scopes are not supported yet.

Reuse content with conref and conkeyref

Write a note, step or paragraph once in a chunks file and pull it into any topic. The build inlines it, and every page that uses it updates together.

1. Put reusable elements, each with an `id`, in a topic under `reuse/`.

```
<topic id="common">
  <title>Reusable content</title>
  <body>
    <note id="restart" type="note">Restart the web service and the
worker after changing a variable.</note>
    <steps>
      <step id="step-build"><cmd>Build the packages: <codeph>pnpm
build</codeph>.</cmd></step>
    </steps>
  </body>
</topic>
```

Leave the file out of every map. A chunks file is not published on its own, and validation does not warn about it as long as some published topic uses one of its chunks.

2. Pull a chunk in with `conref`, using the path, the topic id and the element id.

```
<note conref="../../reuse/common.dita#common/restart"/>
```

3. Or define a key for the chunks file once, and use `conkeyref` so topics need no relative paths.

```
<!-- in a map -->
<keydef keys="reuse" href="../reuse/common.dita"/>

<!-- in any topic -->
<step conkeyref="reuse/step-build"/>
```

This page itself pulls its restart note in with `conkeyref`:

> **Note:** The server reads its settings once, at start-up. After you change a variable, restart (or redeploy) both the web service and the worker.

Results

How references behave:

- References work within a file and across files, and chunks may reference other chunks. Cycles are reported as `conref-cycle`.
- The referencing element should have the same name as the chunk. A `<p>` that pulls a `<note>` works but warns (`conref-type-mismatch`).

- Filtering attributes on the referencing element apply to the pulled content, so `<note conkeyref="reuse/restart" product="core"/>` shows the shared note to core readers only. Attributes on the chunk itself apply too.
- Keys inside a chunk resolve in the keyspace of the map publishing the topic that pulls it.
- `conrefend` and `conaction` are not supported: only the start element is pulled, with a warning.

The release snapshot records which topics use each chunk, so tools can show the impact of changing one.

Add images and diagrams

Reference image files from topics with `<image>`. They are published with the release and shown only to readers who can see a topic that uses them.

1. Put the file in the project, for example in `images/`.
   SVG, PNG, JPEG, GIF and WebP are served on the site and embedded in PDFs. For diagrams, prefer SVG with a `<title>` and `<desc>` and a solid background, so it reads in both light and dark themes. The diagrams on the overview are hand-written SVG.

2. Reference it with a path relative to the topic, and always give alternative text.

   ```
   <fig id="fig-architecture">
     <title>The hosted platform</title>
     <image href="../images/architecture.svg" placement="break">
       <alt>Diagram: the docs repository sends a push webhook to the
   web service...</alt>
     </image>
   </fig>
   ```

   `placement="break"` puts the image on its own line; without it the image is inline. `width` and `height` are passed through. `keyref` works too, with the key's `href` naming the file.

3. Validate.
   A missing file is an error, `missing-image`. External images (an `https:` URL or `scope="external"`) are not checked and are not embedded in PDFs.

Results

On the hosted site, images are served from `/_assets/` only when a topic visible to the signed-in reader references them, so guessing a path reveals nothing. Author files, SVG

included, are served with a sandboxing Content Security Policy.

Links, and links to hidden topics

Link with `xref` or `related-links`. A link to a topic the reader cannot see turns into plain text, so it never gives away what is behind it.

Writing links

```
<xref href="install-cli.dita"/>                        <!-- another
topic; text is its title -->
<xref href="preview.dita#preview/staff">the staff view</xref>   <!--
an element in a topic -->
<xref href="#links/hidden"/>                             <!-- an
element in this topic -->
<xref keyref="env-vars">environment variables</xref>    <!-- through a
key -->
<xref href="https://example.com" scope="external"
format="html">Example</xref>
```

- An empty link shows the target topic's title as the reader sees it.
- Links to other local files, such as a sample YAML file, are served like images, only to readers who can see the linking topic.
- A link to a missing file, topic or element is an error, `broken-link`.
- External links may use `http`, `https`, `mailto`, `tel` or `ftp`, and external images `http` or `https`. Any other scheme, such as `javascript:` or `data:`, renders as plain text.
- `related-links` with `link` elements render as a "Related information" list. Links whose target is hidden are left out of it.

Links to hidden topics

Pages are shared by many readers, so a page often links to a topic some of its readers cannot see. For those readers, the link renders as its authored text, without the target's title and without a URL. A link with no authored text disappears.

So when a link's target may be hidden, give it link text that makes sense on its own, as the examples above do:

```
<!-- Good: reads well as plain text -->
<p>Operators set this in the <xref keyref="env-vars">server
environment</xref>.</p>
<!-- Risky: an empty link vanishes for readers who can't see the
target -->
<p>See <xref keyref="env-vars"/>.</p>
```

Or put the condition on the sentence that holds the link, so it disappears as a whole. For example, this sentence is only shown to readers with the embed product:

To open help in your own app, see Embed the help widget.

Links to topics outside the navigation

A topic that is in no navigation entry is never published, so links to it are plain text for everyone. Validation warns about them as `link-to-unpublished`. Add the topic to the site map, with `toc="no"` if it should not be listed.

In PDF publications, links to topics in the same publication jump within the PDF; links to other topics point at the docs site when the reader can see them there.

Document several releases with release branches

Keep one branch or tag per major release, and use `release` ranges for the differences between minor releases inside it.

About this task

Both tools work together. A branch per major release keeps old docs stable while new work lands on `main`. Ranges within a branch avoid copying a topic for every minor release: a step marked `release="2.1.."` appears for 2.1 readers and later, and is hidden from 2.0 readers.

1. Create a branch for the release you are freezing, such as `v1`, and set `release` in its `opend.config.yaml`, for example to `'1.4'`.
2. On `main`, set `release` to the next major version, such as `'2.0'`.
3. Preview or validate any branch without checking it out.

   ```
   pnpm opend preview path/to/docs --ref v1 --version 1.4
   pnpm opend validate path/to/docs --ref v1
   ```

4. Make sure the docs repository's release refs include every branch that publishes.

A repository registered with `"releaseRefs": ["main", "v*"]` publishes pushes to `main` and to any branch or tag starting with `v`; `*` matches any run of characters. The default is `["main"]`.

Results

Each successful build goes live for its release line, the major number of its `release`. Readers get the line matching their own version, so 1.x and 2.x docs are live side by side. A repository can group releases by minor version instead, so 0.22 and 0.23 are lines of their own; see Release lines by major or minor version. A build replaces the live release of its line only if it comes from a newer push, so a slow build of an old commit cannot overwrite a newer one.

For the exact range syntax, see Release range grammar.

preview-contexts.yaml reference

Sample readers for the previewer. Each is what the entitlement service would produce for a signed-in reader.

```
contexts:
  self-hoster:
    org: acme
    products: [core]
    version: '0.1'
  managed-customer:
    org: globex
    products: [hosted, embed]
    features: []
    version: '0.1'
```

Field	Default	Meaning
`org`	none	Shown in the banner, and on the cover of PDFs printed in the previewer. It does not affect filtering.
`products`	`[]`	Products the reader's organization holds.
`features`	`[]`	Features the reader's organization holds.
`version`	required	The release the reader runs, such as `0.1`. Quote it.

Each name becomes an entry in the banner's **Preview as** list, and a value for `--as`. A malformed file is reported as `invalid-preview-contexts`.

Choosing sample readers

Cover each combination that changes what readers see: every product, the features that gate whole topics, and each release line you publish. Also include a reader with nothing, to check that the ungated pages still make sense on their own. The previewer only shows these readers; it does not check them against your real organizations.

Validate every pull request

Run `opend validate` in GitHub Actions so problems show inline on the pull request, before anything is merged or published.

About this task

The starter template ships `.github/workflows/validate.yml`, which runs the published CLI with `npx @opend/cli@0`. Until the CLI is published to npm, build it from a checkout of the OpenD repository instead, as this workflow does.

1. Add a workflow to the docs repository.

```
name: Validate docs
on:
  pull_request:
  push:
    branches: [main, 'v*']
jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          path: docs
      - uses: actions/checkout@v4
        with:
          repository: desktopmachineshop/opend
          path: opend
      - uses: pnpm/action-setup@v4
        with:
          package_json_file: opend/package.json
      - uses: actions/setup-node@v4
        with:
          node-version: 22
      - run: pnpm install --frozen-lockfile && pnpm build
        working-directory: opend
      - run: node opend/packages/cli/dist/bin.js validate docs --
format github
```

2. Optional: add `--strict` to fail on warnings too.
   This site's own CI validates `site/` with `--strict`.

Results

With `--format github`, each problem becomes a workflow annotation on the line it is about, titled with its code.

On the hosted platform the worker runs the same validation on every push, and a failing push never goes live. CI tells authors earlier.

Web editor

Edit docs in the browser

The web editor lets people who do not use Git change the docs. You save to your own working branch as often as you like, and publish through one pull request when you are ready.

How it works

- The docs repository stays the only copy of the docs. The editor reads from it and writes to it through GitHub; it keeps no content of its own.
- **Save** as often as you like. Each save is a commit to your own working branch. Nothing goes live and nobody is asked to review.
- **Publish** when you are ready. The editor opens one pull request for everything you saved, and the repository's review rules on GitHub decide when it can merge.
- Once it merges, the docs site builds and publishes it like any other push.
- Commits name you as the author, even if you have no GitHub account, and the editor keeps a record of what you did. If you sign in with GitHub, they are commits by your own account, marked Verified by GitHub; see Verified commits.

Sign in

The editor has its own address, separate from the docs site. Your reader account does not open it, and signing in to the editor gives you nothing extra on the docs site.

- **Sign in with GitHub** is for developers. If you can already push to the docs repository, you get in the first time you sign in.
- **Sign in with your company account** is for everyone else. An administrator must add you first, by the email address your company account uses.

If the editor says you cannot edit these docs, or that you are not registered, ask whoever runs the editor to add you. Signing out ends your session at once.

The pages

Files

The topics and maps in the repository, with the ones you changed marked. Also where you create a topic and preview the docs as a customer org.

Publish

Your changes, what the repository needs before they can merge, and your pull request.

Activity

What you did in the editor: sign-ins, saves, merges from the main branch, conflict choices, publishes and previews, with their commits and pull requests.

The editor changes the DITA source of topics (`.dita`) and maps (`.ditamap`, `.bookmap`) in a text box. There is no visual editor yet. Images and other files are still changed in Git.

Next

- Edit and save a topic
- Publish your changes
- Resolve conflicts with the main branch
- Verified commits

Edit and save a topic

Change a topic in the visual editor or its source, watch the preview update as you type, and save it to your working branch.

1. On the **Files** page, choose the topic or map.
   Files you changed on your branch are marked **changed**.

2. Edit the topic on the **Visual** tab.
   Topics open looking much as they read, with the title, short description and body editable in place. The toolbar adds bold, italic, code and other formatting, links, bulleted and numbered lists, steps in a task, notes, code blocks, images and sections. **Block settings** sets who sees the block holding the cursor (release, product, feature, audience and the other filtering attributes) and a note's type. It opens beside the text, or at the bottom of the screen on a phone, and stays in view while you edit; the block it changes is outlined. Blocks shown only to some readers are outlined and labelled.

   Markup the visual editor does not edit directly, such as tables and reused content, appears as a locked block. It is kept exactly as it is; double-click it to change its XML. Saving writes back only the blocks you changed, so the rest of the file keeps its formatting.

   Maps, and topics the visual editor cannot convert and save back without changing them, open on the **Source** tab with a note saying why. Developers can switch to **Source** at any time to edit the DITA directly.

   The **Preview** tab, after **Source**, shows the page with your unsaved text. It is the staff view: everything is shown and conditions are marked. Under **Show the page for**, choose a value for any of the conditions the content uses (a version, product, feature, audience and so on) and choose **Show** to see only what readers with that combination see. **Show everything** goes back to the full staff view. **Problems** lists what validation finds in the file, the same as `opend validate`; see Validation diagnostics.

3. Optional: check what a customer sees. Under the preview, on the **Preview** tab, choose an org and **Open**.
   The page opens with that org's products, features and version, as the docs site would show it. It shows your last save, not unsaved text. The link expires after 30 minutes and is logged.

4. Optional: add a note about the change, then choose **Save**.
   The save is a commit to your working branch, and the page shows its short id. Your first save creates the branch from the main branch. Nothing is published.

Results

- A save is refused, and nothing is saved, when the file is not well-formed XML. Fix the problem it names and save again.
- Validation problems do not stop a save, but they stop you publishing.
- If the file changed since you opened it, by another save or by a merge from the main branch, the save is refused so it cannot overwrite that change. Copy your text, reload the page, and apply it again.
- After a save, the editor merges the latest main branch into yours. If that conflicts, it takes you to Resolve conflicts with the main branch.

Create a topic

Start a topic from a template and, optionally, add it to the navigation.

1. On the **Files** page, under **New topic**, fill in the form.
   - **Title**: the topic's title.
   - **File name**: lowercase letters, digits and dashes. `.dita` is added.
   - **Folder**: where the file goes.
   - **Type**: concept (explains what something is), task (step-by-step instructions), reference (facts to look up) or a general topic.
   - **Add to navigation**: a map to add it to, or **Not yet**.

2. Choose **Create topic**.
   The new topic, and the map if you chose one, are saved to your working branch in one commit, and the topic opens for editing.

Publish your changes

Publishing sends everything saved on your branch as one pull request. The repository's review rules on GitHub decide when it merges.

Before you begin

You can publish once your branch has saved changes, no validation errors and no unresolved conflicts with the main branch. The **Publish** page says which of these is missing.

1. Open **Publish**.
   What publishing needs lists what the main branch requires before a change can merge, read from its branch protection and rulesets on GitHub: a number of approving reviews, a review from a code owner, or checks that must pass. **Your changes** lists the files on your branch.
2. Choose **Publish**, or **Update and publish** when your pull request is already open. The editor opens a pull request for your branch, or updates the open one. If nothing blocks the merge, it merges at once and says `Published: your changes are merged and will go live after the build.` Otherwise the pull request stays open, and the page lists who approved it, its checks, and what still blocks the merge.
3. When reviewers approve and the checks pass, choose **Update and publish** again. A maintainer can also merge the pull request on GitHub.

Results

What can block the merge, and what to do:

The page says	What to do
Needs approving reviews, or a review from a code owner	Wait for reviewers on GitHub, then publish again.
A required check failed or has not passed yet	Wait for it to finish. If it failed, open the pull request to see why, fix it in the editor and save.
Changes requested	Keep editing on your branch: each save updates the pull request. Publish again when you are done.
The pull request has conflicts with the base branch	Open any page in the editor to merge the main branch in, and resolve the conflicts.
GitHub does not let the editor read the rules, or the rules could not be read	The editor cannot tell whether the merge is allowed, so it never merges by itself. A maintainer merges the pull request on GitHub.

What to do next

After the merge, your working branch is retired, and your next save starts a new one. The docs site builds the merged commit like any push, and it goes live once the build passes. The commit on the main branch credits you as co-author.

Resolve conflicts with the main branch

When someone changed the same lines on the main branch as you did, choose which version to keep, or combine them by hand.

About this task

Each time you open a page, and after every save, the editor merges the latest main branch into your working branch. It never rewrites your commits. Changes to different files, or to different parts of one file, merge on their own. When the same lines changed on both sides, the editor shows `The main branch has changes that conflict with yours`, and you cannot publish until you resolve them.

On a repository that requires signed commits, GitHub does the merging instead, through your pull request; see Verified commits.

1. Choose **Resolve the conflicts**.
   Each file in conflict shows **Your version** and the **Current version** on the main branch side by side.

2. For each file, choose one.
   - **Keep your version**.
   - **Keep the current version**, dropping your change to that file.
   - **Edit by hand**, then combine the two in the box below.

   The hand-edit box starts with both versions between marker lines (`<<<<<<<`, `=======`, `>>>>>>>`). Keep the text you want and delete the marker lines. A version with markers left in, or that is not well-formed XML, is refused; see Validation diagnostics.

3. Choose **Save my choices**.
   Your choices are saved to your branch as a merge commit. Nothing is published.

   If your branch or the main branch changed while you were choosing, the editor refuses the save and asks you to review the files again.

Results

Some conflicts cannot be resolved by choosing a version:

- you deleted a file that was changed on the main branch, or it was deleted there after you changed it;

- a file that is not text, such as an image, changed on both sides;
- sections you edited (elements with an `id`) were deleted on the main branch.

The page then says `A developer needs to resolve this`. Choose **Ask a developer**: the editor opens or updates your pull request with a note naming the files, and a developer resolves them there.

Applicability

Applicability attributes

Standard DITA filtering attributes on any map entry, topic or element decide which readers see it. One rule covers both the reader's release and what their organization is entitled to.

The attributes

Attribute	Values	A reader sees the content when
`product`	tokens declared in `opend.config.yaml`	their organization holds at least one listed product
`feature`	tokens declared in `opend.config.yaml`	their organization holds at least one listed feature
`release`	versions and ranges such as `0.1`, `2`, `2.0..`, `..1.9`, `1.2..1.9`	their version matches at least one listed range
`audience`, `platform`, `otherprops`, `props`, `deliveryTarget`	tokens declared in `opend.config.yaml`	the build-time DITAVAL does not exclude them

For example, this site declares `product: [core, hosted, embed]`. The self-hosting guides carry `product="core"`, the embedded-help section carries `product="embed"`, and the architecture overview carries `product="core hosted"` because it matters to anyone with either.

How conditions combine

These are DITAVAL semantics:

- Values within one attribute are alternatives: `product="core hosted"` shows to a reader holding either.
- Separate attributes must all pass: `product="embed" release="0.2.."` needs the embed product and release 0.2 or later.

- A condition on an ancestor applies to everything inside it: an element, a topic, a `topicref` or a `topichead`.
- A topic is published to a reader only when a navigation entry they can see points at it and the topic's own root passes.

```
<topichead navtitle="Embedded help" product="embed">   <!-- whole
section -->
  <topicref href="../topics/help/tokens.dita"/>
</topichead>
<p product="core">Set OPEND_EMBED_SECRET on the server.</p>   <!-- one
paragraph -->
<step release="..0.1">...</step>                              <!-- up
to 0.1.x -->
```

Reader attributes and build attributes

`product`, `feature` and `release` are reader attributes: they are kept in the release snapshot and decided for each reader at request time. The other attributes are build attributes: the build-time DITAVAL decides them once for everyone, and readers are never gated by them. So `audience="admin"` labels content, but does not hide it from anyone unless the DITAVAL excludes that value. Use `product` or `feature` to gate readers.

What readers never see

- Content their filter excludes, in pages, navigation, search, the help panel or PDFs.
- The title of a hidden topic: a link to it renders as its authored text only. See Links, and links to hidden topics.
- Filtering attributes themselves: they are stripped from the HTML.

A page the reader cannot see returns the same 404 as a page that does not exist, so guessing URLs reveals nothing.

> **Important:** Readers are fail-closed. A value they don't hold, a value that is not declared in `opend.config.yaml`, or a release range that does not parse hides the content instead of showing it.

More

- Release range grammar, and its grammar
- Fail-closed filtering
- Conditional keys
- Filter at build time with DITAVAL

Release range grammar

The exact syntax and matching rules of the `release` attribute.

Grammar

```
value   = token { whitespace token }
token   = version                (* exact: that version and
everything under it *)
        | version ".."           (* from: that version and later *)
        | ".." version           (* to: up to and including
everything under it *)
        | version ".." version   (* from .. to *)
version = [ "v" | "V" ] digits { "." digits }
digits  = "0" .. "9" { "0" .. "9" }
```

Content applies when the reader's version matches *any* token. `..` on its own is not a range, and a range whose start is after its end is invalid. Validation reports any token that does not parse as `invalid-release`.

Examples

Token	Matches	Does not match
`1.4`	1.4, 1.4.0, 1.4.7	1.40, 1.5, 1
`2`	2, 2.0, 2.13.1	1.9, 3.0
`2.0..`	2.0, 2.1, 10.0	1.9.9
`..1.9`	0.1, 1.0, 1.9, 1.9.12	1.10, 2.0
`1.2..1.9`	1.2, 1.5.3, 1.9.4	1.1.9, 1.10, 2.0
`v2.0..`	the same as `2.0..`	
`1 3..`	any 1.x, and 3.0 and later	2.x

Matching rules

- Versions compare number by number, so 1.10 is after 1.9. A missing number counts as 0 when comparing, so `2.0..` matches 2.
- An exact token and the end of a range compare only as many numbers as they have. That is why `1.4` matches 1.4.7, and `..1.9` includes every 1.9.x.

- An exact token with more numbers than the reader's version does not match it: `1.4` does not match a reader on `1`.
- The start of a range is inclusive and compares the full version.
- Release values are not declared in `opend.config.yaml`, because they are ranges.

Reader versions and release lines

Each reader has one version, from their organization's entitlements, such as `0.1`. It must be a version of the same form, with an optional leading `v`. A reader whose version does not parse matches no release token, so content with a `release` attribute is hidden from them.

On the hosted platform, a reader is served the live release of their *release line*: the major number of their version, or on a repository with lines by minor version its major and minor numbers. A `1.4` reader keeps getting the docs built from the branch whose `release` is 1.x after 2.0 is published; with minor lines, a `0.22` reader keeps the 0.22 docs after 0.23. See Document several releases with release branches and Release lines by major or minor version.

Fail-closed filtering

When OpenD is unsure whether a reader may see something, it hides it. A mistake in a condition can hide content, but never leak it.

For readers, each of these excludes the content it is on:

- a `product` or `feature` value the reader's organization does not hold;
- a value that is not declared in `opend.config.yaml`, such as a typo like `product="embd"`, because no reader can hold it;
- a `release` token that does not parse, such as `release="next"`;
- any `release` condition, when the reader's own version does not parse.

Because a typo silently hides content, validation reports undeclared values (`undeclared-value`) and bad ranges (`invalid-release`) as errors, and the hosted build refuses to publish them. You find out in the previewer or in CI, before readers do.

Lookups fall back, they don't reveal

The same rule holds wherever content is looked up rather than browsed:

- A link to a topic the reader cannot see renders as its authored text, without the target's title.
- A context-sensitive help key whose topic is hidden falls back to a parent key or the app's default, never to the hidden topic.
- An image is served only when a topic visible to that reader references it.
- A PDF publication with nothing visible returns the same 404 as one that does not exist.

A page the reader cannot see returns the same 404 as a page that does not exist, so guessing URLs reveals nothing.

Build attributes are different

`audience`, `platform`, `otherprops`, `props` and `deliveryTarget` follow DITAVAL rules at build time instead: a value with no rule is included. Declare them anyway; validation still reports undeclared ones.

Conditional keys

Give a key several definitions with different conditions, and each reader gets the first definition their filter keeps, which is DITA's rule applied per reader.

These docs define who to ask for help twice:

```
<keydef keys="help-contact" product="hosted">
  <topicmeta><keywords><keyword>the operator of your docs
site</keyword></keywords></topicmeta>
</keydef>
<keydef keys="help-contact">
  <topicmeta><keywords><keyword>the OpenD maintainers on
GitHub</keyword></keywords></topicmeta>
</keydef>
```

A topic writes `<ph keyref="help-contact"/>`. For you, right now, that reads: ask the operator of your docs site.

How it works

A static DITA build picks one definition per output. OpenD builds once for every reader, so it keeps all the definitions, in precedence order, as alternatives in the release snapshot. When a page is served, the reader's filter removes the alternatives they don't qualify for and keeps the first that remains.

- Put the most specific definition first and an unconditional one last, as a default. Definitions after an unconditional one can never win.
- If no definition survives for a reader, the element's own content is used, as for an undefined key.
- Conditions on the map entries around a `keydef` count as its conditions.
- The build-time DITAVAL does not choose between alternatives, because it runs for every reader at once. It only removes definitions it excludes.

- In the previewer's staff view, every alternative is shown, separated by a bar and marked with its conditions.

Conditional keys work for variable text, links and images, and for `conkeyref`, so a reader can get a different reusable chunk.

Filter at build time with DITAVAL

Remove content for every reader before it reaches the content store, for example staff-only notes, with a DITAVAL file named in `opend.config.yaml`.

1. Write the DITAVAL file.

```
<val>
  <prop att="audience" val="internal" action="exclude"/>
</val>
```

2. Name it in `opend.config.yaml`.

```
buildFilter: filters/publish.ditaval
```

3. Mark the content.

```
<topicref href="../internal/maintainers.dita" audience="internal"/>
<p audience="internal">Ask in the team channel before changing
this.</p>
```

Results

Excluded content is gone from the release snapshot, so no reader, preview or PDF can ever show it. This site keeps maintainer notes in `internal/` this way.

What to do next

Supported rules:

- `prop` with an `action` of `include`, `exclude`, `flag` or `passthrough`. Only `exclude` removes content; the others keep it.
- The most specific rule wins: attribute and value, then attribute only (no `val`), then a global default (no `att`), then include.
- Flagging styles and `revprop` are ignored.
- The root must be `<val>`, and an unknown action is an error (`invalid-ditaval`).

Any filtering attribute can be filtered at build time, but excluding a `product`, `feature` or `release` value here removes it for *every* reader. To gate readers, rely on their entitlements instead.

Context-sensitive help

Check context help in the help lab

See what every help key and route resolves to for each reader, and click through the real help widget, before you publish.

About this task

Host apps open help by context key or by their current path. The help lab, served by the previewer, lists every app, key and route in the project with the topic it resolves to for the reader you are previewing as.

1. Start the previewer and open `http://127.0.0.1:4000/_opend/help-lab`.
2. Read the table for each app.
   Each key shows the topic it opens. When the reader cannot see the topic the key names, the lab shows where it falls back to, for example `(via parent: tenants)` or `(via default: overview)`. `nothing visible` means the reader gets no help for it at all.
3. Type a host-app path, such as `/tenants/acme`, into **Try a host-app path** and choose **Resolve**.
   The lab shows the topic that path opens for the current reader.
4. Choose **Open in widget** next to a key to open the real help panel.
5. Switch **Preview as** and check again for each kind of reader.

Results

When every key opens a topic each reader can see, publish. To learn how keys, routes and fallback work, see Context-sensitive help in your apps.

Tag topics with context keys

Declare which screen of which app a topic helps with, using DITA's standard `resourceid` element, in the topic or on its map entry.

1. Pick a key for each screen. Keys are words separated by dots, from general to specific.
   For example `tenants` and `tenants.detail`. Dots matter: a key falls back to its parents, so `tenants.detail` opens the `tenants` topic until a more specific one exists. Keys use letters, digits, `_` and `-` between the dots.
2. Add a `resourceid` to the topic's `prolog`, naming the app and the key.

```
<prolog>
  <resourceid appname="opend-console" appid="tenants"/>
  <resourceid appname="opend-console" appid="billing"/>
</prolog>
```

A topic can answer several keys, and several apps. `appname` is the app name your widget passes; `appid` is the key.

3. Or put it on the map entry, in its `topicmeta`. Map entries are checked before topics.

```
<topicref href="../topics/hosting/admin-api.dita">
  <topicmeta><resourceid appname="opend-console" appid="builds"/>
</topicmeta>
</topicref>
```

4. Optional: point one key at different topics for different readers.

```
<topicref href="../topics/notes/release-1.dita" release="1">
  <topicmeta><resourceid appname="console" appid="whats-new"/>
</topicmeta>
</topicref>
<topicref href="../topics/notes/release-2.dita" release="2">
  <topicmeta><resourceid appname="console" appid="whats-new"/>
</topicmeta>
</topicref>
```

Conditions on the `resourceid`, on the elements around it and on the map entry all apply. Each reader gets the first target whose conditions pass and whose topic they can see. Two *unconditional* targets for one key are an error, `duplicate-context`, because the second could never be shown.

5. Optional: put a `resourceid` inside a nested topic to open the page at that section.

Results

A `resourceid` without both `appname` and `appid` is an error, `invalid-resourceid`. Keys on topics the build-time filter removes are dropped. For example, this site tags its admin API reference sections with the `opend-console` keys `repos`, `builds`, `orgs`, `readers` and `audit`.

help-contexts.yaml reference

Maps each app's routes to context keys, so an app can open help by sending its current path, and sets the key to use when nothing more specific matches.

Format

The file lives at the docs root. Part of this site's own file:

```
apps:
  opend-console:
    default: overview
    routes:
      /: overview
      /tenants: tenants
      /tenants/:tenant: tenants.detail
      /tenants/:tenant/**: tenants.detail
      /billing/**: billing
      /editor/**: editor
      /editor/:repo/conflicts: editor.conflicts
```

Field	Meaning
`apps.<name>`	One entry per app, named as in `resourceid appname` and in the widget's `app` option. Letters, digits, `_` and `-`.
`default`	Optional. The key used when the requested key, and every parent of it, has nothing this reader can see.
`routes`	Route pattern to key, in any order. Optional.

Route patterns

Segment	Matches	Example
`tenants`	that word, ignoring case	`/tenants`
`:name`	any one segment	`/tenants/:tenant`
`*`	any one segment	`/files/*`
`**`	zero or more segments; last segment only	`/billing/**`

- Every pattern starts with `/`.
- Query strings, fragments, empty segments and trailing slashes in the path are ignored, and percent-encoded segments are decoded, so `/Tenants/acme/?tab=users` matches `/tenants/:tenant`.
- A segment containing `*`, `:`, `?` or `#` other than as above is invalid, and so is `**` before the last segment. Both are `invalid-help-contexts` errors.

Which route wins

When several patterns match a path, the most specific wins:

1. the one with more literal segments;
2. then one without `**`;
3. then the longer pattern;
4. then the one earlier in the file.

So for `/editor/docs/conflicts`, `/editor/:repo/conflicts` (two literals) beats `/editor/**` (one literal). For `/tenants/acme`, `/tenants/:tenant` beats `/tenants/:tenant/**`, which has the same literals but a `**`.

Checks

Every route and default must name a key that some topic declares with `resourceid`, or a key with such a parent: `tenants.detail` is fine while only `tenants` exists. Anything else is an `unknown-context` error. A path that matches no route opens the default.

How a context resolves, and its fallback rules

A help request names an app and either a key or a path. OpenD tries the key, then its parents, then the app's default, and only ever opens a topic the reader can see.

1. If the request has a `key`, that is the requested key, and any `path` is ignored. Otherwise the most specific route matching the `path` gives the key. A path that matches no route has no requested key.
2. The requested key is tried, then each parent: `tenants.detail`, then `tenants`.
3. Then the app's `default`, and its parents.
4. For each key, the targets are tried in order, map entries first. A target counts only when all its conditions pass for this reader *and* its topic is visible to them.
5. The first target that counts is opened. If none does, there is no help for this context: the panel says so and offers search, and the JSON API answers 404.

The result records how it was found: `key` (the requested key itself), `parent` or `default`. The help lab shows it, and the JSON API returns it as `via`.

A lookup never reveals a hidden topic

Because hidden targets are skipped, a host app cannot learn about a topic by asking for its key: a reader without the topic simply gets the fallback, exactly as if the key pointed nowhere. The same holds when only part of a key's targets are hidden, which is how one key can open different topics for different releases.

In this site, the `opend-console` keys for the operator console point at an early-access topic. Readers without the `early-access` feature fall back to the overview.

PDF publications

PDF publications

Curated manuals, such as an administrator guide, defined as bookmaps and printed for each reader through the same filter as the website.

A PDF gets downloaded and passed around, so a leak there would be worse than on the web. Each reader's PDF holds only the chapters, topics and paragraphs their release and entitlements allow. Two readers with different products get two different PDFs from the same bookmap.

This site has two: the *OpenD Administrator Guide* and the *OpenD Author Guide*. The links in the page header download the ones you can see.

What a PDF contains

- A cover with the publication title, the site title, the release and the date it was generated.
- A contents list whose entries link to their chapters.
- Every chapter and topic the reader can see, in map order. Each top-level chapter starts on a new page.
- A PDF outline (bookmarks) built from the headings, page numbers in the footer and the title in the header. The PDF is tagged for accessibility.

Where to get one

- In the previewer: `/pdf/<name>.pdf` for the current "Preview as" reader, and `/_opend/publications` for the list.
- On the hosted site: `/pdf/<name>.pdf` for the signed-in reader, linked from every page header.

Next: Define a publication with a bookmap.

Define a publication with a bookmap

List chapters in a bookmap, name it under `publications`, and preview the PDF as each reader.

1. Write the bookmap.

```
<bookmap>
  <booktitle>
    <mainbooktitle>OpenD Administrator Guide</mainbooktitle>
  </booktitle>
  <mapref href="keys.ditamap"/>
  <chapter href="../topics/hosting/architecture.dita"/>
  <chapter navtitle="Self-hosting" product="core">
    <topicref href="../topics/hosting/docker-compose.dita"/>
    <topicref href="../topics/hosting/railway.dita"/>
  </chapter>
  <chapter href="../topics/hosting/admin-api.dita"/>
</bookmap>
```

- A `chapter` with an `href` is a topic; nested `topicref`s follow it. A `chapter` with only a `navtitle` becomes a heading for what is under it. `part` and `appendix` work the same way.
- The bookmap resolves keys in its own keyspace. Pull in the same key map as the site map, or topics only the bookmap uses will report `unresolved-key`.
- A topic the bookmap includes does not have to be in the site map. If it is not, links to it from the site are plain text and validation warns.

2. Add it to `opend.config.yaml`.

```
publications:
  - maps/admin-guide.bookmap
```

The file name is the URL: `admin-guide.bookmap` is served at `/pdf/admin-guide.pdf`. File names must be unique across publications (`duplicate-publication`).

3. Preview it. Start the previewer, choose a reader, and open `/_opend/publications`. Each publication shows a **PDF** link and a **print HTML** link, or "nothing visible to this reader". The print HTML, at `/pdf/admin-guide.html`, shows the layout without starting a browser; the PDF needs Chromium (see PDF rendering and limits).

Results

On the next published build, readers who can see at least one topic of the publication get a link to it in the page header.

Per-reader filtering in PDFs

A publication is filtered with exactly the same rules as the site. Chapter conditions cover everything in the chapter.

- Conditions on a `chapter`, `part` or `topicref` apply to everything under it. The Administrator Guide's self-hosting chapter has `product="core"`, so a managed customer's copy has no such chapter at all.
- Conditions on topics and elements apply as on the site, paragraph by paragraph.
- A heading whose chapters are all hidden is left out, and so is its contents entry.
- A reader who can see nothing in a publication gets a 404, the same as for a publication that does not exist, and no download link.
- Links to topics in the same publication jump within the PDF. Links to other topics point at the docs site when the reader can see them there, and are plain text otherwise.
- Only images in topics the reader can see are embedded.

Who shares a render

On the hosted site a PDF is printed once per release and entitlement set, then cached in the blob store, so readers with the same products, features and version share one file. A new release gets new files; nothing needs clearing. Because one file serves several organizations, the hosted cover never names an organization.

In the previewer, the cover says "Prepared for" the preview reader's `org`, when the reader has one, so you can tell printouts apart. The staff view prints the first alternative of each conditional key.

Theme the site, the panel and PDFs

One stylesheet, named `theme` in `opend.config.yaml`, styles the docs site, the help panel and every PDF.

1. Override the `--opend-*` custom properties in your theme.

```
:root {
  --opend-accent: #1f5fbf;      /* brand colour: links, highlights,
focus */
  --opend-text: #14213d;
  --opend-muted: #52607a;       /* short descriptions, secondary
text */
  --opend-surface: #ffffff;     /* page background */
  --opend-surface-alt: #f2f5fa; /* code, table headers, the open
menu */
  --opend-border: #d9e0ea;
  --opend-warn: #b45309;        /* warning, caution and restriction
notes */
  --opend-danger: #b3261e;      /* danger notes */
  --opend-info: #2457c5;        /* plain notes */
  --opend-cond: #7c3aed;        /* condition markers in the staff
view */
  --opend-font: system-ui, sans-serif;
  --opend-mono: ui-monospace, Menlo, monospace;
}
@media (prefers-color-scheme: dark) {
  :root { --opend-accent: #7fb0ff; --opend-surface: #10151d; --
opend-text: #e6ebf2; }
}
```

The link, hover, focus and highlight colours, headings, code backgrounds and note tints are derived from these, so a theme that sets only `--opend-accent` rebrands the whole site. To fine-tune, set the derived properties too: `--opend-link`, `--opend-link-hover`, `--opend-accent-soft`, `--opend-focus`, `--opend-heading`, `--opend-border-strong`, `--opend-code-bg` and `--opend-on-accent`. Layout properties are `--opend-radius`, `--opend-radius-sm`, `--opend-shadow`, `--opend-measure` (reading width), `--opend-sidebar-width`, `--opend-header-height` and `--opend-layout-max`.

Keep text readable: body text and links need a contrast of at least 4.5:1 against `--opend-surface`, in both colour schemes.

2. Optional: add a logo.

```
logo: theme/logo.svg
```

The image is shown before the site title in the page header. See the `logo` field in opend.config.yaml reference.

3. Optional: add print rules for PDFs.

```
.publication .pub-cover h1 { color: #13795b; }
.publication .pub-chapter h1 { border-bottom: 2px solid #13795b; }
```

The print document's `body` has the class `publication`. Useful hooks are `.pub-cover`, `.pub-toc`, `.pub-chapter` and `.pub-part`. PDFs always print in the light scheme.

4. Optional: choose how PDFs are laid out.

```
pdf:
  template: report
```

Page size, margins, cover, contents and running header and footer come from a PDF template; see Choose a PDF template.

5. Check the result on a page, in the help lab's panel and in the print HTML, at phone and desktop widths and in the light and dark schemes.

Results

The theme loads after the built-in styles, and in PDFs after the print styles, so its rules win. This site's theme is `site/theme/site.css`. Custom rules can use the page's class names, such as `.site-header`, `.toc` (the sidebar on wide screens), `.site-menu-panel` (the panel the **Menu** button opens on narrow screens, with search, the contents and the PDF links), `.opend-nav` (both copies of the contents) and `main`.

> **Restriction:** In PDFs the theme is inlined into a document without a base URL, so `url()` references to fonts or images in it are not loaded. Use fonts installed on the server, or the system font stack.

Staff of a hosted site can also set a theme on the admin console's **Appearance** page, without changing the repository. It loads after this theme, so it wins where both set the same thing; see Change the site's look from the admin console.

Choose a PDF template

A PDF template sets the page size, margins, text size, cover, contents and running header and footer of every publication. Pick one of four built-in templates with `pdf` in `opend.config.yaml`, and change any of its settings.

Built-in templates

Template	What it gives you
`standard`	The default. A4 portrait, normal margins, 10.5 pt text, a title page with the title in the accent colour, contents, each chapter on a new page, the publication title in the header and page numbers in the footer.
`report`	US Letter with wide margins, a banner title page (a full-width band in the accent colour), the site title in the header and `{title} · Release {release}` in the footer.
`compact`	Dense A4 for printing: narrow margins, 9.5 pt text, no title page (a title block opens the first page) and chapters that run on.
`handbook`	Pocket-sized A5 in a serif face, with a plain left-aligned title page, no header and the publication title in the footer.

Settings

Setting	Values	What it does
`template`	`standard`, `report`, `compact`, `handbook`	The built-in template the other settings change. Default `standard`. Every other setting defaults to the template's own value.
`pageSize`	`A4`, `Letter`, `Legal`, `A5`	The paper size.
`orientation`	`portrait`, `landscape`	Which way up the page is.
`margins`	`normal`, `narrow`, `wide`	Page margins: 20/18 mm, 14/12 mm or 26/25 mm (top and bottom / sides).
`fontSize`	`small`, `normal`, `large`	Body text of 9.5, 10.5 or 11.5 points.
`font`	`default`, `system`, `humanist`, `serif`, `rounded`, `mono`	The body font. `default` follows the site theme. Only fonts installed on the server that prints the PDF are used.
`cover`	`classic`, `banner`, `minimal`, `none`	The title page. `none` puts a title block at the top of the first page instead.
`coverNote`	text	An extra line on the cover, such as `Confidential`.
`toc`	`true`, `false`	Print the contents after the cover.
`chapterBreaks`	`true`, `false`	Start every top-level chapter, and the first page after the contents, on a new page.

Setting	Values	What it does
`header`	text	Text at the top of every page; empty for none. `{title}` (the publication), `{site}` (the site title), `{release}` and `{date}` (when the PDF was printed) are filled in.
`footer`	text	Text at the foot of every page, before the page number, with the same placeholders.
`pageNumbers`	`true`, `false`	Print `page / pages` at the foot of every page.

Example

```
pdf:
  template: report
  pageSize: A4          # the report layout, on A4 paper
  coverNote: Confidential: for customers only
  footer: '{title} · {date}'
```

Quote text that starts with `{`: unquoted, YAML reads it as a mapping. A setting the template does not know, or a value it does not allow, is reported as `invalid-config`. Print CSS goes in the project `theme`; see Theme the site, the panel and PDFs.

On a hosted site, staff can choose a template on the admin console's **Appearance** page instead. Saved there, it replaces these settings for every release until it is reset; see Change the site's look from the admin console.

Supported DITA subset

OpenD parses a documented subset of DITA 1.3 itself, in TypeScript, with no Java toolchain. Every element it supports is listed here.

How elements are matched

Elements are matched by name. DTDs are not fetched or applied, so a `DOCTYPE` line is fine but has no effect, and `@class` specialisation is not used. Common HTML character entities such as ` `, `—`, `→` and `™` are understood without a DTD.

An element outside the subset still renders, as a plain block, and validation reports it as an `unsupported-element` warning.

Topic elements

The table lists every element allowed in topics, its group, and the HTML it renders as. It mirrors `SUPPORTED_ELEMENTS` in `packages/dita/src/elements.ts`, and a test fails if the two drift apart.

Element	Group	Renders as
`topic`	topic	article, or section when nested
`concept`	topic	article, or section when nested
`task`	topic	article, or section when nested
`reference`	topic	article, or section when nested
`glossentry`	topic	article, or section when nested
`troubleshooting`	topic	article, or section when nested
`draft-comment`	removed	removed by the build
`required-cleanup`	removed	removed by the build
`prolog`	metadata	kept, not rendered
`titlealts`	metadata	kept, not rendered
`navtitle`	metadata	kept, not rendered
`searchtitle`	metadata	kept, not rendered
`indexterm`	metadata	kept, not rendered
`data`	metadata	kept, not rendered
`data-about`	metadata	kept, not rendered
`foreign`	metadata	kept, not rendered
`unknown`	metadata	kept, not rendered
`colspec`	metadata	kept, not rendered
`resourceid`	metadata	kept, not rendered
`dita`	structure	children only
`title`	structure	heading, figcaption or caption by context

Element	Group	Renders as
`shortdesc`	structure	`p`
`abstract`	structure	`div`
`body`	structure	`div`
`conbody`	structure	`div`
`taskbody`	structure	`div`
`refbody`	structure	`div`
`troublebody`	structure	`div`
`glossterm`	structure	`h1`
`glossdef`	structure	`p`
`glossBody`	structure	`div`
`section`	structure	`section`
`example`	structure	`section`
`bodydiv`	structure	`div`
`sectiondiv`	structure	`div`
`div`	structure	`div`
`related-links`	structure	nav "Related information"
`link`	structure	li with a link
`linktext`	structure	`span`
`linkpool`	structure	`div`
`linklist`	structure	`div`
`desc`	structure	`span`
`p`	blocks	`p`

Element	Group	Renders as
`ul`	blocks	`ul`
`ol`	blocks	`ol`
`li`	blocks	`li`
`sl`	blocks	`ul`
`sli`	blocks	`li`
`dl`	blocks	`dl`
`dlhead`	blocks	`div`
`dthd`	blocks	`dt`
`ddhd`	blocks	`dd`
`dlentry`	blocks	`div`
`dt`	blocks	`dt`
`dd`	blocks	`dd`
`note`	blocks	aside with a label
`codeblock`	blocks	pre > code
`pre`	blocks	`pre`
`lines`	blocks	`div`
`lq`	blocks	`blockquote`
`fig`	blocks	`figure`
`image`	blocks	`img`
`alt`	blocks	`span`
`table`	blocks	`table`
`tgroup`	blocks	children only

Element	Group	Renders as
`thead`	blocks	`thead`
`tbody`	blocks	`tbody`
`row`	blocks	`tr`
`entry`	blocks	td, or th in a header
`simpletable`	blocks	`table`
`sthead`	blocks	thead row
`strow`	blocks	`tr`
`stentry`	blocks	td, or th in a header
`properties`	blocks	`table`
`prophead`	blocks	thead row
`proptypehd`	blocks	`th`
`propvaluehd`	blocks	`th`
`propdeschd`	blocks	`th`
`property`	blocks	`tr`
`proptype`	blocks	`td`
`propvalue`	blocks	`td`
`propdesc`	blocks	`td`
`refsyn`	blocks	`section`
`prereq`	task	`section`
`context`	task	`section`
`steps`	task	`ol`
`steps-unordered`	task	`ul`

Element	Group	Renders as
`steps-informal`	task	`div`
`step`	task	`li`
`cmd`	task	`span`
`info`	task	`div`
`substeps`	task	`ol`
`substep`	task	`li`
`stepxmp`	task	`div`
`stepresult`	task	`div`
`tutorialinfo`	task	`div`
`choices`	task	`ul`
`choice`	task	`li`
`result`	task	`section`
`postreq`	task	`section`
`tasktroubleshooting`	task	`section`
`condition`	troubleshooting	`section`
`troubleSolution`	troubleshooting	`div`
`cause`	troubleshooting	`section`
`remedy`	troubleshooting	`section`
`ph`	inline	`span`
`b`	inline	`b`
`i`	inline	`i`
`u`	inline	`u`

Element	Group	Renders as
`sup`	inline	`sup`
`sub`	inline	`sub`
`tt`	inline	`code`
`line-through`	inline	`s`
`overline`	inline	`span`
`keyword`	inline	`span`
`term`	inline	`em`
`q`	inline	`q`
`xref`	inline	a, or plain text when hidden
`codeph`	inline	`code`
`filepath`	inline	`code`
`cmdname`	inline	`code`
`varname`	inline	`var`
`userinput`	inline	`kbd`
`systemoutput`	inline	`samp`
`uicontrol`	inline	`span`
`wintitle`	inline	`span`
`menucascade`	inline	span with > separators
`option`	inline	`code`
`parmname`	inline	`code`
`apiname`	inline	`code`
`msgph`	inline	`samp`

Element	Group	Renders as
`synph`	inline	`code`
`text`	inline	`span`
`tm`	inline	span with a ™, ® or ℠ mark
`cite`	inline	`cite`
`fn`	inline	span in parentheses
`abbreviated-form`	inline	`span`

Map and bookmap elements

Maps may contain `map`, `bookmap`, `title`, `topicmeta`, `navtitle`, `linktext`, `shortdesc`, `keywords`, `keyword`, `searchtitle`, `data`, `topicref`, `topichead`, `topicgroup`, `keydef`, `mapref`, `ditavalref`, `anchor`, `anchorref`, `navref`, `chapter`, `part`, `appendix`, `appendices`, `frontmatter`, `backmatter`, `notices`, `booktitle`, `mainbooktitle`, `booktitlealt`, `bookmeta`, `booklists`, `toc`, `indexlist`, `glossarylist`, `tablelist`, `figurelist`, `reltable`, `relheader`, `relcolspec`, `relrow`, `relcell`, `glossref` and `resourceid`.

Of these, navigation is built from `topicref`, `topichead`, `topicgroup`, `mapref`, `chapter`, `part`, `appendix`, `frontmatter` and `backmatter`; keys come from any element with `keys`. Relationship tables, `anchor`, `navref`, `ditavalref` and the book lists are accepted and ignored. See Topics, maps and bookmaps.

Not supported yet

- DTD processing and `@class` specialisation.
- `conrefend` ranges and `conaction`. They produce an `unsupported-conref` warning, and only the start element is pulled.
- Relationship tables, key scopes and branch filtering with `ditavalref`.
- Topic references to external or non-DITA resources (`unsupported-topicref`).
- DITAVAL flagging styles: `flag` includes the content without styling it.

Validation diagnostics

Every problem code `opend validate`, the previewer, the web editor and the hosted build report, what it means and how to fix it.

Reading a diagnostic

```
topics/install.dita:14:9: error [broken-link] link target not found:
setup.dita
```

That is the file, line and column, the severity, the code and a message. An **error** fails validation and stops a hosted build from publishing. A **warning** does not, unless you validate with `--strict`. Codes are stable, so tools can filter on them.

Project files

Code	Severity	Meaning	Fix
`missing-config`	error	`opend.config.yaml` is not in the folder you validated.	Pass the docs root, or create the file.
`invalid-config`	error	The config is not valid YAML, or a field breaks its rules. The hosted build also reports it when `release` is not a version number.	Follow the message, which names the field. See opend.config.yaml reference.
`missing-ditaval`	error	The `buildFilter` file does not exist.	Fix the path, which is relative to the docs root.
`invalid-ditaval`	error	The DITAVAL root is not `<val>`, or a `prop` has an unknown `action`.	Use `include`, `exclude`, `flag` or `passthrough`.
`missing-theme`	error	The `theme` file does not exist.	Fix the path.
`missing-logo`	error	The `logo` file does not exist.	Fix the path.
`duplicate-publication`	error	Two publications have the same file name, so only one could be downloaded at `/pdf/<name>.pdf`.	Rename one bookmap.
`invalid-preview-contexts`	error	`preview-contexts.yaml` is not valid YAML, or a reader is malformed, for example without a `version`. Reported by the CLI only.	See preview-contexts.yaml reference.

Code	Severity	Meaning	Fix
`xml-syntax`	error	A topic, map or DITAVAL file is not well-formed XML.	Look at the line and column: an unclosed tag, a stray `&` or `<` (write `&` and `<`, or use CDATA), or an unknown entity.

Maps and topics

Code	Severity	Meaning	Fix
`missing-map`	error	The root map, or a map a `mapref` points at, does not exist, or a `mapref` has no usable `href`.	Fix the path.
`map-cycle`	error	Maps reference each other in a loop. The message shows the chain.	Remove one `mapref`.
`missing-topic`	error	A `topicref` points at a file that does not exist, or that is not a `.dita` or `.xml` file.	Fix the `href`.
`unsupported-topicref`	warning	A `topicref` points at an external URL, or has a `format` other than `dita`. It is skipped.	Link to external pages from a topic with `xref` instead.
`empty-topicref`	warning	A navigation entry has neither an `href` nor a `navtitle`, so it has nothing to show.	Add one, or use `topicgroup` to group without a heading.
`not-a-topic`	error	A file a map references has no topic root element.	Make the root a topic type, or point the map elsewhere.
`missing-id`	error	A topic's root element has no `id`.	Add one; references use it.
`duplicate-id`	error	Two elements in one topic share an `id`.	Rename one, and update references to it.
`unsupported-element`	warning	An element is outside the supported subset. It renders as a plain block.	Use a supported element: see Supported DITA subset.

Code	Severity	Meaning	Fix
`conflict-markers`	error	The web editor refused a save, or a conflict resolution, because the text still has conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`), such as those in the hand-edit box when you resolve a conflict.	Keep the text you want from each side and delete the marker lines, then save again.
`unreferenced-topic`	warning	A topic file is in no map, and no published topic reuses content from it. It is usually forgotten.	Add it to a map, reuse a chunk from it, or delete it.

Conditions

Code	Severity	Meaning	Fix
`undeclared-value`	error	A filtering attribute has a value that is not declared in `opend.config.yaml`, or the attribute has no declared values at all. Readers never see that content.	Fix the typo, or declare the value under `attributes`.
`invalid-release`	error	A `release` token is not a version or range. Readers never see that content.	Use `1.4`, `2.0..`, `..1.9` or `1.2..1.9`. See Release range grammar.

Links, images, keys and reuse

Code	Severity	Meaning	Fix
`broken-link`	error	A link points outside the docs root, or at a file, topic or element that does not exist.	Fix the `href`. Same-file links take `#topic-id/element-id`.
`link-to-unpublished`	warning	A link points at a topic that is not in the site navigation, so every reader sees plain text.	Add the target to the site map (with `toc="no"` to keep it unlisted), or remove the link.
`missing-image`	error	An image file, or the file a key for an image names, does not exist.	Fix the path, relative to the topic or map.
`unresolved-key`	warning for `keyref`, error for `conkeyref`	The key is not defined in the keyspace of the map publishing this topic. A `keyref` falls back to the element's own content; a `conkeyref` pulls nothing.	Define the key, or pull the shared key map into this map or bookmap with a `mapref`.
`missing-conref`	error	A `conref` is not of the form `file#topic/element`, its file, topic or element does not exist, or a `conkeyref` key has no local `href`.	Check the ids in the chunks file.
`conref-cycle`	error	Content references pull each other in a loop. The message shows the chain.	Break the loop.

Code	Severity	Meaning	Fix
`conref-type-mismatch`	warning	An element pulls a chunk of a different element type, such as a `p` pulling a `note`. The chunk is used as it is.	Make the referencing element the same type as the chunk.
`unsupported-conref`	warning	The element uses `conrefend` or `conaction`. Only the start element is pulled.	Reference each element separately, or wrap the range in one element with an id.

Context-sensitive help

Code	Severity	Meaning	Fix
`invalid-resourceid`	error	A `resourceid` lacks `appname` (the host app) or `appid` (the context key).	Give both.
`duplicate-context`	error	Two topics claim the same app and key without conditions, so the second could never be shown.	Add conditions to tell them apart, or use a different key.
`invalid-help-contexts`	error	`help-contexts.yaml` is not valid YAML, does not match the expected shape, or has a bad route pattern.	See the help-contexts.yaml reference.
`unknown-context`	error	A route or default in `help-contexts.yaml` maps to a key that no topic declares, nor any parent of it.	Add a `resourceid` for the key, or fix the key.

Where diagnostics appear

- In the terminal, from `opend validate` and `opend preview`.

- In the previewer's banner, and at `/_opend/diagnostics.json`.
- As GitHub workflow annotations with `--format github`.
- On the hosted platform: a failing push shows a failed `opend/build` commit status with the first errors, and the full list is in the build's record from the admin API.