Documentation starter pack

© 2025 Canonical Ltd.

All rights reserved.

Contents

1 Tutorials **3**
- 1.1 Set up the documentation repository . . . 3
- 1.2 Set up automated testing for a Sphinx-based tutorial . . . 4

2 How-to guides **12**
- 2.1 Set up Read the Docs . . . 12
- 2.2 Customise the setup . . . 13
- 2.3 Set up Sphinx sitemaps . . . 17
- 2.4 Customise PDF output . . . 20
- 2.5 Migrate from the pre-extension starter pack . . . 26
- 2.6 Update the documentation . . . 30
- 2.7 How to contribute . . . 35

3 Reference **40**
- 3.1 Contents . . . 40

4 In this documentation **78**

The documentation starter pack helps you to quickly set up, build, and publish documentation with Sphinx.

It contains common styling and configuration through the Canonical Sphinx[1] extension, supports both reST (reStructuredText) and Markdown, and includes automatic documentation checks.

[1] https://github.com/canonical/canonical-sphinx

1. Tutorials

Tutorials for using Sphinx and Read the Docs to host and test your documentation.

1.1. Set up the documentation repository

This page contains a short guide on how to set up and use the starter pack.

1.1.1. Initial setup

If you're starting a new project, clone the starter pack repository[2] and begin your project from there.

If you already have a project, download the following files from the repository and copy them into your existing repository:

- the entire `docs` directory
- `.readthedocs.yaml` (configuration for the building on Read the Docs)
- `.wokeignore` (configuration for the Woke tool)
- the entire `.github/workflows` directory

After cloning or copying the starter pack, you **must** delete `.github/workflows/test-starter-pack.yml` from your repository, as this workflow is specific to testing the starter pack itself and should not be included in your project. Optionally, you can also delete `.github/workflows/sphinx-python-dependency-build-checks.yml` and `.github/workflows/markdown-style-checks.yml` if you do not need these workflows in your project.

1.1.2. Build and run the local server

Building the documentation requires `make`, `python3`, `python3-venv`, `python3-pip`.

In `docs`, run:

```
make run
```

This creates and activates a virtual environment in `docs/.sphinx/venv`, builds the documentation and serves it at `http://127.0.0.1:8000/`.

The server watches the source files, including `docs/conf.py`, and rebuilds automatically on changes.

The landing page is `docs/index.rst`. Other pages are under one of the sub-directories under `docs/`.

1.1.3. Configure settings

Work through the settings in `docs/conf.py`. Most parameters can be left with the default values as they can be changed later. *Customise the setup* (page 13) contains further guidance.

1.1.4. Pre-commit hooks (optional)

Use pre-commit[3] hooks with the starter pack to automate checks like spelling and inclusive language.

The starter pack includes a ready-to-use `.pre-commit-config.yaml` file under `docs/.sphinx/`:

[2] https://github.com/canonical/sphinx-docs-starter-pack
[3] https://pre-commit.com/

```yaml
repos:
  - repo: local
    hooks:
      - id: make-spelling
        name: Run make spelling
        entry: make -C docs spelling
        language: system
        pass_filenames: false
        files: ^docs/.*\.(rst|md|txt)$

      - id: make-linkcheck
        name: Run make linkcheck
        entry: make -C docs linkcheck
        language: system
        pass_filenames: false
        files: ^docs/.*\.(rst|md|txt)$

      - id: make-woke
        name: Run make woke
        entry: make -C docs woke
        language: system
        pass_filenames: false
        files: ^docs/.*\.(rst|md|txt)$
```

For a new project, copy this file to your project's root directory; for an existing project using `pre-commit`, add these hooks to your configuration.

To apply the configuration, install the starter pack hooks, for instance:

```
pre-commit install --config docs/.sphinx/.pre-commit-config.yaml
```

After that, you should see the checks running with every commit:

```
git commit -m 'add spelling errors'

Run make spelling.......................................................Failed
Run make linkcheck......................................................Passed
Run make woke...........................................................Passed
```

1.2. Set up automated testing for a Sphinx-based tutorial

When crafting a tutorial, you may want to check that all the steps run smoothly and as expected. You can accomplish this with **Spread** – a system-wide test distribution that automatically assigns jobs to run tests in GitHub CI workflows. Using Spread, you can create a **Spread test** that runs through all the steps in your tutorial and outputs any failures that may occur. And with Sphinx-based directives, you can guarantee that your tutorial uses the same commands that Spread is testing.

Note:

Creating a Spread test for your tutorial is not required to use the Sphinx starter pack; this is an optional capability.

What you'll need

- Multipass[4] installed on your machine
- Spread[5] installed on your machine

What you'll do

- Create a "Hello, world" Spread test called `example_tutorial`
- Run the Spread test locally on your machine using Multipass

1.2.1. Create the Spread test materials

On your local machine, create a new directory called `spread_test_example` and change into it. This is the root directory of your example project.

Inside the `spread_test_example` directory, create the `tests` directory using `mkdir tests` and change into it. This directory can hold materials for multiple Spread tests.

Under the `tests` directory, create a new directory `example_tutorial` to store the files for a "Hello, world" Spread test. This test consists of two files:

- A bash script that echoes "Hello, world" to the terminal.
- A `task.yaml` file that contains all the commands you want the Spread test to run.

In `spread_test_example/tests/example_tutorial`, run this command to create a file named `example_bash_script.sh`:

```
echo -e '#! /usr/bin/bash\n\necho "Hello, world!"' > example_bash_script.sh
```

Now let's create a `task.yaml` file. This file holds all the commands the user will run in your tutorial.

In `spread_test_example/tests/example_tutorial`, paste the following contents into a new file `task.yaml`:

Listing 1: ~/spread_test_example/tests/example_tutorial/task.yaml

```
###########################################
# IMPORTANT
# Comments matter!
# The docs use the wrapping comments as
# markers for including said instructions
# as snippets in the docs.
###########################################
summary: Example tutorial

kill-timeout: 5m
```

(continues on next page)

[4] https://multipass.run/install
[5] https://github.com/canonical/spread

(continued from previous page)

```
execute: |
  # [docs:make-bash-executable]
  chmod +x example_bash_script.sh
  # [docs:make-bash-executable-end]

  # [docs:execute-bash-script]
  bash example_bash_script.sh
  # [docs:execute-bash-script-end]
```

The `summary` section contains a brief description of your tutorial, and the `execute` section contains all the commands that your tutorial uses. The `kill-timeout` option has a default of 10 minutes and doesn't need to be included if your test will complete in that time frame.

By wrapping commands with comments in the form of `# [docs:example-wrapping-command]` and `# [docs-example-wrapping-command-end]`, we can include the exact commands from `task.yaml` in the tutorial file.

1.2.2. Create the tutorial file

Now we have everything we need to create the tutorial file itself. ReStructuredText (.rst)[6] is used for the tutorial file format; MyST-Markdown[7] can also be used.

In `spread_test_example/tests/example_tutorial`, create a text file named `example_tutorial.rst`. To add a title for your tutorial, copy the block below to this file.

Listing 2: ~/spread_test_example/tests/example_tutorial/example_tutorial.rst

```
Demonstrate Spread tests capabilities with a "Hello, world" script
==================================================================
```

In this file, we can use Sphinx's `literalinclude` directives to feed the Spread test materials directly into our tutorial. This way, we guarantee that the Spread test is testing the exact commands that appear in the tutorial.

Let's start with the bash script. In the mock tutorial, we want the the reader to create the file themselves, so let's use that language in `example_tutorial.rst` when we include the script. Add the following text below the title:

Listing 3: ~/spread_test_example/tests/example_tutorial/example_tutorial.rst

```
Demonstrate Spread tests capabilities with a "Hello, world" script
==================================================================

Create a new file ``example_bash_script.sh`` with the following contents:

.. literalinclude:: example_bash_script.sh
   :language: bash
```

Here, we specified that the language of the script is `bash`. Since our tutorial file and the example bash script are located in the same directory, we don't need to specify where the script is located when we use `literalinclude`.

[6] https://www.sphinx-doc.org/en/master/usage/restructuredtext
[7] https://myst-parser.readthedocs.io/en/latest

At the end of the `example_tutorial.rst` file, insert the two commands that appear in our `task.yaml` file, again using the `literalinclude` directive:

Listing 4: ~/spread_test_example/tests/example_tutorial/example_tutorial.rst

```
Demonstrate Spread tests capabilities with a "Hello, world" script
==================================================================

Create a new file ``example_bash_script.sh`` with the following contents:

.. literalinclude:: example_bash_script.sh
    :language: bash

  Make the script executable:

.. literalinclude:: task.yaml
    :language: bash
    :start-after: [docs:make-bash-executable]
    :end-before: [docs:make-bash-executable-end]
    :dedent: 2

Now execute the script:

.. literalinclude:: task.yaml
    :language: bash
    :start-after: [docs:execute-bash-script]
    :end-before: [docs:execute-bash-script-end]
    :dedent: 2

Congratulations! You have created a "Hello, world" script and executed it!
```

If you were to render the tutorial file using Sphinx, then the page would look like the following:

Demonstrate Spread tests capabilities with a "Hello, world" script

Create a new file `example_bash_script.sh` with the following contents:

```
#! /usr/bin/bash

echo "Hello, world!"
```

Make the script executable:

```
chmod +x example_bash_script.sh
```

Now execute the script:

```
bash example_bash_script.sh
```

Congratulations! You have created a "Hello, world" script and executed it!

1.2.3. Create the Spread test

Now let's create the Spread test file and include our example tutorial. From the `spread_test_example` directory, create the file `spread.yaml` and insert the following contents:

Listing 5: ~/spread_test_example/spread.yaml

```
project: spread_test_example

path: /spread_test_example
```

Note that the `project` name matches the main directory's name, `spread_test_example`. The `path` designates the directory where the Spread materials exist.

Now we need to tell Spread about the `example_tutorial` Spread test. Add the following section to the end of `spread.yaml`:

Listing 6: ~/spread_test_example/spread.yaml

```
project: spread_test_example

path: /spread_test_example

suites:
  tests/:
    summary: example tutorial
    systems:
      - ubuntu-24.04-64
```

The `suites` section is how we tell Spread about the various Spread tests in our project. We tell Spread to look in the `tests` directory for all Spread tests (which it will only find one, `example_tutorial`). We also use the `suites` section to tell Spread about the systems we want Spread

to test. For our mock tutorial, we will use Ubuntu 24.04.

1.2.4. Configure the Spread test to use Multipass

Each job in Spread has a backend, or a way to obtain a machine on which to run your Spread test. The Spread repository[8] contains more information on backends like Google or QEMU, but let's set up Multipass as a backend to run local tests.

Include the following `backends` section of `spread.yaml` between the `path` and `suites` sections:

Listing 7: ~/spread_test_example/spread.yaml

```yaml
project: spread_test_example

path: /spread_test_example

backends:
  multipass:
    type: adhoc
    allocate: |
      multipass_image=24.04
      instance_name="example-multipass-vm"

      # Launch Multipass VM
      multipass launch --cpus 2 --disk 10G --memory 2G --name "${instance_name}" "
${multipass_image}"

      # Enable PasswordAuthentication for root over SSH.
      multipass exec "$instance_name" -- \
        sudo sh -c "echo root:${SPREAD_PASSWORD} | sudo chpasswd"
      multipass exec "$instance_name" -- \
        sudo sh -c \
        "if [ -d /etc/ssh/sshd_config.d/ ]
        then
          echo 'PasswordAuthentication yes' > /etc/ssh/sshd_config.d/10-spread.
conf
          echo 'PermitRootLogin yes' >> /etc/ssh/sshd_config.d/10-spread.conf
        else
          sed -i /etc/ssh/sshd_config -E -e 's/^#?PasswordAuthentication.*/
PasswordAuthentication yes/' -e 's/^#?PermitRootLogin.*/PermitRootLogin yes/'
        fi"
      multipass exec "$instance_name" -- \
        sudo systemctl restart ssh

      # Get the IP from the instance
      ip=$(multipass info --format csv "$instance_name" | tail -1 | cut -d\, -f3)
      ADDRESS "$ip"

    discard: |
      instance_name="example-multipass-vm"
      multipass delete --purge "${instance_name}"
```

(continues on next page)

[8] https://github.com/canonical/spread

(continued from previous page)

```
    systems:
      - ubuntu-24.04-64:
          workers: 1

suites:
  tests/:
    summary: example tutorial
    systems:
      - ubuntu-24.04-64
```

The backends section contains the following sections:

- The backend is designated as `type: adhoc` as we are explicitly scripting the procedure to allocate and discard the Multipass VM.
- In the `allocate` section, we define the image and name of the VM, launch the VM, and then set up the proper SSH permissions so that Spread can log in (via root) into the VM and insert the Spread test. We also must tell Spread about the IP address of the Multipass VM and set the environment variable `ADDRESS`.
- In the `discard` section, we delete the Multipass VM once the Spread test has finished running.

1.2.5. Run the Spread test locally

List all available Spread tests in the code repository:

```
spread --list
```

The terminal should respond with a single line representing the test for `example_tutorial`:

```
user@host:spread_test_example$ spread --list

multipass:ubuntu-24.04-64:tests/example_tutorial
```

Now let's run the Spread test for `example_tutorial`:

```
spread -vv -debug multipass:ubuntu-24.04-64:tests/example_tutorial
```

The test can take several minutes to complete. The `-vv -debug` flags provide useful debugging information as the test runs.

1.2.6. Validate the Spread test results

The terminal will output various messages about allocating the Multipass VM, connecting to the VM, sending the Spread test to the VM and executing the test. If the test is successful, the terminal will output something similar to the following:

```
user@host:spread_test_example$

2025-02-04 16:17:10 Successful tasks: 1
2025-02-04 16:17:10 Aborted tasks: 0
```

Another sign of a successful test is whether the Multipass VM was deleted as expected. We

can check by running `multipass list`, and if the Spread test was successful (and you have no other Multipass VMs created at the time), the terminal should respond with the following:

```
user@host:spread_test_example$ multipass list

No instances found.
```

If the Spread test failed, then the `-debug` flag will open a shell into the Multipass VM so that additional debugging can happen. In that case, the terminal will output something similar to the following:

```
user@host:spread_test_example$

2025-02-04 16:17:10 Starting shell to debug...
2025-02-04 16:17:10 Sending script for multipass:ubuntu-24.04-64
(multipass:ubuntu-24.04-64:tests/example_tutorial):
```

1.2.7. Next steps

Congratulations! You set up the materials needed to run a Spread test locally using Multipass with commands that explicitly appear in a Sphinx-based tutorial. This section provides additional examples of Spread tests:

- Spread tests included in Rockcraft documentation[9]
- Spread tests included in Charmcraft documentation[10]

[9] https://github.com/canonical/rockcraft/tree/main/docs/tutorial/code
[10] https://github.com/canonical/charmcraft/tree/main/docs/tutorial/code

2. How-to guides

These guides will walk you through the processes involving setting up, maintaining, and contributing to the starter pack.

2.1. Set up Read the Docs

For Canonical-specific information on how to set up your documentation on Read the Docs, see the Read the Docs at Canonical[11] and How to publish documentation on Read the Docs[12] guides.

In general, after enabling the starter pack for your documentation, follow these steps to build and publish your documentation on Read the Docs:

1. Make sure your documentation *builds without errors or warnings* (page 31).
2. Log into Read the Docs.
3. In your account settings, navigate to *Connected services* and check that your GitHub account is listed. If it's not listed, add a connection to GitHub. See How to connect your Read the Docs account to your Git provider[13].
4. Use the manual import[14] to create a project.
5. Specify the path to the `.readthedocs.yaml` file for your build. To do this, navigate to *Admin* > *Settings* and specify the path under "Path for `.readthedocs.yaml`".

   For example, if your documentation folder is `docs/`, specify the path as `docs/.readthedocs.yaml`.
6. Update the relative paths in the `.readthedocs.yaml` file to match the structure of your project. You might need to update the file paths specified in the following fields:
   - `job.post_checkout`
   - `sphinx.configuration`
   - `python.install.requirements`

After this initial setup, your documentation should build successfully if your project is hosted from a public repository. If you get any errors, check the build log for indications on what the problem is.

If your project was imported from a private repository, your initial build will fail because Read the Docs won't have access to clone the repository. You need to copy your project's private key from Read the Docs and add it as a deploy key to the repository, then re-run the build in Read the Docs.

2.1.1. Configure the webhook

If you have administrator privileges for the GitHub repository that you are adding, the integration webhook (which is responsible for automatically building the documentation when the repository changes) is created automatically.

[11] https://library.canonical.com/documentation/read-the-docs-at-canonical
[12] https://library.canonical.com/documentation/publish-on-read-the-docs
[13] https://docs.readthedocs.com/platform/stable/guides/connecting-git-account.html
[14] https://readthedocs.com/dashboard/import/manual/

If you don't have administrator privileges, the webhook must be set up by someone who does. The person with administrator privileges must have connected their Read the Docs account to GitHub. See How to connect your Read the Docs account to your Git provider[15].

See How to manually configure a Git repository integration[16] if you want to set up the webhook manually.

2.1.2. Make your documentation public

By default, Read the Docs publishes your documentation for logged-in users only.

To make the documentation public, you must configure the privacy level for each version of the documentation separately. You can do this by navigating to the *Versions* tab and changing the *Privacy Level* for each version.

2.1.3. Enable PR previews

To make Read the Docs automatically build your documentation when a pull request is opened or updated on GitHub, enable PR reviews for your project.

To do so, navigate to *Admin* > *Settings* and select *Build pull requests for this project*.

Read the Docs will then automatically build the documentation for each pull request, and the link to the output will be available as one of the checks in the pull request.

2.2. Customise the setup

Important:

After setting up your repository with the starter pack, you should track the changes made to the starter pack.
Changes to the look and feel, as well as common functionality, will be automatically available through updates to the Canonical Sphinx[17] extension.
Changes to files that are part of the starter pack, for example, *Automatic checks* (page 40), might require you to manually update your repository with the required files. See the starter pack's change log[18] for the most relevant (and of course all breaking) changes.

[17] https://github.com/canonical/canonical-sphinx
[18] https://github.com/canonical/sphinx-docs-starter-pack/wiki/Change-log

Configuration for a starter pack based documentation is set in the `docs/conf.py` Sphinx configuration file.

The starter pack's default configuration is prepared in a way that makes sense for most projects. However, you must set some critical parameters that are unique for your project, like the project's name.

In addition, you can find some optional parameters or add your own configuration parameters to the file.

2.2.1. Required customisation

You must check and update some of the parameters specific to your project. Mandatory parameters are commented with the *TODO* keyword.

The following are some highlights of the available configuration parameters.

[15] https://docs.readthedocs.com/platform/stable/guides/connecting-git-account.html
[16] https://docs.readthedocs.io/en/stable/guides/setup/git-repo-manual.html

Update the project information

Edit the `docs/conf.py` file and update the configuration in the `Project information` section. See the comments in the file for more information about each setting.

Open Graph configuration

When you post a link to your documentation somewhere (for example, on Mattermost or Discourse), it might be shown with a preview. This preview is configured through the Open Graph Protocol () configuration.

If you don't know yet where your documentation will be hosted, you can leave the URL empty. If you do, specify the hosting URL. You can leave the defaults for the website name and the preview image or specify your own.

Adjust the header

The header is the top section of a page template. By default, the starter pack template header contains your product's tag image and name (taken from the `project` setting in the `docs/conf.py` file), a link to your product's page (if available), and a drop-down menu for "More resources".

This configuration is sufficient for many cases but can be further customised. For example, you might want to add links to announcements or videos that are not part of the documentation.

To override the default template header, create a new folder called `_templates` in the same folder as your `conf.py` file. Default templates are pulled from the `canonical-sphinx` extension. Copy its header.html file[19] to your `.sphinx/_templates` folder and edit it as needed.

Finally, find the following line in `docs/conf.py`:

```
#templates_path = ["_templates"]
```

Enable this variable initialisation by removing the # at the beginning:

```
templates_path = ["_templates"]
```

2.2.2. Optional customisation

The starter pack contains several features that you can configure, or turn off if they aren't suitable for your documentation.

Deactivate the feedback button

By default, the starter pack includes a feedback button at the top of each page. This button redirects users to your GitHub issues page, and populates an issue for them with details of the page they were on when they clicked the button.

If your project does not use GitHub issues, set the `github_issues` variable in the `docs/conf.py` file to an empty value to disable both the feedback button and the issue link in the footer.

If you want to deactivate the feedback button, but keep the link in the footer, set `disable_feedback_button` in the `docs/conf.py` file to `True`.

[19] https://github.com/canonical/canonical-sphinx/blob/main/canonical_sphinx/theme/templates/header.html

Configure the contributor display

By default, the starter pack will display a list of contributors at the bottom of each page. This requires the GitHub URL and folder to be configured.

If you want to turn this contributor listing off, you can do so by setting the `display_contributors` variable in the `docs/conf.py` file to `False`.

To configure that only recent contributors are displayed, you can set the `display_contributors_since` variable. It takes any Linux date format (for example, a full date, or an expression like "3 months").

Add redirects

If you rename a source file, its URL will change. To prevent broken links, you should add a redirect from the old URL to the new URL in this case.

You can add redirects in the `redirects` variable in the `docs/conf.py` file.

Configure included extensions

The starter pack includes a set of extensions that are useful for all documentation sets. They are pre-configured as needed, but you can customise their configuration in the `docs/conf.py` file.

The following extensions are included by default:

- `canonical_sphinx`
- `sphinxcontrib.cairosvgconverter`
- `sphinx_last_updated_by_git`

The `canonical_sphinx` extension is required for the starter pack. It automatically enables and sets default configurations for the following extensions:

- `custom-rst-roles`
- `myst_parser`
- `notfound.extension`
- `related-links`
- `sphinx_copybutton`
- `sphinx_design`
- `sphinx_reredirects`
- `sphinx_tabs.tabs`
- `sphinxcontrib.jquery`
- `sphinxext.opengraph`
- `terminal-output`
- `youtube-links`

To add new extensions needed for your documentation set, add them to the `extensions` parameter in `docs/conf.py`.

> **Note:**
>
> If any additional extensions need specific Python packages, ensure they are installed alongside the other requirements by adding them to the `.sphinx/requirements.txt` file.

Add page-specific configuration

You can override some global configuration for specific pages.

For example, you can configure whether to display Previous/Next buttons at the bottom of pages by setting the `sequential_nav` variable in the `docs/conf.py` file.

```
html_context = {
    ...
    "sequential_nav": "both"
}
```

You can then override this default setting for a specific page (for example, to turn off the Previous/Next buttons by default, but display them in a multi-page tutorial).

To do so, add file-wide metadata[20] at the top of a page. See the following examples for how to enable Previous/Next buttons for one page:

reST:

```
:sequential_nav: both

[Page contents]
```

MyST:

```
---
sequential_nav: both
---

[Page contents]
```

Possible values for the `sequential_nav` field are `none`, `prev`, `next`, and `both`. See the `docs/conf.py` file for more information.

Another example for page-specific configuration is the `hide-toc` field (provided by Furo[21]), which can be used to hide the page-internal table of content. See Hiding Contents sidebar[22].

2.2.3. Add your own configuration

Custom configuration parameters for your project can be used to extend or override the common configuration, or to define additional configuration that is not covered by the common `conf.py` file.

The following links can help you with additional configuration:

- Sphinx configuration[23]

[20] https://www.sphinx-doc.org/en/master/usage/restructuredtext/field-lists.html
[21] https://pradyunsg.me/furo/quickstart/
[22] https://pradyunsg.me/furo/customisation/toc/
[23] https://www.sphinx-doc.org/en/master/usage/configuration.html

- Sphinx extensions[24]
- Furo documentation[25] (Furo is the Sphinx theme we use as our base)

If you need additional Python packages for any custom processing you do in your documentation, add them to the `.sphinx/requirements.txt` file.

2.3. Set up Sphinx sitemaps

It is recommended to generate a sitemap for your documentation using the sphinx-sitemap[26] extension.

Read the Docs generated sitemaps

RTD generates a basic sitemap pointing to the index page, and relies on crawlers to index the site. This is sufficient for some projects, but RTD does not generate sitemaps for subprojects.

This means any project under the Ubuntu documentation library project must generate its own sitemap.

2.3.1. Sitemap prerequisites

The standard Starter Pack uses the `dirhtml` builder for Sphinx recipes in the project's Makefile. If your project uses an older version of the Starter Pack or changes the builder, the links generated by the sitemap will be malformed. Either update to the latest version of the Starter Pack or ensure your project's recipes use the `dirhtml` builder, not `html`.

Ensure `sphinx-sitemap` has been added to your `requirements.txt` file.

Add `sphinx_sitemap` to `extensions` in your configuration file (`docs/conf.py`):

```
extensions = ['sphinx_sitemap']
```

2.3.2. Required sitemap configuration

Sphinx Sitemap requires a `html_baseurl` configured for the project in your configuration file. For example, in `docs/conf.py`:

```
html_baseurl = 'https://canonical-starter-pack.readthedocs-hosted.com/'
```

Make sure to include the trailing slash (/) to avoid errors in the concatenated URLs in the sitemap.

Note:

Sitemap configuration is included in the Starter pack's default configuration file[27].

[27] https://github.com/canonical/sphinx-docs-starter-pack/blob/a489ae041f6cebb7948fdf21b996e8c67d636a83/docs/conf.py#L176

2.3.3. URL configuration

Sphinx sitemap uses a configurable URL scheme to set language and version options for your documentation. If you have no languages and no versions in your URL, add the following to your `conf.py` file:

[24] https://www.sphinx-doc.org/en/master/usage/extensions/index.html
[25] https://pradyunsg.me/furo/quickstart/
[26] https://sphinx-sitemap.readthedocs.io/en/latest/index.html

```
sitemap_url_scheme = "{link}"
```

To add versioning, this can be done manually, or you can read the version from the RTD instance. To implement a manual version:

```
sitemap_url_scheme = "<version>/{link}"
```

Or, if the version is set with the `version` key in your configuration file:

```
sitemap_url_scheme = "{version}{link}"
```

To read from the provided RTD environment variable:

```
if 'READTHEDOCS_VERSION' in os.environ:
    version = os.environ["READTHEDOCS_VERSION"]
    sitemap_url_scheme = '{version}{link}'
else:
    sitemap_url_scheme = 'MANUAL/{link}'
```

Note:

If you are implementing a sitemap on an RTD instance that is not a subproject, and it uses `{link}` for the `sitemap_url_scheme`, RTD will replace your sitemap with their own. This is a known bug. The only current workaround is to use a different sitemap name[28] and a custom `robots.txt` pointing to it.

[28] https://sphinx-sitemap.readthedocs.io/en/latest/advanced-configuration.html#changing-the-filename

2.3.4. `lastmod` configuration

As of version 2.7.0, the sitemap extension supports adding a `lastmod` date. Make sure that your configuration file has:

```
sitemap_show_lastmod = True
```

2.3.5. Exclude pages

Pages can be excluded from the sitemap by adding them to *sitemap_excludes* in `docs/conf.py`:

```
sitemap_excludes = [
    "genindex/",
    "404/",
    "search/",
]
```

2.3.6. Validating your sitemap

A sitemap will be available at different locations, depending on how it is generated.

Read the Docs generated sitemaps are available at the base domain of a project, while sitemaps generated with this extension will be placed in the base of the URL schema used.

For example, two sitemaps are generated for the Sphinx sitemap's documentation as it is hosted on RTD:

- The first is generated by RTD and is available at the root of the domain: https://sphinx-sitemap.readthedocs.io/sitemap.xml

- The second is generated by the *sphinx-sitemap* extension and is available at the base of the URL schema used by the RTD instance: https://sphinx-sitemap.readthedocs.io/en/latest/sitemap.xml

How to specify a sitemap

A *robots.txt* file dictates which sitemap is used to index a website. You can use a custom *robots.txt* file by creating your own and adding it to *html_static_path* in your configuration file. An example can be found in the Ubuntu documentation library[29] project.

2.3.7. Supporting multiple versions

Sphinx sitemap does not support multiple versions by default. Configuring your versioned documentation to use an appropriate version may be sufficient, as Google and other automated tools will crawl websites for the purposes of indexing. However, if you want comprehensive sitemaps for your documentation and all its versions, you will need to deploy your own `robots.txt` file and sitemap index.

For instance, using the starter pack as an example, with three versions (1.0, 2.0, 3.0), using the RTD URL schema of `{version}{link}`:

1. Ensure each version of your documentation has a sitemap generated by this extension with the appropriate version.
2. Create a `robots.txt` file, in the same directory as your configuration file, pointing to a custom `sitemapindex.xml` file:

```
User-agent: *

Disallow: # Allow everything

Sitemap: https://canonical-starter-pack.readthedocs-hosted.com/latest/
sitemapindex.xml
```

3. Create a `sitemapindex.xml` file, in the same directory as your configuration file, which points to the sitemap files of each of your documentation sets:

```
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
<sitemap>
<loc>https://canonical-starter-pack.readthedocs-hosted.com/latest/sitemap.xml
</loc>
</sitemap>
<sitemap>
<loc>https://canonical-starter-pack.readthedocs-hosted.com/3.0/sitemap.xml</
loc>
</sitemap>
<sitemap>
<loc>https://canonical-starter-pack.readthedocs-hosted.com/2.0/sitemap.xml</
loc>
</sitemap>
<sitemap>
<loc>https://canonical-starter-pack.readthedocs-hosted.com/1.0/sitemap.xml</
loc>
```

(continues on next page)

[29] https://github.com/canonical/ubuntu-documentation-library

(continued from previous page)

```
</sitemap>
</sitemapindex>
```

4. Add `robots.txt` and `sitemapindex.xml` to your configuration file:

```
html_extra_path = ["sitemapindex.xml", "robots.txt"]
```

Note:

You may want to automate the generation of the `sitemapindex.xml` file. To see how this is done for the Ubuntu documentation library project, which generates a sitemap containing subproject sitemaps, see the script here[30].

[30] https://github.com/canonical/ubuntu-documentation-library/blob/main/scripts/generate_sitemap.py

This will provide a *sitemapindex.xml* file which points to the *sphinx-sitemap* generated sitemap for each version.

2.4. Customise PDF output

2.4.1. Overview

The starter pack supports PDF output via LaTeX using the `make pdf` command. This build process relies on system packages, Sphinx configurations, and a LaTeX template from the `canonical-sphinx` extension.

Customising PDF output involves two levels of configuration:

- **Sphinx configuration**: built-in options for configuring LaTeX build process in `conf.py`, for example: the engine used to generate the PDF, output file name, and input file paths.
- **LaTeX configuration**: the LaTeX packages, styling, and configuration for the PDF output, set through the `latex_elements`[31] dictionary in the project `conf.py`. In the starter-pack, a default set of LaTeX elements is provided by the `canonical-sphinx` extension. Changing the LaTeX configuration requires overriding the default values loaded from the extension.

This guide covers common practices and tips for customising PDF output from your documentation project using the starter pack and the `canonical-sphinx` extension.

For basic instructions about building the PDF, see *Build and preview* (page 31).

2.4.2. Configure document properties

The `latex_documents`[32] variable in the Sphinx `conf.py` file controls properties of the intermediate LaTeX file and the final PDF output. The values are defined in a tuple of six elements:

```
latex_documents = [
    (
        'startdocname',    # Root document (e.g., 'index' or 'pdf-index')
        'targetname.tex', # Output LaTeX file name (no spaces)
        'title',           # Document title (can be empty to use the root doc's
```

(continues on next page)

[31] https://www.sphinx-doc.org/en/master/latex.html#the-latex-elements-configuration-setting
[32] https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-latex_documents

(continued from previous page)

```
title)
        'author',          # Author name(s).
        'theme',           # Document type: 'manual' or 'howto'
        True,              # toctree_only: if True, only include docs in toctree
    ),
]
```

- `startdocname`: The root document for the PDF (without the .rst extension).
- `targetname`: The filename for the generated LaTeX source file. The PDF file name is derived from this filename with the `.pdf` extension. Blank space characters are not allowed.
- `title`: The title for the PDF document on the cover page.
- `author`: The author(s) of the document. Use `\\and` to separate multiple authors.
- `theme`: Either `manual` or `howto`.
  - `manual`: This is the default and is intended for comprehensive, book-style documentation. It produces a PDF with chapters, sections, and a table of contents. Use this for user guides, reference manuals, or any documentation that should be structured as a book.
  - `howto`: This type is for shorter, task-oriented documents. It produces a simpler PDF without chapters, and is best for single-topic guides or tutorials.
- `toctree_only`: Boolean. If set to True, the `startdocname` document itself is not included in the output, only the documents referenced by the `toctree` directive are included. This is useful for creating a PDF-specific index file.

For more details, see latex_documents[33] in the Sphinx documentation.

Change PDF document filename

By default, the PDF output filename is derived from the `project` name in the `conf.py` file: lowercase characters with blank spaces removed.

To override the filename, update the second element (`targetname`) of the `latex_documents` tuple in `conf.py`. The following example shows how to replace blank spaces in the project name with underscores:

```
latex_documents = [
    (
        'index',
        f"{project.replace(' ', '_')}.tex",
        '',
        'Author Name',
        'manual',
        True,
    ),
]
```

[33] https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-latex_documents

Change PDF document title

By default, the PDF title on the cover page comes from the title of the main index document. To override it, update the third element (title) of the `latex_documents` tuple in `conf.py`. Use an empty string (`''`) to keep the default behaviour.

Use a different index document for PDF builds

Because the PDF output has a different usage and structure from the HTML output, it is sometimes useful to create a PDF-specific index document. For example, you may want to create a PDF-specific index file that includes only a subset of the pages in the HTML index.

To use a different index document for PDF builds:

1. Create a PDF-specific index document, for example, `pdf-index.rst`.
2. Update the first element (`startdocname`) of the `latex_documents` tuple in `conf.py` to point to the new index document.
3. Set the last element (`toctree_only`) of the `latex_documents` tuple in `conf.py` to `False` to ensure only referenced documents are included in the PDF output.
4. Exclude the PDF-specific index document from the HTML build. This is done by changing the `exclude_patterns` list in `conf.py`:

```
# Identify the Sphinx builder being used
if '-b' in sys.argv:
   builder = sys.argv[sys.argv.index('-b') + 1]
elif '-M' in sys.argv:
   builder = sys.argv[sys.argv.index('-M') + 1]
else:
   builder = 'html'  # default builder

# Exclude the PDF-specific index from the HTML build
if builder in ['html', 'dirhtml']:
   exclude_patterns.append('pdf-index.rst')
```

Check both the HTML and PDF outputs to confirm that different index documents are used for each output.

Note:

The order and hierarchy of your `toctree` entries determine the chapters and sections in the PDF.
Any headings placed before the main `toctree` in your root document will cause all referenced documents to be nested under that heading in the PDF. To avoid this, do not add extra headings before the `toctree`.

2.4.3. Override the LaTeX template

The LaTeX template is a text file in the `canonical-sphinx` extension that provides the default styling and layout of the PDF document. The template contains a Python dictionary of LaTeX elements, which will be imported by Sphinx when the PDF is built.

Any additions or changes to the default settings of LaTeX elements in the PDF document requires overriding the default template.

1. Download the default template file latex_elements_template.txt[34] from the `canonical/canonical-sphinx` GitHub repository, and save it to your documentation project directory. For example, at `.sphinx/latex_elements_custom.txt`.
2. In the Python dictionary, add or modify the LaTeX elements you want to change. Details of changing the dictionary are covered in the sub-sections below.
3. In your project `conf.py`, add or update the `latex_elements` dictionary to load the local override of the LaTeX template. Change the file path to the location of your local override file.

```
# Replace with the path to your local override file
latex_elements_file = ".sphinx/latex_elements_custom.txt"

with open(latex_elements_file, "rt") as file:
   latex_config = file.read()
   if latex_elements == {}:
      latex_elements = ast.literal_eval(latex_config)
```

Warning:

Defining other settings directly in `latex_elements` will override the values loaded from the template file or your local file.

Add more LaTeX packages to the preamble

You can use two methods to add additional LaTeX packages to the preamble:

- Add the `extrapackages` key in your local template file:

  ```
  {
    ...
    'extrapackages': r'\usepackage{packagename}',
    ...
  }
  ```

- Modify the values of the `preamble` key in your local template file. This is more flexible for adding LaTeX configurations and commands to the preamble.

Note:

The format of the element values is a multi-line string, so use a raw string with the `r` prefix.

Remove the table of contents

For a short, compact document where navigation is not needed, you may want to remove the table of contents from the PDF output.

To do this, provide a local copy of the default template file, and add a new key `tableofcontents` with an empty string as the value:

[34] https://github.com/canonical/canonical-sphinx/blob/main/canonical_sphinx/theme/PDF/latex_elements_template.txt

```
{
    ...
    'tableofcontents': '',
    ...
}
```

Include images or other assets

If the local template requires additional images or other assets, for example, a custom title page or header, the file paths must be added to the Sphinx `conf.py` file to be included in the PDF build.

Provide a `latex_additional_files` variable in `conf.py` as a list of file paths to the additional assets. If the variable already exists, add the new file paths to the list. The paths should be relative to the `conf.py` file.

```
# path relative to the conf.py file
latex_additional_files = [
    'path/to/image.svg',
    'path/to/other-asset.pdf',
]
```

> **Note:**
>
> For better quality in the PDF output, it is recommended to use vector images (like SVG or PDF) rather than raster images (like PNG or JPEG). Raster images may lose quality when scaled up in the PDF.
> Do not use `.tex` as suffix, otherwise the file is processed as source files for the PDF build process. Instead, use `.tex.txt` or `.sty` to avoid conflicts with the LaTeX build process.

Use landscape layout

The PDF output uses portrait orientation by default. To use landscape orientation, you need to add more packages to the LaTeX preamble and use a specific LaTeX environment to rotate the content.

1. Add the `extrapackages` key to your local template file, and set the value to the `pdflscape` package:

   ```
   {
       ...
       'extrapackages': r'\usepackage{pdflscape}',
       ...
   }
   ```

   > **Note:**
   >
   > The format of the element values is a multi-line string, so use a raw string with the `r` prefix.

2. Use the landscape environment in your documentation source file, and only in the PDF output.

   Wherever you want a section (such as a wide table or figure) to appear in the landscape view, use the `.. raw:: latex` directive to include raw LaTeX code that opens and closes the landscape environment. Only the content between `\begin{landscape}` and `\end{landscape}` will be rotated:

```
.. only:: latex

   .. raw:: latex

      \begin{landscape}

.. list-table:: Example of a landscape table
   :header-rows: 1

   * - Column 1
     - Column 2
     - Column 3
   * - Data 1
     - Data 2
     - Data 3

.. only:: latex

   .. raw:: latex

      \end{landscape}
```

2.4.4. Check PDF build log files

If you encounter an issue that requires further debugging, check the PDF build logs for more detailed error messages. The full logs are generated in the `_build/latex` output directory, and then cleaned up after the build completes.

To temporarily save the log files for debugging:

1. Open the `Makefile` and locate the `pdf` target. Disable the cleanup step by commenting out the `@rm -r $(BUILDDIR)/latex` line.
2. Run the `make pdf` again.
3. Navigate to the output directory `_build/latex` and check the `*.log` and `*.tex` files.
4. After debugging, restore the cleanup step by uncommenting the same line.

> **Warning:**
>
> Keeping the build log files from the previous build might cause conflicts with the current build

2.4.5. Related

- *Build and preview* (page 31)
- *Customise the setup* (page 13)

2.5. Migrate from the pre-extension starter pack

This guide outlines the steps required to migrate a documentation project from the legacy Sphinx Documentation Starter Pack (*pre-extension* version) to the latest version that adopts the `canonical-sphinx` Sphinx extension.

The extension-based documentation starter pack provides a set of features and configurations that are common across Canonical documentation projects. Key components, such as configuration and styling, are loaded as an add-on to your documentation project. It can significantly reduce maintenance concerns when managing your documentation.

2.5.1. Update to the last pre-extension version

To ensure a smooth migration, update your documentation project to use the last pre-extension version of the Sphinx Documentation Starter Pack. This update ensures that your project is using the latest features and configurations available, minimising the changes required during the migration.

You can find the release tag and branch for this version in the following links:

- pre-extension branch[35]
- pre-extension release tag[36]

2.5.2. Set up a new project

1. Back up all existing files in your original documentation project. For example, you can rename the original `docs/` folder to `docs_backup/`.

   > **Warning:**
   >
   > If you proceed in the same directory, the following steps will overwrite some of the configuration files in the original project.

2. Follow the steps in the *Initial setup* (page 3) guide to initialise an empty project with the extension-based starter pack, at the original file path.

3. Ensure the following files are at the root of your repository:
   - `.github/workflows/*`

4. Ensure the following files are moved to their original paths in the project. These files are defaulted to the repository root, but may have be changed upon project needs:
   - `.gitignore`
   - `.readthedocs.yml`

5. Validate the project setup locally by running `make run` in the new project directory.

[35] https://github.com/canonical/sphinx-docs-starter-pack/blob/pre-extension
[36] https://github.com/canonical/sphinx-docs-starter-pack/releases/tag/pre-extension

2.5.3. Migrate source files

The documentation starter pack has undergone breaking changes with the introduction of the `canonical-sphinx` extension. This section guides you through:

- Configuration file changes
- Extension dependencies
- Documentation source migration

For a complete list of the structural changes, refer to the *directory-structure-change* (page 28) section.

Sphinx configuration

A significant change in the new starter pack is the organisation of the configuration files, summarised in the following table:

Configuration file	Pre-extension	Extension-based
`conf.py`	Common configurations shared by all starter pack projects	Project-specific configurations
`custom_conf.py`	Project-specific configuration	Merged into `conf.py` and removed

In the new starter pack, many common configurations are provided by the extension and are loaded automatically when building the documentation. `docs/conf.py` is the only configuration file, and it contains all project-specific configuration. Sensible defaults are set for general configuration by inclusion of the *canonical-sphinx* extension.

Ensure that all the previous changes in the original `custom_conf.py` file are copied to the new `conf.py` file.

Dependencies

If your project requires additional extensions beyond the default list, add the extension list to the new project in `docs/.sphinx/requirements.txt`.

Documentation source files

1. Remove the starter pack's documentation files (`index.rst` and any files in the `docs/**/*` sub-directory).
2. Copy all documentation source files from your original project to the new project, keeping their original structure. These file may include but are not limited to:
    - `.md`
    - `.rst`
    - `.txt`
    - `.json`
    - images
    - scripts

3. Validate the migration by running `make run`.

2.5.4. Apply customisation

If your projects have custom configurations or styles, ensure that you identify and apply these changes to the new documentation project.

For general information on customising the extension configuration, see *Customise the setup* (page 13).

Static resources

The extension provides a set of static resources, such as images, fonts, CSS files, and HTML templates, that are used to style the documentation for Canonical-branded design. These resources are bundled with the extension and are no longer provided as source files in the new starter pack.

If you have customised any of these resources in the original project, you need to manually migrate these changes to the new project.

For example, if you added customised styling in the original `.sphinx/_static/custom.css` file, follow the steps:

1. Compare the changes between your customised file and the default CSS file provided by the extension[37]. This comparison helps you identify the changes that need to be migrated to the new project.
2. Create a new CSS file under `docs/.sphinx/_static`. You can choose any other file location in the project directory, but it's recommended to keep the file structure similar to the original project.
3. Copy the additions and changes to the new empty file.
4. In the `conf.py`, add the new files into the pre-defined `html_css_files` list variable to overwrite the default settings.
5. Build the documentation to verify that the customised styling is applied correctly.

2.5.5. Directory structure changes

After you migrate to the extension, some directories and files are either deleted from the project or moved to a new location.

Assuming that all previous documentation files were in the `docs/` sub-directory, the following list illustrates the changes in the directory structure after the migration.

```
.
├── .github
│   └── workflows
│       ├── automatic-doc-checks.yml
│       └── markdown-style-checks.yml
├── .sphinx                         # moved to `docs/.sphinx`
│   ├── fonts                       # removed, files are part of the extension
│   │   ├── Ubuntu-B.ttf
│   │   ├── ubuntu-font-licence-1.0.txt
│   │   ├── UbuntuMono-B.ttf
│   │   ├── UbuntuMono-RI.ttf
```

(continues on next page)

[37] https://github.com/canonical/canonical-sphinx/blob/main/canonical_sphinx/theme/static/custom.css

(continued from previous page)

```
│   │   ├── UbuntuMono-R.ttf
│   │   ├── Ubuntu-RI.ttf
│   │   └── Ubuntu-R.ttf
│   ├── images                  # removed, files are part of the extension
│   │   ├── Canonical-logo-4x.png
│   │   ├── front-page-light.pdf
│   │   ├── front-page.png
│   │   └── normal-page-footer.pdf
│   ├── _static                 # removed, files are part of the extension
│   │   ├── 404.svg
│   │   ├── custom.css
│   │   ├── favicon.png
│   │   ├── footer.css
│   │   ├── footer.js
│   │   ├── furo_colors.css
│   │   ├── github_issue_links.css
│   │   ├── github_issue_links.js
│   │   ├── header.css
│   │   ├── header-nav.js
│   │   └── tag.png
│   ├── _templates              # removed, files are part of the extension
│   │   ├── sidebar
│   │   │   └── search.html
│   │   ├── 404.html
│   │   ├── base.html
│   │   ├── footer.html
│   │   ├── header.html
│   │   └── page.html
│   ├── build_requirements.py   # removed
│   ├── get_vale_conf.py
│   ├── latex_elements_template.txt     # removed, now part of the extension
│   ├── pa11y-ci.json           # renamed to `pa11y.json`
│   └── spellingcheck.yaml
├── metrics                     # moved to `docs/.sphinx/metrics/`
│   └── scripts                 # removed, files moved to parent directory
│       ├── build_metrics.sh
│       └── source_metrics.sh
├── reuse                       # moved to `docs/reuse`
│   └── links.txt
├── .custom_wordlist.txt        # moved to `docs/.custom_wordlist.txt`
├── .gitignore
├── .readthedocs.yaml
├── .wordlist.txt               # moved to `docs/.sphinx/.wordlist.txt`
├── .wokeignore                 # removed, check replaced by Vale
├── conf.py                     # removed, now part of the extension
├── custom_conf.py              # renamed and moved to `docs/conf.py`
├── doc-cheat-sheet-myst.md     # moved to `docs/doc-cheat-sheet-myst.md`
├── doc-cheat-sheet.rst         # moved to `docs/doc-cheat-sheet.rst`
├── index.rst                   # moved to `docs/index.rst`
```

(continues on next page)

(continued from previous page)

```
├── init.sh                    # removed
├── make.bat                   # removed
├── Makefile                   # moved to `docs/Makefile`
├── Makefile.sp                # removed
└── readme.rst                 # renamed to `README.rst`
```

2.6. Update the documentation

This section is intended for documentation contributors and covers how to work with the documentation after the repository has been set up with the starter pack.

You'll find information on how to build and preview the documentation, and some pointers on what you need to know about the documentation framework and where to get help with it.

2.6.1. Install prerequisites

The documentation framework that the starter pack uses bundles most prerequisites in a Python virtual environment, so you don't need to worry about installing them. There are only a few packages that you need to install on your host system.

Install prerequisite software

Before you start, make sure that you have `make`, `python3`, `python3-venv`, and `python3-pip` on your system:

```
sudo apt update
sudo apt install make python3 python3-venv python3-pip
```

Python environment

The Python prerequisites from the `.sphinx/requirements.txt` file are automatically installed when you build the documentation.

If you want to install them manually, you can run the following command from within your documentation folder:

```
make install
```

This command creates a virtual environment (`.sphinx/venv/`) and installs dependency software within it.

If you want to remove the installed Python packages (for example, to enforce a re-installation), run the following command from within your documentation folder:

```
make clean
```

Note:

- By default, the starter pack uses the latest compatible version of all tools and does not pin its requirements. This might change temporarily if there is an incompatibility with a new tool version. There is therefore no need to use a tool like Renovate to automatically update the requirements.
- If you encounter the error `locale.Error: unsupported locale setting` when activating the Python virtual environment, include the environment variable in the command and try again: `LC_ALL=en_US.UTF-8 make run`

2.6.2. Build and preview

The starter pack provides **make** commands to build and view the documentation.

All these commands will automatically set up the Python environment if it isn't set up yet.

Important:

Run these commands from within your documentation folder.

Build the documentation

To build the documentation, run the following command:

```
make html
```

This command installs the required tools and renders the output to the `_build/` folder in your documentation folder.

Important:

When you run **make html** again, it updates the documentation for changed files only. This speeds up the build, but it can cause you to miss warnings or errors that were displayed before. To force a clean build, see *Run a clean build* (page 31).

Make sure that the documentation builds without any warnings (warnings are treated as errors).

Run a clean build

To delete all existing output files and build all files again, run the following command:

```
make clean-doc html
```

To delete both the existing output files and the Python environment and build the full documentation again, run the following command:

```
make clean html
```

View the documentation

To view the documentation output, run the following command:

```
make serve
```

This command builds the documentation and serves it on `http://127.0.0.1:8000/`.

Live view

Instead of building the documentation for each change and then serving it, you can run a live preview of the documentation:

```
make run
```

This command builds the documentation and serves it on `http://127.0.0.1:8000/`. When you change a documentation file and save it, the documentation will be automatically rebuilt and refreshed in the browser.

> **Important:**
>
> The **run** target is very convenient while working on documentation updates.
> However, it is quite error-prone because it displays warnings or errors only when they occur. If you save other files later, you might miss these messages.
> Therefore, you should always *Run a clean build* (page 31) before finalising your changes.

Build a PDF

Build a PDF locally with the following command:

```
make pdf
```

PDF generation requires specific software packages. If these files are not found, a prompt will be presented and the generation will stop.

Required software packages include:

- dvipng
- fonts-freefont-otf
- latexmk
- plantuml
- tex-gyre
- texlive-font-utils
- texlive-fonts-recommended
- texlive-lang-cjk
- texlive-latex-extra
- texlive-latex-recommended
- texlive-xetex

- xindy

On Linux, required packages can be installed with:

```
make pdf-prep-force
```

Note:

When generating a PDF, the index page is considered a 'foreword' and will not be labelled with a chapter.

Important:

When generating a PDF, it is important to not use additional headings before a `toctree`. Documents referenced by the `toctree` will be nested under any provided headings.
A `rubric` directive can be combined with the `h2` class to provide a heading-styled rubric in the HTML output. See the default `index.rst` for an example. Rubric-based headings aren't included as entries in the table of contents or the navigation sidebar.

2.6.3. Edit content

The landing page is stored in the `docs/index.rst` file by default.

The Navigation Menu structure is set by `.. toctree::` directives. These directives define the hierarchy of included content throughout the documentation. The `index.rst` page's `toctree` block contains the top level Navigation Menu, default to the Diátaxis[38] documentation structure.

To add a new page to the documentation:

1. Create a new file in the *docs/* folder. For example, to create a new **Reference** page, create a document under *docs/reference/* directory called `settings.rst`, insert the following reST-formatted heading `Settings` at the beginning, and then save the file:

   Listing 1: reStructuredText title example

   ```
   Settings
   ========
   ```

   If you prefer to use Markdown (MyST) syntax instead of reST, you can create a Markdown file. For example, `settings.md` file with the following Markdown-formatted heading at the beginning:

   Listing 2: Markdown title example

   ```
   # Settings
   ```

2. Add the new page to the Navigation Menu: open the `docs/reference/index.rst` file or another file where you want to nest the new page; at the bottom of the file, locate the `toctree` directive and add a properly indented line containing the relative path (without a file extension) to the new file created in the first step. For example, `settings`.

   The `toctree` block will now look like this:

[38] https://diataxis.fr/

```
.. toctree::
   :hidden:
   :maxdepth: 2

   Documentation checks <automatic_checks>
   style-guide
   style-guide-myst
   settings
```

The documentation will now show the new page added to the navigation when rebuilt.

By default, the page's title (the first heading in the file) is used for the Navigation Menu entry. You can overwrite a name of a Menu element by specifying it explicitly in the `toctree` block, for example: `Reference </reference/index>`.

2.6.4. Get guidance

The starter pack uses Sphinx[39] as the documentation framework. It supports markup in both reStructuredText[40] and Markdown[41] with MyST[42].

It's outside of the scope of the starter pack to teach you how to use these tools to create documentation, but you can check the *Set up the documentation repository* (page 3) for a very brief introduction. For more detailed information and syntax guides, see the linked documents.

To make it easier for you to get started, and to keep our documentation consistent, the following syntax guides give recommendations and conventions for using reST and Markdown/MyST:

- *reStructuredText syntax guide* (page 47)
- *MyST syntax guide* (page 62)

The starter pack also contains cheat sheets for both markup languages that allow to easily copy and paste the markup that you want. See the `doc-cheat-sheet.rst` and `doc-cheat-sheet-myst.md` files.

Other resources

Canonical documentation uses Diátaxis[43].

The following documentation repository contains an example for how to implement this structure:

- Example product documentation[44]
- Example product documentation repository[45]

[39] https://www.sphinx-doc.org/
[40] https://www.sphinx-doc.org/en/master/usage/restructuredtext/index.html
[41] https://commonmark.org/
[42] https://myst-parser.readthedocs.io/
[43] https://diataxis.fr/
[44] https://canonical-example-product-documentation.readthedocs-hosted.com/
[45] https://github.com/canonical/example-product-documentation

2.7. How to contribute

We believe that everyone has something valuable to contribute, whether you're a coder, a writer or a tester. Here's how and why you can get involved:

- **Why join us?** Work with like-minded people, develop your skills, connect with diverse professionals, and make a difference.
- **What do you get?** Personal growth, recognition for your contributions, early access to new features and the joy of seeing your work appreciated.
- **Start early, start easy**: Dive into code contributions, improve documentation, or be among the first testers. Your presence matters, regardless of experience or the size of your contribution.

The guidelines below will help keep your contributions effective and meaningful.

2.7.1. Code of conduct

When contributing, you must abide by the Ubuntu Code of Conduct[46].

2.7.2. Licence and copyright

By default, all contributions to ACME are made under the AGPLv3 licence. See the licence[47] in the ACME GitHub repository for details.

All contributors must sign the Canonical contributor licence agreement[48], which grants Canonical permission to use the contributions. The author of a change remains the copyright owner of their code (no copyright assignment occurs).

2.7.3. Releases and versions

ACME uses semantic versioning[49]; major releases occur once or twice a year.

The release notes can be found TODO: here[50].

2.7.4. Environment setup

To work on the project, you need the following prerequisites:

- TODO: Prerequisite 1[51]
- TODO: Prerequisite 2[52]

To install and configure these tools:

```
TODO: prerequisite command 1
TODO: prerequisite command 2
```

2.7.5. Submissions

If you want to address an issue or a bug in ACME, notify in advance the people involved to avoid confusion; also, reference the issue or bug number when you submit the changes.

- Fork[53] our GitHub repository[54] and add the changes to your fork, properly structuring

[46] https://ubuntu.com/community/ethos/code-of-conduct
[47] https://github.com/canonical/ACME/blob/main/COPYING
[48] https://ubuntu.com/legal/contributors
[49] https://semver.org/
[50] https://example.com
[51] http://example.com
[52] http://example.com
[53] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks
[54] https://github.com/canonical/ACME

your commits, providing detailed commit messages and signing your commits.

- Make sure the updated project builds and runs without warnings or errors; this includes linting, documentation, code and tests.
- Submit the changes as a pull request (PR)[55].

Your changes will be reviewed in due time; if approved, they will be eventually merged.

Describing pull requests

To be properly considered, reviewed and merged, your pull request must provide the following details:

- **Title**: Summarise the change in a short, descriptive title.
- **Description**: Explain the problem that your pull request solves. Mention any new features, bug fixes or refactoring.
- **Relevant issues**: Reference any related issues, pull requests and repositories[56].
- **Testing**: Explain whether new or updated tests are included.
- **Reversibility**: If you propose decisions that may be costly to reverse, list the reasons and suggest steps to reverse the changes if necessary.

Commit structure and messages

Use separate commits for each logical change, and for changes to different components. Prefix your commit messages with names of components they affect, using the code tree structure, e.g. start a commit that updates the ACME service with `ACME/service:`.

Use conventional commits[57] to ensure consistency across the project:

```
Ensure correct permissions and ownership for the content mounts

 * Work around an ACME issue regarding empty dirs:
   https://github.com/canonical/ACME/issues/12345

 * Ensure the source directory is owned by the user running a container.

Links:
- ...
- ...
```

Such structure makes it easier to review contributions and simplifies porting fixes to other branches.

[55] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requ
creating-a-pull-request-from-a-fork

[56] https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/autolinked-references-and-urls

[57] https://www.conventionalcommits.org/

Signing commits

To improve contribution tracking, we use the developer certificate of origin (DCO 1.1[58]) and require signed commits (using the `-S` or `--gpg-sign` option) for all changes that go into the ACME project.

```
git commit -S -m "acme/component: updated life cycle diagram"
```

Signed commits will have a GPG, SSH, or S/MIME signature that is cryptographically verifiable, and will be marked with a "Verified" or "Partially verified" badge in GitHub. This verifies that you made the changes or have the right to commit it as an open-source contribution.

To set up locally signed commits and tags, see GitHub Docs - About commit signature verification[59].

> **Tip:**
>
> You can configure your Git client to sign commits by default for any local repository by running `git config --global commit.gpgsign true`. Once you have configured this, you no longer need to add `-S` to your commits explicitly.
> See GitHub Docs - Signing commits[60] for more information.
>
> [60] https://docs.github.com/en/authentication/managing-commit-signature-verification/signing-commits

If you've made an unsigned commit and encounter the "Commits must have verified signatures" error when pushing your changes to the remote:

1. Amend the most recent commit by signing it without changing the commit message, and push again:

   ```
   git commit --amend --no-edit -n -S
   git push
   ```

2. If you still encounter the same error, confirm that your GitHub account has been set up properly to sign commits as described in the GitHub Docs - About commit signature verification[61].

   > **Tip:**
   >
   > If you use SSH keys to sign your commits, make sure to add a "Signing Key" type in your GitHub account. See [GitHub Docs - Adding a new SSH key to your account](https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account) for more information.

2.7.6. Code

Formatting and linting

ACME relies on these formatting and linting tools:

[58] https://developercertificate.org/
[59] https://docs.github.com/en/authentication/managing-commit-signature-verification/about-commit-signature-verification
[61] https://docs.github.com/en/authentication/managing-commit-signature-verification/about-commit-signature-verification

- TODO: Tool 1[62]
- TODO: Tool 2[63]

To configure and run them:

```
TODO: lint command 1
TODO: lint command 2
```

Structure

- **Check linked code elements**: Check that coupled code elements, files and directories are adjacent. For instance, store test data close to the corresponding test code.
- **Group variable declaration and initialisation**: Declare and initialise variables together to improve code organisation and readability.
- **Split large expressions**: Break down large expressions into smaller self-explanatory parts. Use multiple variables where appropriate to make the code more understandable and choose names that reflect their purpose.
- **Use blank lines for logical separation**: Insert a blank line between two logically separate sections of code. This improves its structure and makes it easier to understand.
- **Avoid nested conditions**: Avoid nesting conditions to improve readability and maintainability.
- **Remove dead code and redundant comments**: Drop unused or obsolete code and comments. This promotes a cleaner code base and reduces confusion.
- **Normalise symmetries**: Treat identical operations consistently, using a uniform approach. This also improves consistency and readability.

Best practices

2.7.7. Tests

All code contributions must include tests.

To run the tests locally before submitting your changes:

```
TODO: test command 1
TODO: test command 2
```

2.7.8. Documentation

ACME's documentation is stored in the `DOCDIR` directory of the repository. It is based on the Canonical starter pack[64] and hosted on Read the Docs[65].

For syntax help and guidelines, refer to the Canonical syntax guides (*reStructuredText* (page 47) and *MyST* (page 62)).

In structuring, the documentation employs the Diátaxis[66] approach.

To run the documentation locally before submitting your changes:

[62] http://example.com
[63] http://example.com
[64] https://canonical-starter-pack.readthedocs-hosted.com/latest/
[65] https://about.readthedocs.com/
[66] https://diataxis.fr/

```
make run
```

Automatic checks

GitHub runs automatic checks on the documentation to verify spelling, validate links and suggest inclusive language.

You can (and should) run the same checks locally:

```
make spelling
make linkcheck
make woke
```

3. Reference

These documents provide an overview of different features of the starter pack.

Also see the following information:

- Example product documentation[67] and Example product documentation repository[68]
- Sphinx documentation starter pack repository[69]

3.1. Contents

3.1.1. Automatic checks

The starter pack comes with several automatic checks that you can (and should!) run on your documentation before committing and pushing changes.

The following checks are available:

Accessibility check

The accessibility check uses Pa11y[70] to check for accessibility issues in the documentation output.

It is configured to use the Web Content Accessibility Guidelines (WCAG) 2.2[71], requiring Level AA conformance[72].

> **Note:**
>
> This check is only available locally.

Install prerequisite software

`Pa11y` must be installed through `npm`. If you need to install `npm`, run the following command from any location on your system:

```
sudo apt install npm
```

Once `npm` is installed, install `Pa11y` by running this command from within your documentation folder.

```
make pa11y-install
```

[67] https://canonical-example-product-documentation.readthedocs-hosted.com/
[68] https://github.com/canonical/example-product-documentation
[69] https://github.com/canonical/starter-pack
[70] https://pa11y.org/
[71] https://www.w3.org/TR/WCAG22/
[72] https://www.w3.org/WAI/WCAG2AA-Conformance

Run the accessibility check

Run the following command from within your documentation folder.

Look for accessibility issues in rendered documentation:

```
make pa11y
```

Configure the accessibility check

The `pa11y.json` file in the `.sphinx` folder provides basic defaults.

To browse the available settings and options, see `Pa11y`'s README[73] on GitHub.

Inclusive language check

The inclusive language check uses Vale[74] to check for violations of inclusive language guidelines.

Run the inclusive language check

Run the following command from within your documentation folder:

```
make woke
```

Configure the inclusive language check

By default, the inclusive language check is applied to Markdown and reST files located in the documentation folder (usually `docs/`).

Inclusive language check exemptions

Sometimes, you might need to use some non-inclusive words. In such cases, you may exclude them from the check.

Exempt a word in a single instance

To exempt an individual word, give the word the `woke-ignore` role:

```
:woke-ignore:`<SOME_WORD>`
```

For instance:

```
This is your text. The word in question is here: :woke-ignore:`whitelist`.
```

> **Note:**
>
> Vale will lint the displayed text of a link, not the URL of a link. If you wish to use a link that contains non-inclusive language, use appropriate link text with the syntax appropriate for your source file.

[73] https://github.com/pa11y/pa11y#command-line-configuration

[74] https://vale.sh/

Exempt a word globally

Vale will ignore any word listed in the `.custom_wordlist.txt` file. To exempt a word, add it to this file globally.

Note:

Entries in `.custom-wordlist` are case-sensitive only when a capitalised word is used. For instance:

- Adding `kustom` will cause all instances of `Kustom` and `kustom` to be ignored.
- Adding `Kustom` will cause only instances of `Kustom` to be ignored.

Exclude multiple lines from a file

Vale can be switched on and off within a file using syntax appropriate to that format.

To turn Vale off entirely for a section of Markdown:

```
<!-- vale off -->

This text will be ignored.

<!-- vale on -->
```

Important:

Only use this when other options are not suitable.

To turn Vale off entirely for a section of reST:

```
.. vale off

This text will be ignored.

.. vale on
```

Link check

The link check uses Sphinx to access the links in the documentation output and validate whether they are working.

Run the link check

Run the following command from within your documentation folder.

Validate links within the documentation:

```
make linkcheck
```

Configure the link check

If you have links in the documentation that you don't want to be checked (for example, because they are local links or give random errors even though they work), you can add them to the `linkcheck_ignore` variable in the `conf.py` file.

Lint check

Markdown

The Markdown lint check is used to enforce standards and consistency in Markdown files.

Run the lint check

Run the following command from within your documentation folder to lint your Markdown files:

```
make lint-md
```

Configure the lint check

You can update the linting rules to enforce in the `.sphinx/.pymarkdown.json` file. Refer to the pymarkdown rules documentation[75] for all the available rules.

Spelling check

The spelling check uses `vale` to check the spelling in your documentation. It ignores code (both code blocks and inline code) and URLs (but it does check the link text).

Run the spelling check

Run the following commands from within your documentation folder.

Ensure there are no spelling errors in the documentation:

```
make spelling
```

Configure the spelling check

The Vale repository includes a common list of words[76] that will be excluded from the check. To add custom exceptions for your project, add them to the `.custom_wordlist.txt` file.

Exclude specific terms

Sometimes, you need to use a term in a specific context that should usually fail the spelling check. (For example, you might need to refer to a product called `ABC Docs`, but you do not want to add `docs` to the word list because it isn't a valid word.)

In this case, you can use the `:vale-ignore:` role, and ensure your configuration file contains a class association in the `rst_prolog`:

[75] https://pymarkdown.readthedocs.io/en/latest/rules/

[76] https://github.com/canonical/documentation-style-guide/blob/main/styles/config/vocabularies/Canonical/accept.txt

```
rst_prolog = """
.. role:: vale-ignore
    :class: vale-ignore
"""
```

Style guide linting

The starter pack includes a method to run the Vale[77] documentation linter configured with the Vale rules for the current style guide[78].

Run the style guide linting

Run the following commands from within your documentation folder.

Check documentation with Vale:

```
make vale
```

Vale can run against individual files, folders, or globs. To set a specific target:

```
make vale TARGET=example.file
make vale TARGET=example-folder
```

Note:

Running Vale against a folder will also run against its subfolders.

You can use wildcards to run against all files matching a string, or an extension.

For example, to run against all `.md` files within a folder:

```
make vale TARGET=*.md
```

To match, for example, `doc_1.md` and `doc_2.md`:

```
make vale TARGET=doc*
```

Exempt paragraphs

To disable Vale linting within individual files, specific markup can be used.

For Markdown:

```
<!-- vale off -->

This text will be ignored by Vale.

<!-- vale on -->
```

For reST:

[77] https://vale.sh/
[78] https://github.com/canonical/documentation-style-guide

```
.. vale off

This text will be ignored by Vale.

.. vale on
```

Exempt directives

To disable Vale linting for a specific directive, you can apply a class to the section.

For Markdown:

`````
````{class} vale-ignore
```{code-block}

This content will be ignored by Vale.
```
````
`````

Note:

This should not be necessary for Markdown, as Vale has an expanded scope for ignoring Markdown content by default.

For reST:

```
.. class:: vale-ignore
.. code-block::

    This content will be ignored by Vale.
```

Note:

The .. *class::* directive does not need to encapsulate content, it applies to the next logical block (which can be another directive or even a paragraph of content).

Exempt words

Use the `:vale-ignore:` role to ignore specific words inline, but first ensure your configuration file contains a class association in the `rst_prolog`:

```
rst_prolog = """
.. role:: vale-ignore
    :class: vale-ignore
"""
```

Important:

The spelling check might still flag some terms that contain hyphens or spaces.
For example, "Juju 3" was unable to be ignored by this method, and needed to be added to the a specific exception within a rule[79].

[79] https://github.com/canonical/documentation-style-guide/blob/a6f530b07d774bee67dd79d146ae5bbedc9ddef1/styles/Canonical/013-Spell-out-numbers-below-10.yml#L15

Install prerequisite software

Some of the tools used by the automatic checks require `npm`. Install `npm` using the appropriate method for your operating system through one of the following methods:

- Your preferred package manager
- By following the node version manager installation process[80]
- For Debian and Ubuntu Linux distributions, the `sudo apt install npm` command

To install the validation tools:

```
make pa11y-install
make pymarkdownlnt-install  # if using Markdown
```

Note:

pa11y is a non-blocking check in our current documentation workflow.

Default GitHub actions

The starter pack uses default workflows from the documentation-workflows[81] repository.

The current defaults force usage of Canonical hosted runners, which some projects may not be able to use. You may select your own runners with an override, see line 7 below:

vale-ignore

```
1 jobs:
2 documentation-checks:
3   uses: canonical/documentation-workflows/.github/workflows/documentation-checks.
yaml@main
4   with:
5     working-directory: "docs"
6     fetch-depth: 0
7     runs-on: "ubuntu-22.04"
```

[80] https://docs.npmjs.com/downloading-and-installing-node-js-and-npm#using-a-node-version-manager-to-install-nodejs-and-
[81] https://github.com/canonical/documentation-workflows/

Workflow triggers

For efficiency, the documentation check workflows are configured to run **only** when changes are made to files in the `docs/` directory. If your project is structured differently, or if you want to run the checks on other directories, modify the trigger paths in the workflow files:

vale-ignore

```
on:
  pull_request:
    paths:
      - 'docs/**'    # Only run on changes to the docs directory
```

3.1.2. Check for removed URLs

Added in version 1.2.0.

The starter pack includes a GitHub action to identify when page URLs have been removed. This includes moving pages to another path, or removing them completely.

This does not cover higher-level changes to URL paths, such as changing the RTD project name, or language and versioning structure provided by RTD.

This check is available to ensure that redirects are implemented when pages are moved, or appropriate information can be provided when anything is removed.

3.1.3. reStructuredText syntax reference

The documentation files use reStructuredText[82] (reST) syntax.

See the following sections for syntax help and conventions.

> **Note:**
>
> This guide assumes that you are using the Sphinx documentation starter pack[83]. Some of the mentioned syntax requires Sphinx extensions (which are enabled in the starter pack).
>
> [83] https://github.com/canonical/starter-pack

For general style conventions, see the Canonical Documentation Style Guide[84].

[82] https://www.sphinx-doc.org/en/master/usage/restructuredtext/index.html
[84] https://docs.ubuntu.com/styleguide/en

Headings

Input	Description
`Title` `=====`	Page title and H1 heading
`Heading` `-------`	H2 heading
`Heading` `~~~~~~~`	H3 heading
`Heading` `^^^^^^^`	H4 heading
`Heading` `.......`	H5 heading

Underlines must be at least as long as the title or heading.

Adhere to the following conventions:

- Do not use consecutive headings without intervening text.
- Be consistent with the characters you use for each level. Use the ones specified above.
- Use sentence style for headings (capitalise only the first word).

Inline formatting

Input	Output
`:guilabel:`UI element``	*UI element*
` ``code`` `	`code`
`:file:`file path``	`file path`
`:command:`command``	**`command`**
`:kbd:`Key``	`Key`
`*Italic*`	*Italic*
`**Bold**`	**Bold**

Adhere to the following conventions:

- Use italics sparingly. Common uses for italics are titles and names (for example, when referring to a section title that you cannot link to, or when introducing the name for a concept).
- Use bold sparingly. Avoid using bold for emphasis and rather rewrite the sentence to get your point across.

Code blocks

To start a code block, either end the introductory paragraph with two colons (`::`) and indent the following code block, or explicitly start a code block with `.. code::`. In both cases, the code block must be surrounded by empty lines.

When explicitly starting a code block, you can specify the code language to enforce a specific lexer, but in many cases, the default lexer works just fine.

For a list of supported languages and their respective lexers, see the official Pygments documentation[85].

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>

```
Demonstrate a code block::

  code:
  - example: true
```

</td><td>

Demonstrate a code block:

```
code:
- example: true
```

</td></tr>
<tr><td>

```
.. code::

   # Demonstrate a code block
   code:
   - example: true
```

</td><td>

```
# Demonstrate a code block
code:
- example: true
```

</td></tr>
<tr><td>

```
.. code:: yaml

   # Demonstrate a code block
   code:
   - example: true
```

</td><td>

```yaml
# Demonstrate a code block
code:
- example: true
```

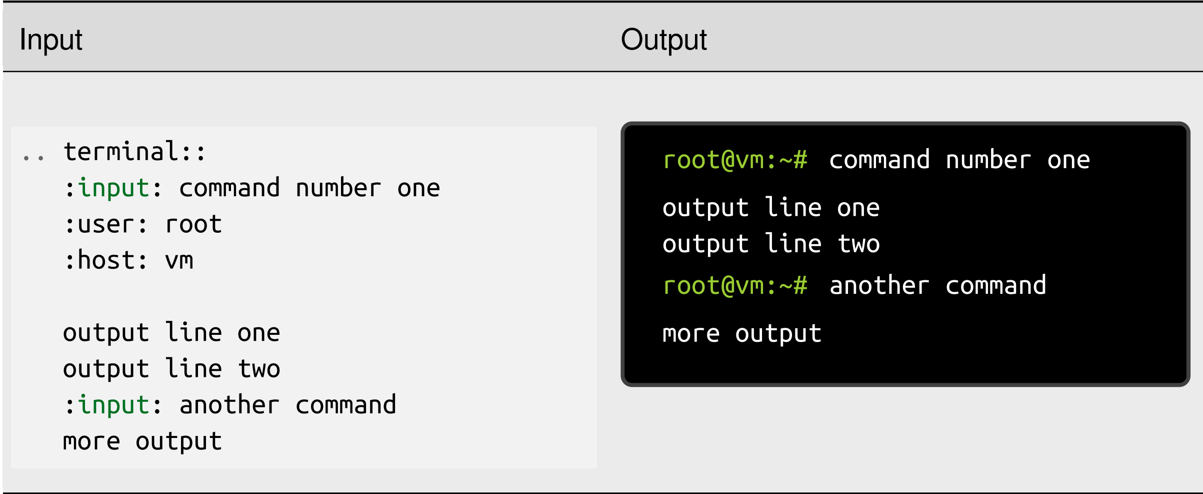
</td></tr>
</table>

Terminal output

Showing a terminal view can be useful to show the output of a specific command or series of commands, where it is important to see the difference between input and output. In addition, including a terminal view can help break up a long text and make it easier to consume, which is especially useful when documenting command-line-only products.

[85] https://pygments.org/languages/

To include a terminal view, use the following directive:



Input	Output
(see code below)	(see terminal output below)

Input:

```
.. terminal::
   :input: command number one
   :user: root
   :host: vm

   output line one
   output line two
   :input: another command
   more output
```

Output:

```
root@vm:~# command number one
output line one
output line two
root@vm:~# another command
more output
```

Input is specified as the `:input:` option (or prefixed with `:input:` as part of the main content of the directive). Output is the main content of the directive.

To override the prompt (`user@host:~$` by default), specify the `:user:` and/or `:host:` options. To make the terminal scroll horizontally instead of wrapping long lines, add `:scroll:`.

Links

Link markup depends on whether you need an external URL or a page in the same documentation set.

External links

For external links, use one of the following methods.

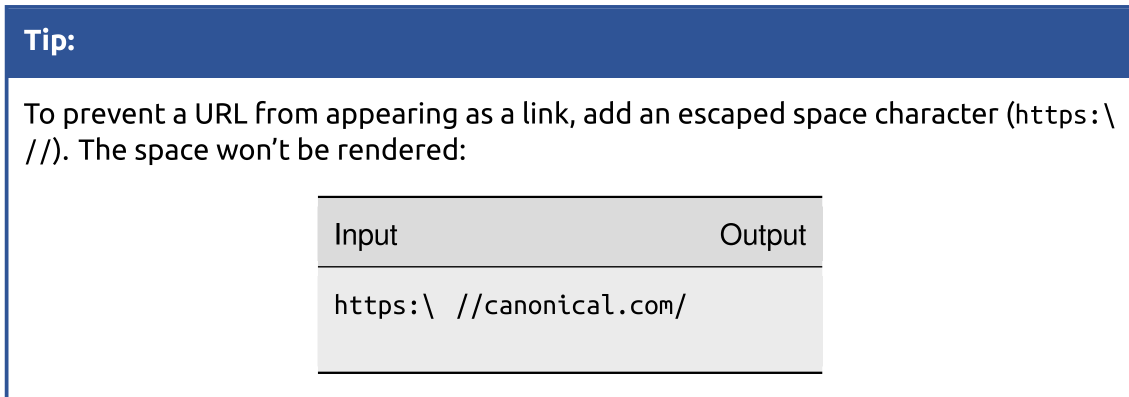
Link inline:

Define occasional links directly within the surrounding text. To make the link text show up in code-style (which excludes it from the spelling check), use the `:literalref:` role.

Input	Output
`` `Canonical website <https://canonical.com/>`_ ``	Canonical website[86]
`` :literalref:`ubuntu.com` ``	`ubuntu.com`[87]
`` :literalref:`xyzcommand <https://example.com>` ``	`xyzcommand`[88]

You can also use a URL as is (`https://example.com`), but that might cause spellchecker errors.

[86] https://canonical.com/
[87] https://ubuntu.com
[88] https://example.com



Tip:

To prevent a URL from appearing as a link, add an escaped space character (`https:\ //`). The space won't be rendered:

Input	Output
`https:\ //canonical.com/`	

Define the links at the bottom of the page:

To keep the text readable, group the link definitions below.

Input	Output	Description
`` `Canonical website`_ ``	Canonical website[89]	Using the below defined link
`.. LINKS` `.. _Canonical website: https://canonical.com/`	*n/a*	Defining links at the bottom

Define the links in a shared file:

To keep the text readable and links maintainable, put all link definitions in a file named `reuse/links.txt` to include it in a custom `rst_epilog` directive (see the Sphinx documentation[90]).

Listing 1:

```
custom_rst_epilog = """
    .. include:: reuse/links.txt
    """
```

Input	Output
`` `Canonical website`_ ``	Canonical website[91]

Related links

You can add links to related websites or Discourse topics to the sidebar.

To add a link to a related website, add the following field at the top of the page:

[89] https://canonical.com/

[90] https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-rst_epilog

[91] https://canonical.com/

```
:relatedlinks: https://github.com/canonical/lxd-sphinx-extensions, [RTFM](https://
www.google.com)
```

To override the title, use Markdown syntax. Note that spaces are ignored; if you need spaces in the title, replace them with ` `, and include the value in quotes if Sphinx complains about the metadata value because it starts with `[`.

To add a link to a Discourse topic, configure the Discourse instance in the `custom_conf.py` file. Then add the following field at the top of the page (where `12345` is the ID of the Discourse topic):

```
:discourse: 12345
```

Manual-page links

When mentioning command line utilities, you may wish to link to the corresponding manual page for the command. Ensure that the `manpages_url` setting in your `conf.py` is set appropriately and use the `:manpage:` inline role within your text to create a link.

For example, to link to man pages from the 24.04 LTS (Noble Numbat) release, include the following in your `conf.py`:

```
manpages_url = "https://manpages.ubuntu.com/manpages/noble/en/man{section}/{page}.
{section}.html"
```

Then within your documentation, use the following reST:

```
You can use the :manpage:`dd(1)` utility to write the disk image to your
SD card. If the image is compressed, use :manpage:`aunpack(1)` to extract
it first.
```

YouTube links

To add a link to a YouTube video, use the following directive:

Input	Output
`.. youtube:: https://www.youtube.com/watch?v=iMLiK1fX4I0` `   :title: Demo`	

The video title is extracted automatically and displayed when hovering over the link. To override the title, add the `:title:` option.

Internal references

You can reference pages and targets in this documentation set, and also in other documentation sets using Intersphinx.

Referencing a section

To reference a section within the documentation (either on the same page or on another page), add a target to that section and reference that target.

You can add targets at any place in the documentation. However, if there is no heading or title for the targeted element, you must specify a link text.

Input	Output	Description
`.. _target_ID:`		Adds the target `target_ID`. **Note:** When defining the target, you must prefix it with an underscore. Do not use the starting underscore when referencing the target.
``:ref:`a_section_target` ``	*Referencing a section* (page 53)	References a target that has a title.
``:ref:`Link text <a_random_target>` ``	*Link text* (page 53)	References a target and specifies a title.
``:ref:`starter-pack:home` ``	Sphinx example[92]	You can also reference targets in other doc sets.

Adhere to the following conventions:

- Never use external links to reference a section in the same doc set or a doc set that is linked with Intersphinx. It would likely cause a broken link in the future.
- Override the link text only when it is necessary. If you can use the referenced title as link text, do so, because the text will then update automatically if the title changes.
- Never "override" the link text with the same text that would be generated automatically.

[92] https://canonical-example-product-documentation.readthedocs-hosted.com/en/latest/#home

Referencing a page

If a documentation page does not have a target, you can still reference it by using the :doc: role with the file name and path.

Input	Output
:doc:`index`	*Reference* (page 40)
:doc:`Link text <index>`	*Link text* (page 40)
:doc:`starter-pack:how-to/index`	How-to guides[93]
:doc:`Link text <starter-pack:how-to/index>`	Link text[94]

Adhere to the following conventions:

- Only use the :doc: role when you cannot use the :ref: role, thus only if there is no target at the top of the file and you cannot add it. When using the :doc: role, your reference will break when a file is renamed or moved.
- Override the link text only when it is necessary. If you can use the document title as link text, do so, because the text will then update automatically if the title changes.
- Never "override" the link text with the same text that would be generated automatically.

Navigation

Every documentation page must be included as a sub-page to another page in the navigation.

This is achieved with the toctree[95] directive in the parent page:

```
.. toctree::
   :hidden:

   sub-page1
   sub-page2
```

If a page should not be included in the navigation, you can suppress the resulting build warning by putting :orphan: at the top of the file. Use orphan pages sparingly and only if there is a clear reason for it.

> **Tip:**
>
> Instead of hiding pages that you do not want to include in the documentation from the navigation, you can exclude them from being built. This method will also prevent them from being found through the search.
> To exclude pages from the build, add them to the custom_excludes variable in the custom_conf.py file.

[93] https://canonical-example-product-documentation.readthedocs-hosted.com/en/latest/how-to/
[94] https://canonical-example-product-documentation.readthedocs-hosted.com/en/latest/how-to/
[95] https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html#directive-toctree

Lists

Input	Output

`- Item 1`
`- Item 2`
`- Item 3`
	• Item 1
	• Item 2
	• Item 3

`1. Step 1`
`#. Step 2`
`#. Step 3`
	1. Step 1
	2. Step 2
	3. Step 3

`a. Step 1`
`#. Step 2`
`#. Step 3`
	a. Step 1
	b. Step 2
	c. Step 3

You can also nest lists:

Input

```
1. Step 1

   - Item 1

     * Sub-item
   - Item 2

     i. Sub-step 1
     #. Sub-step 2
#. Step 2

   a. Sub-step 1

      - Item
   #. Sub-step 2
```

Output

1. Step 1

   - Item 1

     - Sub-item

   - Item 2

     i. Sub-step 1

ii. Sub-step 2

2. Step 2

   a. Sub-step 1

      - Item

   b. Sub-step 2

Adhere to the following conventions:

- In numbered lists, number the first item and use `#.` for all subsequent items to generate the step numbers automatically.
- Use `-` for unordered lists. When using nested lists, you can use `*` for the nested level.

Definition lists

Input	Output

`Term 1:`
`  Definition`
`Term 2:`
`  Definition`
	Term 1:
	Definition
	Term 2:
	Definition

Tables

reST supports different markup for tables. Grid tables are most similar to tables in Markdown, but list tables are usually much easier to use. See the Sphinx documentation[96] for all table syntax alternatives.

Both markups result in the following output:

Header 1	Header 2
Cell 1 Second paragraph cell 1	Cell 2
Cell 3	Cell 4

Grid tables

See grid tables[97] for reference.

```
+----------------------+------------+
| Header 1             | Header 2   |
+======================+============+
```

(continues on next page)

[96] https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html#table-directives
[97] https://docutils.sourceforge.io/docs/ref/rst/restructuredtext.html#grid-tables

(continued from previous page)

```
| Cell 1               | Cell 2     |
|                      |            |
| 2nd paragraph cell 1 |            |
+----------------------+------------+
| Cell 3               | Cell 4     |
+----------------------+------------+
```

List tables

See list tables[98] for reference.

```
.. list-table::
   :header-rows: 1

   * - Header 1
     - Header 2
   * - Cell 1

       2nd paragraph cell 1
     - Cell 2
   * - Cell 3
     - Cell 4
```

Notes

Input	Output
`.. note::` `   A note.`	**Note:** A note.
`.. warning::` `   This might damage your hardware!`	**Warning:** This might damage your hardware!

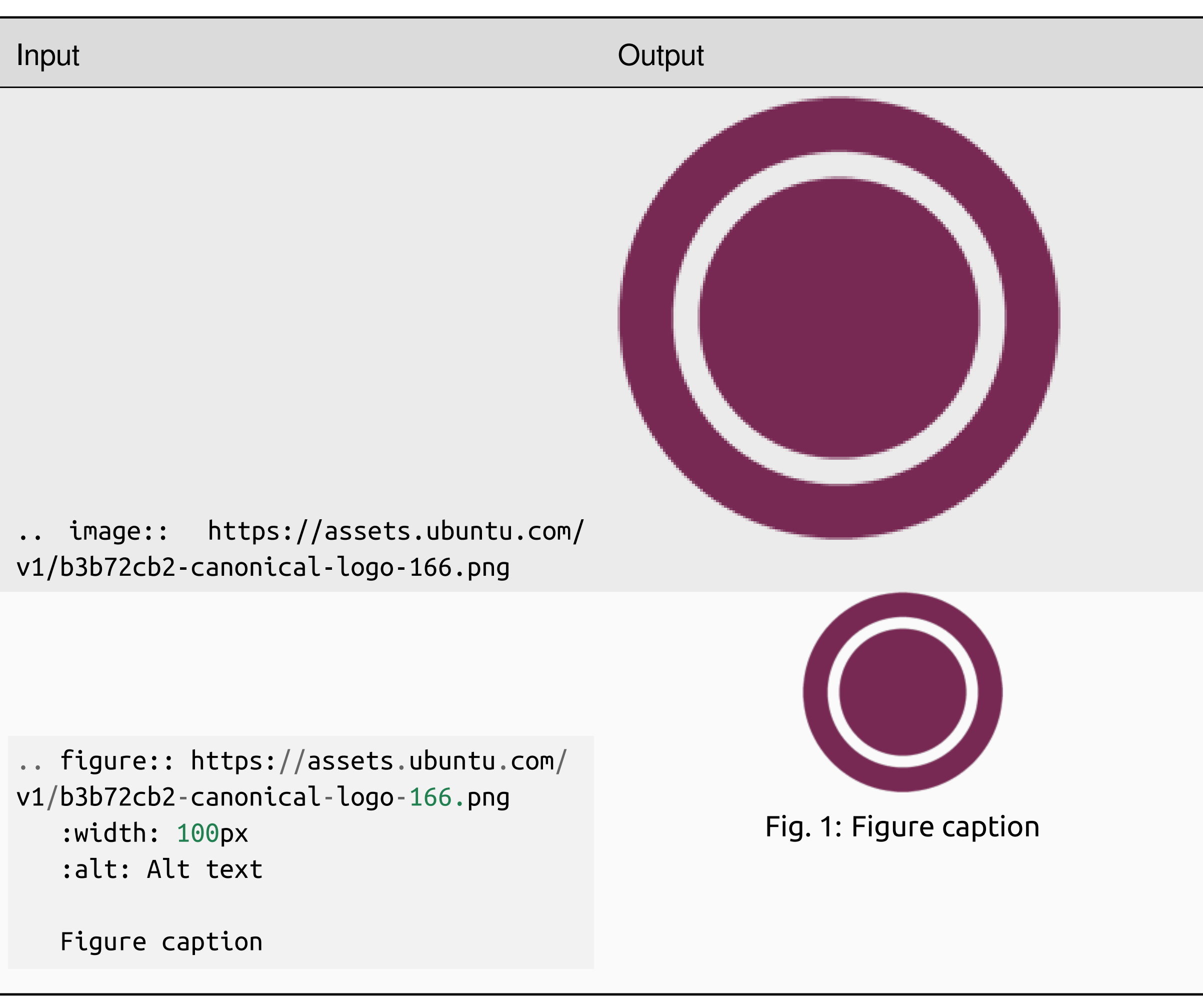
Adhere to the following conventions:

- Use notes sparingly.
- Only use the following note types: `note`, `warning`

[98] https://docutils.sourceforge.io/docs/ref/rst/directives.html#list-table

- Only use a warning if there is a clear hazard of hardware damage or data loss.

Images



Input	Output
`.. image:: https://assets.ubuntu.com/v1/b3b72cb2-canonical-logo-166.png`	

`.. figure:: https://assets.ubuntu.com/v1/b3b72cb2-canonical-logo-166.png`
`:width: 100px`
`:alt: Alt text`
 	
`Figure caption`
	Fig. 1: Figure caption

Adhere to the following conventions:

- For local pictures, start the path with / (for example, `/images/image.png`).
- Use `PNG` format for screenshots and `SVG` format for graphics.
- If producing multiple output formats, use `*` as the file extension to have Sphinx select the best image format for the output
- See Five golden rules for compliant alt text[99] for information about how to word the alt text.

Reuse

A big advantage of reST in comparison to plain Markdown is that it allows to reuse content.

[99] https://abilitynet.org.uk/news-blogs/five-golden-rules-compliant-alt-text

Substitution

To reuse sentences and entire paragraphs that have little markup or special formatting, define substitutions[100] for them in two possible ways.

Globally, in a file named `reuse/substitutions.txt` that is included in a custom `rst_epilog` directive (see the Sphinx documentation[101]):

Listing 2:

```
rst_epilog = """
    .. include:: reuse/substitutions.txt
    """
```

Listing 3:

```
.. |version_number| replace:: 0.1.0

.. |rest_text| replace:: *Multi-line* text
                         that uses basic **markup**.

.. |site_link| replace:: Website link
.. _site_link: https://example.com
```

Locally, putting the same directives in any reST file:

Listing 4:

```
.. |version_number| replace:: 0.1.0

.. |rest_text| replace:: *Multi-line* text
                         that uses basic **markup**.

.. And so on
```

> **Note:**
>
> Mind that substitutions can't be redefined; for instance, accidentally including a definition twice causes an error:
>
> ```
> ERROR: Duplicate substitution definition name: "rest_text".
> ```

The definitions from the above examples are rendered as follows:

[100] https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html#substitutions
[101] https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-rst_epilog

Input	Output
`|version_number|`	0.1.0
`|rest_text|`	*Multi-line* text that uses basic **markup**.
`|site_link|_`	Website link[102]

Tip:

Use substitution names that hint at the included content (for example, `note_not_supported` instead of `note_substitution`).

File inclusion

To reuse longer sections or text with more advanced markup, you can put the content in a separate file and include the file or parts of the file in several locations.

To select parts of the text in a file, use `:start-after:` and `:end-before:` if possible. You can combine those with `:start-line:` and `:end-line:` if required (if the same text occurs more than once). Using only `:start-line:` and `:end-line:` is error-prone though.

You cannot put any targets into the content that is being reused (because references to this target would be ambiguous then). You can, however, put a target right before including the file.

By combining file inclusion and substitutions defined directly in a file, you can even replace parts of the included text.

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>

```
.. include:: index.rst
   :start-after: Also see the following
information:
   :end-before: Contents
```

</td><td>

- Example product documentation[103] and Example product documentation repository[104]
- Sphinx documentation starter pack repository[105]

</td></tr>
</table>

Adhere to the following conventions:

- Files that only contain text that is reused somewhere else should be placed in the `reuse` folder and end with the extension `.txt` to distinguish them from normal content files.
- To make sure inclusions don't break, consider adding comments (`.. some comment`) to the source file as markers for starting and ending.

[102] https://example.com
[103] https://canonical-example-product-documentation.readthedocs-hosted.com/
[104] https://github.com/canonical/example-product-documentation
[105] https://github.com/canonical/starter-pack

Tabs

The recommended way of creating tabs is to use the tabs that the Sphinx design[106] extension provides.

Input	Output

<pre>.. tab-set::
 	
    .. tab-item:: Tab 1
       :sync: key1
 	
       Content Tab 1
 	
    .. tab-item:: Tab 2
       :sync: key2
 	
       Content Tab 2</pre>
	Tab 1
	Content Tab 1
	Tab 2
	Content Tab 2

Alternatively, you can use the Sphinx tabs[107] extension, which is also enabled by default. This was previously recommended due to limitations in Sphinx Design that are now fixed.

Input	Output

<pre>.. tabs::
 	
    .. group-tab:: Tab 1
 	
       Content Tab 1
 	
    .. group-tab:: Tab 2
 	
       Content Tab 2</pre>
	Tab 1
	Tab 2
	Content Tab 1
	Content Tab 2

Glossary

You can define glossary terms in any file. Ideally, all terms should be collected in one glossary file though, and they can then be referenced from any file.

[106] https://sphinx-design.readthedocs.io/en/latest/
[107] https://sphinx-tabs.readthedocs.io/en/latest/

Input	Output

`.. glossary::`
 	
`  an example term`
`    Definition of an example term.`
	an example term
	Definition of an example term.

`` :term:`an example term` ``	*an example term*

More useful markup

Input	Output	Description
`.. versionadded:: X.Y`	Added in version X.Y.	Can be used to distinguish between different versions.

`| Line 1`
`| Line 2`
`| Line 3`
	Line 1
	Line 2
	Line 3
		Line breaks that are not paragraphs. Use this sparingly.

`----`	A horizontal line	Can be used to visually divide sections on a page.
`.. This is a comment`		Not visible in the output.
`` :abbr:`API (Application Programming Interface)` ``	API (Application Programming Interface)	Hover to display the full term.
`` :spellexception:`PurposelyWrong` ``		Explicitly exempt a term from the spelling check.

3.1.4. MyST syntax guide

The documentation files use a mixture of Markdown[108] and MyST[109] syntax.

See the following sections for syntax help and conventions.

Note:

This guide assumes that you are using the Sphinx documentation starter pack[110]. Some of the mentioned syntax requires Sphinx extensions (which are enabled in the starter pack).

[110] https://github.com/canonical/starter-pack

[108] https://commonmark.org/
[109] https://myst-parser.readthedocs.io/

For general style conventions, see the Canonical Documentation Style Guide[111].

Headings

Input	Description
`# Title`	Page title and H1 heading
`## Heading`	H2 heading
`### Heading`	H3 heading
`#### Heading`	H4 heading
`...`	Further headings

Adhere to the following conventions:

- Do not use consecutive headings without intervening text.
- Do not skip levels (for example, do not follow an H2 heading with an H4 heading).
- Use sentence style for headings (capitalise only the first word).

Inline formatting

Input	Output
``{guilabel}`UI element` ``	*UI element*
`` `code` ``	`code`
``{file}`file path` ``	`file path`
``{command}`command` ``	**`command`**
``{kbd}`Key` ``	`Key`
`*Italic*`	*Italic*
`**Bold**`	**Bold**

Adhere to the following conventions:

- Use italics sparingly. Common uses for italics are titles and names (for example, when referring to a section title that you cannot link to, or when introducing the name for a concept).
- Use bold sparingly. Avoid using bold for emphasis and rather rewrite the sentence to get your point across.

[111] https://docs.ubuntu.com/styleguide/en

Code blocks

Start and end a code block with three back ticks:

````
```
````

You can specify the code language after the back ticks to enforce a specific lexer, but in many cases, the default lexer works just fine.

Input	Output
``` # Demonstrate a code block code: - example: true ```	# Demonstrate a code block code: - example: true

``yaml
# Demonstrate a code block 	
code:
- example: true
``
	# Demonstrate a code block
	code:
	- example: true

To include back ticks in a code block, increase the number of surrounding back ticks:

Input	Output

``
``
``
	``

Terminal output

Showing a terminal view can be useful to show the output of a specific command or series of commands, where it is important to see the difference between input and output. In addition, including a terminal view can help break up a long text and make it easier to consume, which is especially useful when documenting command-line-only products.

To show a terminal view, use the following directive:
````

Input	Output

``{terminal}
:input: command number one
:user: root
:host: vm
 	
output line one
output line two
:input: another command
more output
``
	root@vm:~# command number one
	output line one
	output line two
	root@vm:~# another command
	
	more output

Input is specified as the `:input:` option (or prefixed with `:input:` as part of the main content of the directive). Output is the main content of the directive.

To override the prompt (`user@host:~$` by default), specify the `:user:` and/or `:host:` options. To make the terminal scroll horizontally instead of wrapping long lines, add `:scroll:`.

Links

How to link depends on if you are linking to an external URL or to another page in the documentation.

External links

For external links, use Markdown syntax. You can also use just the URL, but this will usually cause issues with the spelling check, so you should specify the link text as code in this case.

Input	Output
[Canonical website](https://canonical.com)	Canonical website[112]
https://canonical.com	[113]
[`https://canonical.com`](https://canonical.com)	https://canonical.com

To display a URL as text and prevent it from being linked, add a :

Input	Output
https://canonical.com	

[112] https://canonical.com
[113] https://canonical.com

Related links

You can add links to related websites or Discourse topics to the sidebar

To add a link to a related website, add the following field at the top of the page:

```
relatedlinks: https://github.com/canonical/canonical-sphinx-extensions,
[RTFM](https://www.google.com)
```

To override the title, use Markdown syntax. Note that spaces are ignored; if you need spaces in the title, replace them with ` `, and include the value in quotes if Sphinx complains about the metadata value because it starts with `[`.

To add a link to a Discourse topic, configure the Discourse instance in the `custom_conf.py` file. Then add the following field at the top of the page (where `12345` is the ID of the Discourse topic):

```
discourse: 12345
```

YouTube links

To add a link to a YouTube video, use the following directive:

Input	Output
` ```{youtube} https://www.youtube.com/watch?v=iMLiK1fX4I0` `:title: Demo` ` ``` `	

The video title is extracted automatically and displayed when hovering over the link. To override the title, add the `:title:` option.

Internal references

For internal references, both Markdown and MyST syntax are supported. In most cases, you should use MyST syntax though, because it resolves the link text automatically and gives an indication of the link in GitHub rendering.

Referencing a section

To reference a section within the documentation (either on the same page or on another page), add a target to that section and reference that target.

You can add targets at any place in the documentation. However, if there is no heading or title for the targeted element, you must specify a link text.

Input	Output	Output on GitHub	Description
`(target_ID)=`		(target_ID)=	Adds the target `target_ID`.
``{ref}`a_section_target_myst` ``	*Referencing a section* (page 66)	{ref}a_section_target_myst	References a target that has a title.
``{ref}`link text <a_random_target_myst>` ``	*link text* (page 67)	{ref}link text <a_random_target_myst>	References a target and specifies a title.
``{ref}`starter-pack:home` ``	Sphinx example[114]	{ref}starter-pack:home	You can also reference targets in other doc sets.
``[`xyz`](a_random_target_myst)``	*xyz* (page 67)	*xyz* (page 67) (link is broken)	Use Markdown syntax if you need markup on the link text.

Adhere to the following conventions:

- Never use external links to reference a section in the same doc set or a doc set that is linked with Intersphinx. It would likely cause a broken link in the future.
- Override the link text only when it is necessary. If you can use the section title as link text, do so, because the text will then update automatically if the title changes.
- Never "override" the link text with the same text that would be generated automatically.

Referencing a page

If a documentation page does not have a target, you can still reference it by using the `{doc}` role with the file name and path. Use MyST syntax to automatically extract the link text. When overriding the link text, use Markdown syntax.

Input	Output	Output on GitHub	Status
``{doc}`index` ``	*Reference* (page 40)	{doc}index	Preferred.
`[](index)`	*Reference* (page 40)		Do not use.
`[Index page](index)`	*Index page* (page 40)	*Index page* (page 40)	Preferred when overriding the link text.
``{doc}`Index page <index>` ``	*Index page* (page 40)	{doc}Index page <index>	Alternative when overriding the link text.

Adhere to the following conventions:

[114] https://canonical-example-product-documentation.readthedocs-hosted.com/en/latest/#home

- Only use the {doc} role when you cannot use the {ref} role, thus only if there is no target at the top of the file and you cannot add it. When using the {doc} role, your reference will break when a file is renamed or moved.
- Override the link text only when it is necessary. If you can use the document title as link text, do so, because the text will then update automatically if the title changes.
- Never "override" the link text with the same text that would be generated automatically.

Navigation

Every documentation page must be included as a sub-page to another page in the navigation.

This is achieved with the toctree[115] directive in the parent page:

````
```{toctree}
:hidden:

sub-page1
sub-page2
```
````

If a page should not be included in the navigation, you can suppress the resulting build warning by putting the following instruction at the top of the file:

```
---
orphan: true
---
```

Use orphan pages sparingly and only if there is a clear reason for it.

> **Tip:**
>
> Instead of hiding pages that you do not want to include in the documentation from the navigation, you can exclude them from being built. This method will also prevent them from being found through the search.
> To exclude pages from the build, add them to the custom_excludes variable in the custom_conf.py file.

[115] https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html#directive-toctree

Lists

Input	Output

`- Item 1`
`- Item 2`
`- Item 3`
	• Item 1
	• Item 2
	• Item 3

`1. Step 1`
`1. Step 2`
`1. Step 3`
	1. Step 1
	2. Step 2
	3. Step 3

`1. Step 1`
`   - Item 1`
`     * Sub-item`
`   - Item 2`
`1. Step 2`
`   1. Sub-step 1`
`   1. Sub-step 2`
	1. Step 1
	  • Item 1
	    – Sub-item
	  • Item 2
	2. Step 2
	  1. Sub-step 1
	  2. Sub-step 2

Adhere to the following conventions:

- In numbered lists, use `1.` for all items to generate the step numbers automatically. You can also use a higher number for the first item to start with that number.
- Use `-` for unordered lists. When using nested lists, you can use `*` for the nested level.

Definition lists

Input	Output

`Term 1`
`: Definition`
 	
`Term 2`
`: Definition`
	Term 1
	  Definition
	Term 2
	  Definition

Tables

You can use standard Markdown tables. However, using the reST list table[116] syntax is usually much easier. See the Sphinx documentation[117] for all table syntax alternatives.

[116] https://docutils.sourceforge.io/docs/ref/rst/directives.html#list-table
[117] https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html#table-directives

Both markups result in the following output:

Header 1	Header 2
Cell 1 Second paragraph cell 1	Cell 2
Cell 3	Cell 4

Markdown tables

```
| Header 1                           | Header 2 |
|------------------------------------|----------|
| Cell 1<br><br>2nd paragraph cell 1 | Cell 2   |
| Cell 3                             | Cell 4   |
```

List tables

See list tables[118] for reference.

````
```{list-table}
   :header-rows: 1

* - Header 1
  - Header 2
* - Cell 1

    2nd paragraph cell 1
  - Cell 2
* - Cell 3
  - Cell 4
```
````

[118] https://docutils.sourceforge.io/docs/ref/rst/directives.html#list-table

Notes

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td><pre>```{note}
A note.
```</pre></td><td><b>Note:</b><br>A note.</td></tr>
<tr><td><pre>```{tip}
A tip.
```</pre></td><td><b>Tip:</b><br>A tip.</td></tr>
<tr><td><pre>```{important}
Important information
```</pre></td><td><b>Important:</b><br>Important information.</td></tr>
<tr><td><pre>```{caution}
This might damage your hardware!
```</pre></td><td><b>Caution:</b><br>This might damage your hardware!</td></tr>
</table>

Adhere to the following conventions:

- Use notes sparingly.
- Only use the following note types: `note`, `tip`, `important`, `caution`
- Only use a caution if there is a clear hazard of hardware damage or data loss.

Images

Input	Output
`![Alt text](https://assets.ubuntu.com/v1/b3b72cb2-canonical-logo-166.png)`	

```` ```{figure} https://assets.ubuntu.com/v1/b3b72cb2-canonical-logo-166.png ````
`   :width: 100px`
`   :alt: Alt text`
 	
`   Figure caption`
```` ``` ````
	Fig. 2: Figure caption

Adhere to the following conventions:

- For local pictures, start the path with / (for example, `/images/image.png`).
- Use `PNG` format for screenshots and `SVG` format for graphics.
- See Five golden rules for compliant alt text[119] for information about how to word the alt text.

Reuse

A big advantage of MyST in comparison to plain Markdown is that it allows to reuse content.

Substitution

To reuse sentences or paragraphs that have little markup and special formatting, use substitutions[120].

Substitutions can be defined in the following locations:

[119] https://abilitynet.org.uk/news-blogs/five-golden-rules-compliant-alt-text
[120] https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html#substitutions

- Globally, in a file named `reuse/substitutions.yaml` that is loaded into the `myst_substitutions`[121] variable in `custom_conf.py`:

Listing 5:

```
import os
import yaml

...

if os.path.exists('./reuse/substitutions.yaml'):
  with open('./reuse/substitutions.yaml', 'r') as fd:
      myst_substitutions = yaml.safe_load(fd.read())
```

Listing 6:

```
# Key/value substitutions to use within the Sphinx doc.
{version_number: "0.1.0",
 formatted_text: "*Multi-line* text\n that uses basic **markup**.",
 site_link: "[Website link](https://example.com)"}
```

- Locally, putting the definitions at the top of a single file in the following format:

```
---
myst:
  substitutions:
    version_number: "0.1.0"
    formatted_text: "*Multi-line* text
                     that uses basic **markup**."
    advanced_reuse_key: "This is a substitution that includes a code block:
                       ```
                       code block
                       ```"

---
```

You can combine both options by defining a default substitution in `reuse/substitutions.py` and overriding it at the top of a file.

The definitions from the above examples are rendered as follows:

Input	Output
`{{version_number}}`	0.1.0
`{{formatted_text}}`	*Multi-line* text that uses basic **markup**.
`{{site_link}}`	Website link[122]
`{{advanced_reuse_key}}`	This is a substitution that includes a code block: `code block`

[121] https://myst-parser.readthedocs.io/en/v0.13.5/using/syntax-optional.html#substitutions-with-jinja2

Adhere to the following convention:

- Substitutions do not work on GitHub. Therefore, use substitution names that indicate the included content (for example, `note_not_supported` instead of `reuse_note`).

File inclusion

To reuse longer sections or text with more advanced markup, you can put the content in a separate file and include the file or parts of the file in several locations.

To select parts of the text in a file, use `:start-after:` and `:end-before:` if possible. You can combine those with `:start-line:` and `:end-line:` if required (if the same text occurs more than once). Using only `:start-line:` and `:end-line:` is error-prone though.

You cannot put any targets into the content that is being reused (because references to this target would be ambiguous then). You can, however, put a target right before including the file.

By combining file inclusion and substitutions, you can even replace parts of the included text.

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>

````
% Include parts of the content from
% file rst-syntax-reference.rst
```{include} rst-syntax-reference.rst
    :start-after: "Adhere to the following conventions:"
    :end-before: "  Use the ones specified above."
```
````

</td><td>

- Do not use consecutive headings without intervening text.
- Be consistent with the characters you use for each level.

</td></tr>
</table>

Adhere to the following convention:

- File inclusion does not work on GitHub. Therefore, always add a comment linking to the included file.
- Files that only contain text that is reused somewhere else should be placed in the `reuse` folder and end with the extension `.txt` to distinguish them from normal content files.
- To make sure inclusions don't break, consider adding HTML comments (`<!-- some comment -->`) to the source file as markers for starting and ending.

Tabs

The recommended way of creating tabs is to use the tabs that the Sphinx design[123] extension provides.

[122] https://example.com
[123] https://sphinx-design.readthedocs.io/en/latest/

Input	Output

``{tab-set}
 	
``{tab-item} Tab 1
:sync: key1
 	
Content Tab 1
``
 	
``{tab-item} Tab 2
:sync: key2
 	
Content Tab 2
``
 	
``
	Tab 1
	
	Content Tab 1
	
	Tab 2
	
	Content Tab 2

Alternatively, you can use the Sphinx tabs[124] extension, which is also enabled by default. This was previously recommended due to limitations in Sphinx Design that are now fixed.

Input	Output

``{tabs}
 	
``{group-tab} Tab 1
 	
Content Tab 1
``
 	
``{group-tab} Tab 2
 	
Content Tab 2
``
 	
``
	Tab 1
	Tab 2
	Content Tab 1
	Content Tab 2

[124] https://sphinx-tabs.readthedocs.io/en/latest/

Collapsible sections

There is no support for details sections in MyST, but you can insert HTML to create them.

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>

```
<details>
<summary>Details</summary>

Content
</details>
```

</td><td>Content</td></tr>
</table>

Glossary

You can define glossary terms in any file. Ideally, all terms should be collected in one glossary file though, and they can then be referenced from any file.

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>

````
```{glossary}

MyST example term
  Definition of the example term.
```
````

</td><td>MyST example term
Definition of the example term.</td></tr>
<tr><td><code>{term}`MyST example term`</code></td><td>MyST example term</td></tr>
</table>

More useful markup

Input	Output	Description
```{versionadded} X.Y ```	Added in version X.Y.	Can be used to distinguish between different versions.
---	A horizontal line	Can be used to visually divide sections on a page.
<!-- This is a comment -->		Not visible in the output.
{abbr}`API (Application Programming Interface)`	API	Hover to display the full term.
{spellexception}`PurposelyWrong`		Explicitly exempt a term from the spelling check.

4. In this documentation

Tutorial **Get started** - use Sphinx and Read the Docs to host and test your documentation.

Tutorials (page 3) How-to guides **Step-by-step guides** - learn key operations and customisation.

How-to guides (page 12) Reference **Technical information** - understand the automatic checks and Sphinx capabilities.

Reference (page 40)