Canonical

Documentation starter pack

Canonical

Contents

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. 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** - review the automatic checks and Sphinx capabilities.

Reference (page 65) Explanation **Concepts** - understand the design and architecture of the starter pack.

Explanation (page 109)

1.1. Tutorials

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

1.1.1. Set up the documentation repository

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

Copy the starter pack

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

If you're adding documentation to an existing software project, copy the following files from the starter pack repository into your project:

- 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

Remove the unneeded files

Next, review the starter pack files and remove those that could interfere with your project.

Remove the files that can't be reused:

- `.github/CODEOWNERS`
- `.github/workflows/test-starter-pack.yml`

Review and remove the GitHub workflows in `.github/workflows/` that your project might not need:

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

- `cla-check.yml` verifies whether contributors have signed the Canonical License Agreement[3]. All Canonical projects require this check, so if you're adding docs to an existing Canonical project that already has it, remove this workflow.
- `sphinx-python-dependency-build-checks.yml` verifies Python dependencies for the documentation system. If your project has its own dependency checks, remove this workflow.
- `markdown-style-checks.yml` runs the built-in Markdown linter. If your project already validates its Markdown files, remove this workflow.

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/`.

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 14) contains further guidance.

Pre-commit hooks (optional)

Use pre-commit[4] 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/`:

```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
```

(continues on next page)

[3] https://canonical.com/legal/contributors
[4] https://pre-commit.com/

(continued from previous page)

```
        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
```

1.1.2. 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[5] installed on your machine
- Spread[6] installed on your machine

[5] https://documentation.ubuntu.com/multipass/latest/how-to-guides/install-multipass/
[6] https://github.com/canonical/spread

Warning:

Spread requires elevated permissions to run as root, and this tutorial will not work if Spread is installed as a snap. Use the Go install method recommended in the Spread README to install Spread.

What you'll do

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

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

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

(continues on next page)

(continued from previous page)

```
  # [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.

Create the tutorial file

Now we have everything we need to create the tutorial file itself. ReStructuredText (.rst)[7] is used for the tutorial file format; MyST-Markdown[8] 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`.

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:

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

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!

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.

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[9] 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

```
project: spread_test_example

path: /spread_test_example

backends:
  multipass:
    type: adhoc
    allocate: |
      multipass_image=24.04
```

(continues on next page)

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

(continued from previous page)

```
      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}"

    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.

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.

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:

```
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:

```
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):
```

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[10]
- Spread tests included in Charmcraft documentation[11]

1.2. How-to guides

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

1.2.1. Basic setup and maintenance

Set up, configure, upgrade, and customize your project to keep it organized and aligned with your documentation needs.

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[12] and How to publish documentation on Read the Docs[13] 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 33).
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[14].
4. Use the manual import[15] 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`

[10] https://github.com/canonical/rockcraft/tree/main/docs/tutorial/code
[11] https://github.com/canonical/charmcraft/tree/main/docs/tutorial/code
[12] https://library.canonical.com/documentation/read-the-docs-at-canonical
[13] https://library.canonical.com/documentation/publish-on-read-the-docs
[14] https://docs.readthedocs.com/platform/stable/guides/connecting-git-account.html
[15] https://readthedocs.com/dashboard/import/manual/

- `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 public key from Read the Docs and add it as a deploy key to the repository, then re-run the build in Read the Docs.

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.

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[16].

See How to manually configure a Git repository integration[17] if you want to set up the webhook manually.

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.

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.

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

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[18] extension.
Changes to files that are part of the starter pack, for example, *Automatic checks* (page 65), might require you to manually update your repository with the required files. See the starter pack's change log[19] for the most relevant (and of course all breaking) changes.

[18] https://github.com/canonical/canonical-sphinx
[19] 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.

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.

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.

Optional customisation

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

Modify the template

The default starter pack templates provide an initial configuration for your documentation set, including:

- Header template - The top section of the page that contains your product's tag image and name, a link to your product's page (if available), and a drop-down menu for "More resources".
- Footer template - The bottom section of the page that contains sequential navigation controls, copyright information, licensing details, and other relevant links.

These project settings are configured in `docs/conf.py` and are generally sufficient for most cases. However, if you have additional requirements – such as adding links to announcements or videos that are not part of the documentation – you can override the default templates to customize them as needed.

See the *Use custom templates* (page 62) guide for details on how to do so.

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`.

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`
- `notfound.extension`
- `sphinx.ext.intersphinx`
- `sphinx_config_options`
- `sphinx_contributor_listing`
- `sphinx_copybutton`
- `sphinx_design`
- `sphinx_filtered_toctree`
- `sphinx_last_updated_by_git`
- `sphinx_related_links`
- `sphinx_reredirects`
- `sphinx_roles`
- `sphinx_sitemap`
- `sphinx_tabs.tabs`
- `sphinx_terminal`
- `sphinx_ubuntu_images`
- `sphinx_youtube_links`
- `sphinxcontrib.cairosvgconverter`
- `sphinxcontrib.jquery`
- `sphinxext.opengraph`

The `canonical_sphinx` extension is required for the starter pack and provides the Furo-based theme and custom templates.

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 `docs/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].

[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/

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]
- 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 `docs/requirements.txt` file.

Disable failure on warning

The docs build (`make html`) is, by default, set to fail when a warning (`WARNING` in the build log) is encountered. To disable this setting, remove the `--failure-on-warning` option from the command specified in the `html` target in the `Makefile`.

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.

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 `docs/requirements.txt` file.

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

[23] https://www.sphinx-doc.org/en/master/usage/configuration.html
[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

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

Sitemap configuration

The Sphinx starter pack's configuration file (`docs/conf.py`) includes default sitemap configuration.

The `sphinx-sitemap` extension requires a `html_baseurl` variable to be configured. This is set as follows:

```
html_baseurl = os.environ.get("READTHEDOCS_CANONICAL_URL", "/")
```

When building on Read the Docs, this sets `html_baseurl` dynamically to the value of the `READTHEDOCS_CANONICAL_URL` environment variable, which resolves to the full URL of the documentation including the version and language (if applicable).

In local builds and builds on other hosts, `html_baseurl` defaults to `/`.

The `sitemap_url_scheme` variable is set to `'{link}'` by default. This uses the value of `html_baseurl` to generate the full URL for each page for the sitemap.

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[27] and a custom `robots.txt` pointing to it.

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

`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
```

Exclude pages

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

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

Wildcards are supported. For example, `_modules/*` excludes the path `_modules/` and all paths such as `_modules/foo/bar/`. For details, see Excluding Pages[28].

[28] https://sphinx-sitemap.readthedocs.io/en/latest/advanced-configuration.html#excluding-pages

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.

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/stable/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:

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

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

Customise PDF output

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.

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

For basic instructions about building the PDF, see *Build and preview* (page 33).

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 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.

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

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,
   ),
]
```

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
```

(continues on next page)

(continued from previous page)

```
# 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`.

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.

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

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:

```
{
   ...
   '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
```

(continues on next page)

(continued from previous page)

```
.. only:: latex

   .. raw:: latex

      \end{landscape}
```

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

Related

- *Build and preview* (page 33)
- *Customise the setup* (page 14)

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.

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]

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 *Copy the starter pack* (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.

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 30) section.

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

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/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`.

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 14).

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.

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
```

(continues on next page)

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

(continued from previous page)

```
│   │   ├── ubuntu-font-licence-1.0.txt
│   │   ├── UbuntuMono-B.ttf
│   │   ├── UbuntuMono-RI.ttf
│   │   ├── 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                     # removed
│   └── scripts
│       ├── 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`
├── init.sh                     # removed
├── make.bat                    # removed
├── Makefile                    # moved to `docs/Makefile`
├── Makefile.sp                 # removed
```

(continues on next page)

(continued from previous page)

```
└── readme.rst                  # renamed to `README.rst`
```

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.

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 `docs/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`

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 33).

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 33) 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.

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 8: 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:

[38] https://diataxis.fr/

Listing 9: 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:

```
.. 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>`.

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 73)
- *MyST syntax guide* (page 92)

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.

[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/

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]

Troubleshoot issues

This page provides guidance to resolve issues with the Starter Pack and ReadTheDocs that are difficult to identify or that we don't expect to be solved.

Stable version won't build from the latest tag

If your project has the `stable` version configured to build from tags, such as with the default semantic versioning behavior[46], your `stable` version can become out-of-step and continue building a particular tag, even when the repository has newer tags.

Possible causes

An unwanted tag might have been pushed to the repository and then removed. Once ReadTheDocs creates a version from a tag, it doesn't later verify that the tag still exists, so the version will persist and become a zombie.

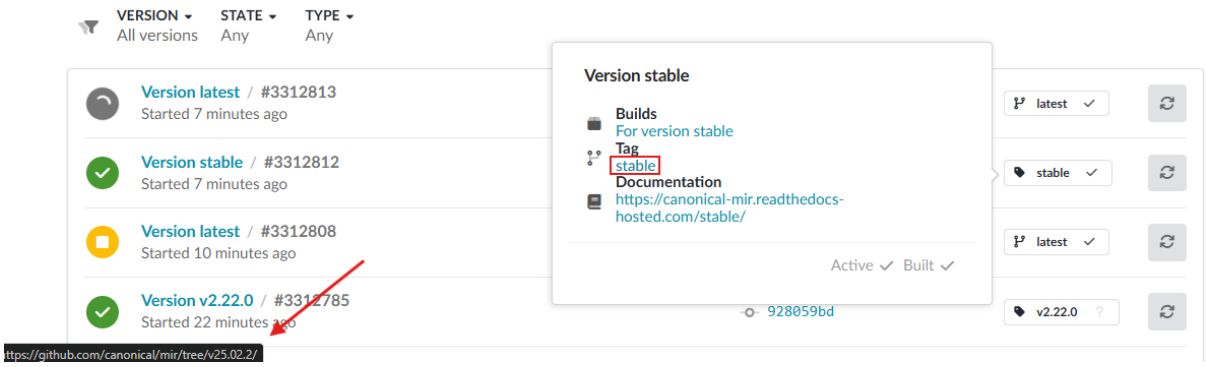
If the unwanted tag is a higher iterator than any existing tag, the zombie version will always take precedence. For example, if tag `20.2.0` was pushed by accident and then replaced with `2.20.0`, the corresponding version 20.2.0 will persist, and `stable` will continue pointing to it.

Diagnosis

There's a roundabout procedure to verify whether your project is affected. Start by opening your project dashboard on ReadTheDocs.

On the **Builds** tab, locate the most recent `stable` build. For that version, hover over the status indicator. In the hover box, open the **stable** link. If the resulting GitHub page is a 404, then your project has a zombie version.

[43] https://diataxis.fr/
[44] https://canonical-example-product-documentation.readthedocs-hosted.com/
[45] https://github.com/canonical/example-product-documentation
[46] https://docs.readthedocs.com/platform/stable/versions.html#versions-are-git-tags-and-branches



Resolution

On your project dashboard, open the **Versions** tab and click **Add version**.

Find the zombie version, deactivate it, then update it.

Rebuild `stable` by retriggering it on the dashboard or pushing a new tag to the repository.

Issue tracking

readthedocs/readthedocs.org#12450[47]

1.2.2. Optional features and workflows

As your documentation grows, you may need more advanced features to support richer content. This can include diagrams as code, automated API references, and data tables.

While some of these features are available by default in the starter pack, others may require additional extensions. The following guides will help you get started:

Bridge project and docs builds

The starter pack can be used as a standalone docs repository, or embedded inside a parent project. This guide demonstrates how to bridge the docs build with a Python project's main build. Once bridged, project contributors can install, build, and check the docs from the root of the project with the main build system.

Parent projects and the build (page 109) describes the full benefits of bridging the build in a larger project.

[47] https://github.com/readthedocs/readthedocs.org/issues/12450

Plan and requirements

The bridge is built by making up to three changes to the build.

The bridge **shims the docs build targets in the main build**. Any build system is capable of adding targets that call other systems. When shimmed, the docs targets like `html` and `clean` pass through to `docs/Makefile`, with arguments.

The bridge also **merges the virtual environments**, removing the need for a separate docs environment. This change is optional but recommended. To combine environments, your project must provide **Python 3.11** or higher to the starter pack. Any Python dependency manager will do, and this guide illustrates with three:

- pip 25.1 and higher
- uv 0.4.27 and higher
- Poetry 1.2.0a2 and higher

After adding the bridge, it's also possible to **adjust the documentation workflows** to use your project's main build. The workflows were designed with Make, so they only work if you use it for your build system.

Example project layout

This guide illustrates the bridge through an example project. In the example project, the file tree contains:

```
Project
│
├── ...
├── docs
│   └── Makefile
├── .venv
├── Makefile
└── pyproject.toml
```

The example project's root `Makefile` has two conventional targets that need adjustment:

- `setup` for building the virtual environment and syncing dependencies
- `clean` for cleaning up the virtual environment and temporary files

Set the build paths

Where your main build sets environment variables, redeclare the docs environment variables that specify the build paths:

- `BUILDDIR` is the destination for the docs. If you have special distribution needs you can override this, but for most builds this can be left as-is.
- `VENVDIR` is the virtual environment of the docs. If you're merging the virtual environments, set this as a relative path from the docs directory to your project's virtual environment.

- `VALEDIR` is the path to the Vale binary. The full path depends on the location of your virtual environment, so it's best to copy this as-is.

In the example project, this looks like:

Listing 10: Makefile

```
# Env vars for the docs build
export BUILDDIR ?= _build
export VENVDIR ?= ../.venv
export VALEDIR ?= $(VENVDIR)/lib/python*/site-packages/vale
```

Integrate the docs setup

The next step is to incorporate the docs installation target, and optionally the dependencies, into the main build.

Separate virtual environments

If you don't plan to merge the virtual environments, override the installation target by calling all three doc installation targets in a row.

In the example project, this is written as:

Listing 11: Makefile

```
# Override the
.PHONY: docs-install
docs-install:
    $(MAKE) -C docs install --no-print-directory
    $(MAKE) -C docs vale-install --no-print-directory
    $(MAKE) -C docs pymarkdownlnt-install --no-print-directory
```

Merged virtual environments

To merge virtual environments, you make the main `setup` target handle both development and docs packages, and enumerate all docs packages in `pyproject.toml`.

By adding dependency groups, the docs packages, plus any custom Sphinx extensions, can be managed by the main build and stored in one virtual environment. The result will be three dependency groups in `pyproject.toml`:

- `dev` for development builds
- `docs` for extra docs packages that your project needs
- `docs-starter-pack` for the core docs packages set by the starter pack

First, add these dependency groups, and make the docs dependencies include the starter pack packages:

Listing 12: pyproject.toml

```
[dependency-groups]
dev = [
    # Packages for main development and testing
]
docs = [
    # Packages for extra docs features
    {include-group = "docs-starter-pack"},
]
docs-starter-pack = [
    # Core docs packages
]
```

If you don't already have a `dev` dependency group, review the packages listed in the file's `dependencies` key, then move any non-runtime dependencies to the `dev` dependency group.

If your project needs extra docs features, like the Mermaid or LaTeX Sphinx extensions, add their packages to the `docs` group.

Copy the contents of `docs/requirements.txt` into the `docs-starter-pack` group.

In the main build, override the docs installation target and make the project's `setup` target depend on it. In the example project, it is written like this:

pip

uv

Poetry

Listing 13: Makefile

```
.PHONY: setup
setup: docs-install
    pip install --group dev

# ...

# Override for `install` target in docs project
.PHONY: docs-install
docs-install:
    pip install --group docs
    $(MAKE) -C docs vale-install --no-print-directory
    $(MAKE) -C docs pymarkdownlnt-install --no-print-directory
```

Listing 14: Makefile

```
.PHONY: setup
setup: docs-install
    uv sync --group dev

# ...

# Override for `install` target in docs project
.PHONY: docs-install
docs-install:
```

(continues on next page)

(continued from previous page)

```
    uv sync --no-dev --group docs
    $(MAKE) -C docs vale-install --no-print-directory
    $(MAKE) -C docs pymarkdownlnt-install --no-print-directory
```

Listing 15: Makefile

```
.PHONY: setup
setup: docs-install
    poetry install --only dev

# ...

# Override for `install` target in docs project
.PHONY: docs-install
docs-install:
    poetry install --only docs
    $(MAKE) -C docs vale-install --no-print-directory
    $(MAKE) -C docs pymarkdownlnt-install --no-print-directory
```

If your docs aren't written in Markdown, remove the command that runs the `pymarkdownlnt-install` target.

Shim the remaining targets

The docs build has many targets, but only a handful of them overlap or collide with most project builds, so we only need to override two more. The rest can pass straight through to the docs build.

In the example project, the main build calls the targets like this:

Listing 16: Makefile

```
# Override for `clean` target in docs project. We don't want to touch `.venv`,
# so we pass a null dir instead.
.PHONY: docs-clean
docs-clean:
    VENVDIR=null $(MAKE) -C docs clean --no-print-directory

# Override for `help` target
.PHONY: docs-help
docs-help:
    @echo "Commands in the documentation subproject:"
    $(MAKE) -C docs help --no-print-directory
    @echo "Run these commands with 'make docs-<command>' in the project root."

# Shim for the rest of the targets in docs Makefile
.PHONY: docs-%
docs-%: docs-install
    $(MAKE) -C docs $(@:docs-%=%) --no-print-directory
```

Adjust the Read the Docs build

With the Makefile in a different location than usual, and its being a separate process, it's simplest to override the Read The Docs build in `.readthedocs.yaml` to call the same build targets that developers use locally.

If you use an uncommon system, you might need to install it during the workflow's `create_environment` job.

If you merged the virtual environments, make sure to set `VENVDIR=${READTHEDOCS_VIRTUALENV_PATH}` in all commands.

Here's what it looks like in the example project:

Listing 17: .readthedocs.yaml

```
build:
  os: ubuntu-24.04
  tools:
    python: "3.12"
  jobs:
    post_checkout:
      - git fetch --tags --unshallow # Also fetch tags
    create_environment:
      - python3 -m venv "${READTHEDOCS_VIRTUALENV_PATH}"
    install:
      - make docs-install VENVDIR="${READTHEDOCS_VIRTUALENV_PATH}"
    build:
    html:
      - make docs VENVDIR="${READTHEDOCS_VIRTUALENV_PATH}" BUILDDIR="$READTHEDOCS_OUTPUT/
html/"
```

Adjust the doc workflows

If your project uses the starter pack's docs workflows *and* Make, adjust the workflows to use the bridged targets.

For the main checks, override the target names and paths through the workflow inputs[48]:

Listing 18: .github/workflows/automatic-doc-checks.yml

```
jobs:
  documentation-checks:
    uses: canonical/documentation-workflows/.github/workflows/documentation-checks.
yaml@main
    with:
      working-directory: "."
      fetch-depth: 0
      install-target: docs-install
      spelling-target: docs-spelling
      woke-target: docs-woke
      linkcheck-target: docs-linkcheck
      pa11y-target: docs-pa11y
```

If your docs are written in Markdown, override the path and command inputs in the Markdown linter workflow:

Listing 19: .github/workflows/markdown-style-checks.yml

```
- name: Create venv
  working-directory: "."
  run: make docs-install
- name: Lint markdown
  working-directory: "."
  run: make docs-lint-md
```

[48] https://github.com/canonical/documentation-workflows/blob/main/.github/workflows/documentation-checks.yaml#L5-L54

Create diagrams as code using Mermaid

Diagrams help users quickly understand and visualize complex ideas, but they can easily become outdated and difficult to maintain. Creating diagrams as code solves this by keeping them alongside the software source, making updates and reviews simpler.

Mermaid is a popular choice that allows diagrams to be created, maintained, and updated directly in the documentation. This guide explains how to set up the sphinxcontrib-mermaid[49] extension and use Mermaid diagrams in your documentation, with examples included.

Note:

While there are many other tools and/or approaches for creating diagrams in visualizations in your documentation (e.g. C4 model[50], Dia[51], PlantUML[52], Structurizr[53], etc), we only provide support for `sphinxcontrib-mermaid` in the starter pack.

[50] https://c4model.com/
[51] http://dia-installer.de/
[52] https://plantuml.com/
[53] https://structurizr.com/

Installation and setup

First, add the "sphinxcontrib-mermaid" extension to `requirements.txt` so that it's installed as part of your Sphinx project dependencies:

```
canonical-sphinx[full]
packaging
sphinxcontrib-svg2pdfconverter[CairoSVG]
sphinx-last-updated-by-git
sphinx-sitemap
sphinxcontrib-mermaid
```

Then add "sphinxcontrib.mermaid" in the "extensions" list in `conf.py`:

```
extensions = [
    [...]
    "sphinxcontrib.mermaid",
]
```

You are now ready embed Mermaid diagrams and visualization in your documentation using the `mermaid` directive.

[49] https://sphinxcontrib-mermaid-demo.readthedocs.io/en/latest/index.html

Optional configuration

You can further configure Mermaid's default settings in your project's `conf.py`, such as specifying the image output format (e.g., "png", "raw"), enabling zoom on diagrams, or pinning the Mermaid version[54] used for rendering.

See Mermaid configuration values[55] for more information.

Add a new diagram

Use the `mermaid` directive to embed a Mermaid diagram into your documentation. You start by declaring the type of diagram (e.g. "flowchart", "sequenceDiagram", "timeline", etc), followed by definition and contents. It is also possible to specify additional configuration or custom styles, depending on the diagram type.

Some examples will be covered below.

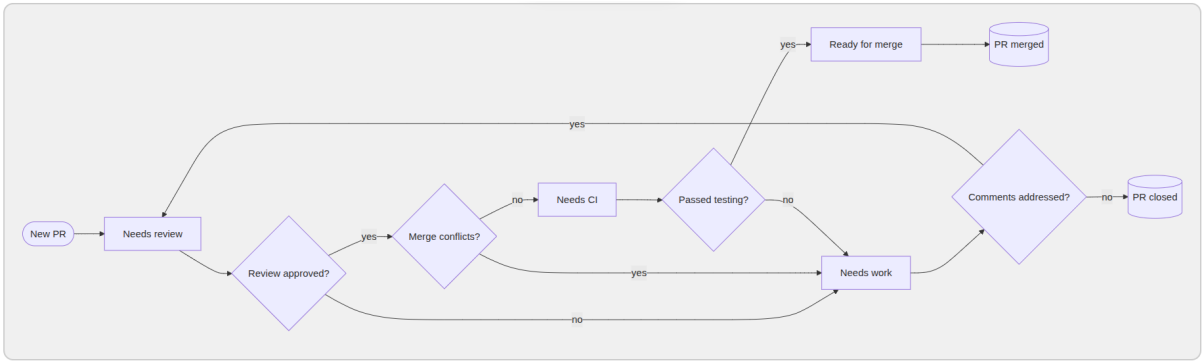
See also

See the Mermaid - Diagram syntax[56] reference for details on the syntax and customization options for each diagram type.

Flowchart diagram with default settings

The left-to-right flowchart below uses the default Mermaid settings.

Diagram



[54] https://unpkg.com/browse/mermaid/
[55] https://sphinxcontrib-mermaid-demo.readthedocs.io/en/latest/#config-values
[56] https://mermaid.js.org/intro/syntax-reference.html

MyST

````
```{mermaid}
flowchart LR
    A([New PR])
    B[Needs review]
    C{Review approved?}
    D{Merge conflicts?}
    E[Needs work]
    F[Needs CI]
    G{Passed testing?}
    H[Ready for merge]
    I[(PR merged)]
    J{Comments addressed?}
    K[(PR closed)]

    A --> B
    B --> C
    C -- no --> E
    C -- yes --> D
    D -- yes --> E
    D -- no --> F
    F --> G
    G -- yes --> H
    G -- no --> E
    H --> I
    E --> J
    J -- yes --> B
    J -- no --> K
```
````

rST

```
.. mermaid::

   flowchart LR
       A([New PR])
       B[Needs review]
       C{Review approved?}
       D{Merge conflicts?}
       E[Needs work]
       F[Needs CI]
       G{Passed testing?}
       H[Ready for merge]
       I[(PR merged)]
       J{Comments addressed?}
       K[(PR closed)]

       A --> B
       B --> C
       C -- no --> E
       C -- yes --> D
       D -- yes --> E
       D -- no --> F
       F --> G
```

(continues on next page)

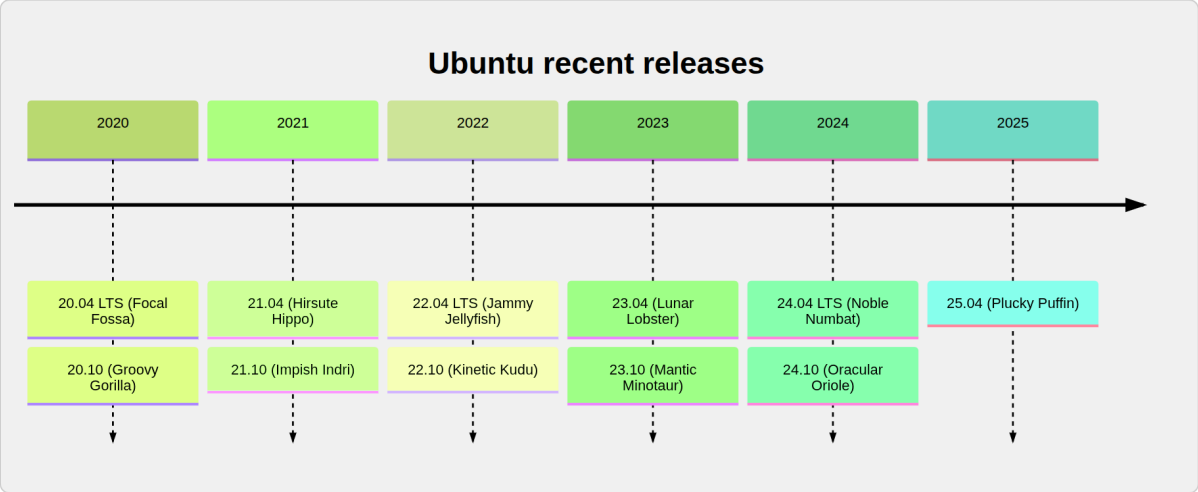
(continued from previous page)

```
      G -- yes --> H
      G -- no --> E
      H --> I
      E --> J
      J -- yes --> B
      J -- no --> K
```

Timeline diagram with pre-defined theme

The timeline diagram below uses a pre-defined Mermaid theme[57].

Diagram



MyST

```
```{mermaid}
%%{init: {"theme":"forest"}}%%
timeline
    title Ubuntu recent releases

    2020 : 20.04 LTS (Focal Fossa) : 20.10 (Groovy Gorilla)
    2021 : 21.04 (Hirsute Hippo) : 21.10 (Impish Indri)
    2022 : 22.04 LTS (Jammy Jellyfish) : 22.10 (Kinetic Kudu)
    2023 : 23.04 (Lunar Lobster) : 23.10 (Mantic Minotaur)
    2024 : 24.04 LTS (Noble Numbat) : 24.10 (Oracular Oriole)
    2025 : 25.04 (Plucky Puffin)
```
```

[57] https://mermaid.js.org/syntax/timeline.html#themes

rST

```
.. mermaid::

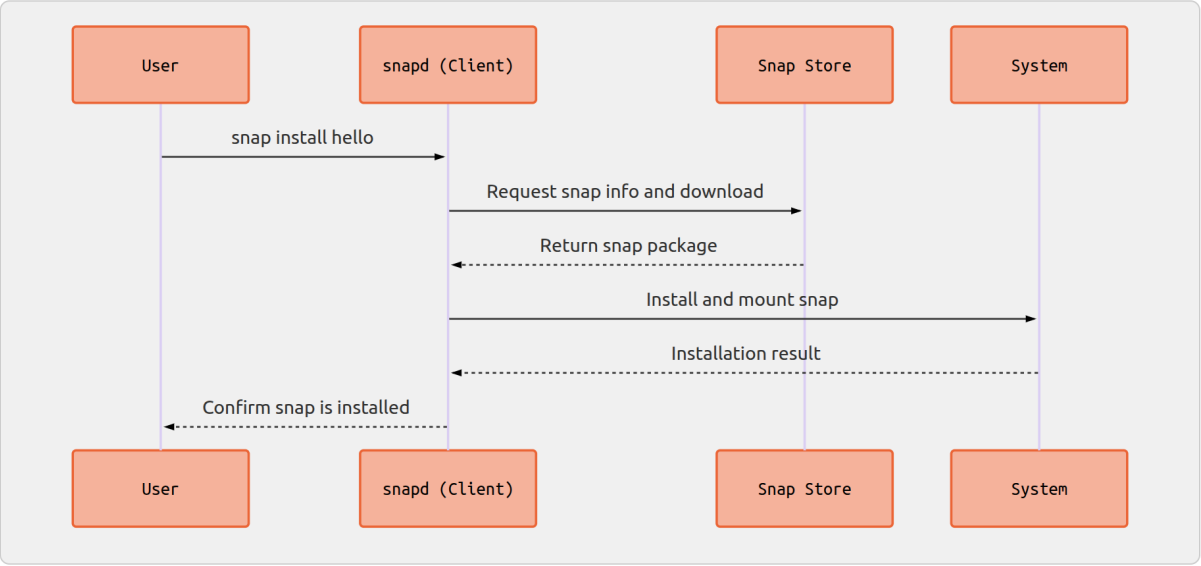
   %%{init: {"theme":"forest"}}%%
   timeline
       title Ubuntu recent releases

       2020 : 20.04 LTS (Focal Fossa) : 20.10 (Groovy Gorilla)
       2021 : 21.04 (Hirsute Hippo) : 21.10 (Impish Indri)
       2022 : 22.04 LTS (Jammy Jellyfish) : 22.10 (Kinetic Kudu)
       2023 : 23.04 (Lunar Lobster) : 23.10 (Mantic Minotaur)
       2024 : 24.04 LTS (Noble Numbat) : 24.10 (Oracular Oriole)
       2025 : 25.04 (Plucky Puffin)
```

Sequence diagram with global custom CSS

The sequence diagram below has custom styling applied using a global CSS file. A global CSS file enables the styles to be easily applied to all sequence diagrams, based on the classes defined in your stylesheet. You can also use the global CSS file to customize the diagrams in dark mode.

Diagram



Custom CSS

First, create a Mermaid stylesheet in the `_static` directory and customize it as needed. For example, `mermaid-sequence.css`.

Then, enable custom stylesheets in `conf.py`. Uncomment this line so Sphinx can use local stylesheets:

Listing 20: conf.py

```
html_static_path = ["_static"]
```

Link to your custom Mermaid stylesheet:

Listing 21: conf.py

```
html_css_files = [
    "mermaid-sequence.css",
]
```

Here's an example stylesheet that changes the default typefaces and colors of Mermaid components. It also sets a solid background color for all Mermaid diagrams.

Listing 22: mermaid-sequence.css

```
.actor {
  stroke: #eb6536 !important;
  stroke-width: 2;
  fill: #f5b29b !important;
}

text.actor {
  fill: black;
  stroke: none;
  font-family: Helvetica;
}

.actor-line {
  stroke: #77216f;
}

.mermaid {
    background-color: #f0f0f0;
    border: 1px solid #ccc;
    border-radius: 6px;
    padding: 0.5em;
}
```

MyST

````
```{mermaid}
sequenceDiagram
    participant User
    participant Snap as snapd (Client)
    participant Store as Snap Store
    participant System

    User->>Snap: snap install hello
    Snap->>Store: Request snap info and download
    Store-->>Snap: Return snap package
    Snap->>System: Install and mount snap
    System-->>Snap: Installation result
    Snap-->>User: Confirm snap is installed
```
````

rST

```
.. mermaid::

   sequenceDiagram
       participant User
       participant Snap as snapd (Client)
       participant Store as Snap Store
       participant System

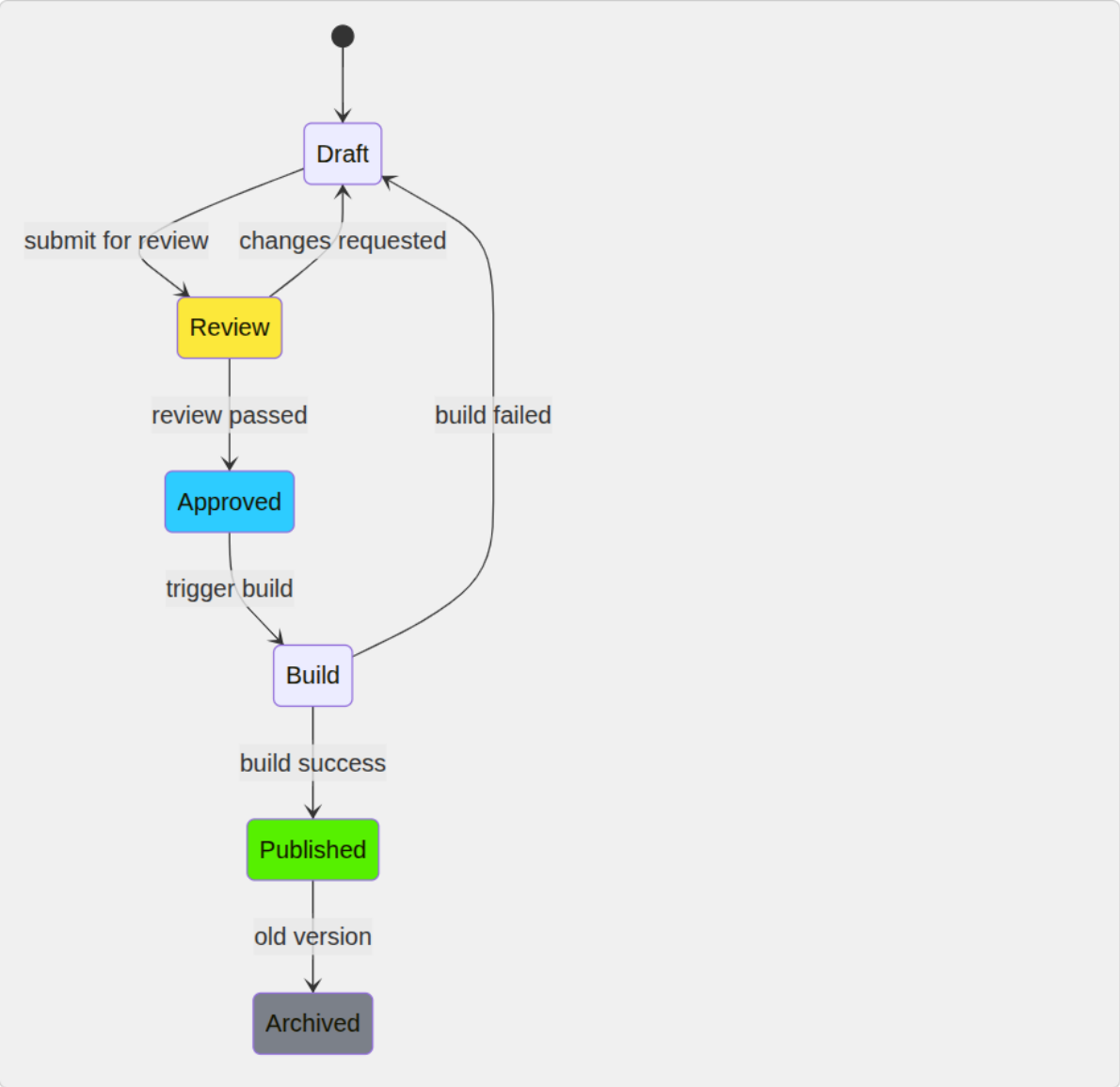
       User->>Snap: snap install hello
       Snap->>Store: Request snap info and download
       Store-->>Snap: Return snap package
       Snap->>System: Install and mount snap
       System-->>Snap: Installation result
       Snap-->>User: Confirm snap is installed
```

State diagram with image-specific styles

The state diagram below has image-specific custom styling applied using the `classDef` keyword[58].

[58] https://mermaid.js.org/syntax/stateDiagram.html#styling-with-classdefs

Diagram



MyST

```
```{mermaid}
stateDiagram-v2
    classDef review fill:#FCE83A
    classDef approved  fill:#2DCCFF
    classDef published fill:#56F000
    classDef archived fill:#7B8089

    [*] --> Draft
    Draft --> Review : submit for review
    Review --> Draft : changes requested
    Review --> Approved : review passed
    Approved --> Build : trigger build
    Build --> Published : build success
    Build --> Draft : build failed
```

(continues on next page)
```

(continued from previous page)

```
    Published --> Archived : old version

    class Review review
    class Approved approved
    class Published published
    class Archived archived
```
```

### rST

```
.. mermaid::

    stateDiagram-v2
        classDef review fill:#FCE83A
        classDef approved  fill:#2DCCFF
        classDef published fill:#56F000
        classDef archived fill:#7B8089

        [*] --> Draft
        Draft --> Review : submit for review
        Review --> Draft : changes requested
        Review --> Approved : review passed
        Approved --> Build : trigger build
        Build --> Published : build success
        Build --> Draft : build failed
        Published --> Archived : old version

        class Review review
        class Approved approved
        class Published published
        class Archived archived
```

Projects using Mermaid

Here are some Canonical projects that use Mermaid for diagramming in their documentation:

- Checkbox[59]
- cloud-init[60]
- Charmed MySQL K8s[61]
- Ubuntu Pro[62]

[59] https://canonical-checkbox.readthedocs-hosted.com/stable/explanation/remote/
[60] https://docs.cloud-init.io/en/latest/explanation/boot.html
[61] https://canonical-charmed-mysql-k8s.readthedocs-hosted.com/explanation/flowcharts/#
[62] https://documentation.ubuntu.com/pro/support-overview/
```

Related topics

- See the Official Mermaid documentation[63] for setup and configuration instructions, full syntax for the different types of Mermaid diagrams.
- Try out the Mermaid Live Editor[64] if you want a playground to work on your diagrams.

Create data tables

The starter pack can render comma-separated values (CSV) data as tables.

For other table types, see *tables in reStructuredText* (page 84) and *tables in MyST* (page 101).

Create a table from inline data

If you have a small amount of CSV data, you can include the data in the doc source. For example:

Table

Animal	Number of legs	Size
Worm	0	Small
Penguin	2	Medium
Horse	4	Large
Ant	6	Small
Octopus	8	Medium

reST

```
.. csv-table::
   :header: "Animal", "Number of legs", "Size"

   "Worm", 0, "Small"
   "Penguin", 2, "Medium"
   "Horse", 4, "Large"
   "Ant", 6, "Small"
   "Octopus", 8, "Medium"
```

If the table needs it, customize the column widths, character encoding, and so on, as described in the csv-table reference[65].

[63] https://mermaid.js.org/intro/
[64] https://mermaid.live/
[65] https://docutils.sourceforge.io/docs/ref/rst/directives.html#csv-table

MyST

```
```{csv-table}
:header: "Animal", "Number of legs", "Size"

"Worm", 0, "Small"
"Penguin", 2, "Medium"
"Horse", 4, "Large"
"Ant", 6, "Small"
"Octopus", 8, "Medium"
```
```

If the table needs it, customize the column widths, character encoding, and so on, as described in the csv-table reference (MyST)[66].

Create a table from a CSV file

If you have a large amount of CSV data, or the data is generated by an automated process, you can include the data from a file. For example:

Table

Animal	Number of legs	Size
Worm	0	Small
Penguin	2	Medium
Horse	4	Large
Ant	6	Small
Octopus	8	Medium

reST

```
.. csv-table::
   :file: /reuse/animals.csv
   :header-rows: 1
```

[66] https://mystmd.org/guide/directives#directive-csv-table

MyST

```
```{csv-table}
:file: /reuse/animals.csv
:header-rows: 1
```
```

docs/reuse/animals.csv

```
"Animal", "Number of legs", "Size"
"Worm", 0, "Small"
"Penguin", 2, "Medium"
"Horse", 4, "Large"
"Ant", 6, "Small"
"Octopus", 8, "Medium"
```

Create an interactive table

The Sphinx DataTables[67] extension enables interactive tables. Users can sort columns and filter rows. For examples, see the extension's documentation.

The extension isn't available by default in the starter pack.

To include the extension in your documentation, add `sphinx-datatables` to `docs/requirements.txt`. Then add `sphinx_datatables` to the `extensions` list in `docs/conf.py`.

Make a table interactive by adding a special CSS class to the directive:

reST

```
.. csv-table::
   :class: sphinx-datatable
   :file: /reuse/animals.csv
   :header-rows: 1
```

MyST

```
```{csv-table}
:class: sphinx-datatable
:file: /reuse/animals.csv
:header-rows: 1
```
```

[67] https://sharm294.github.io/sphinx-datatables/

Import docstrings with Sphinx `autodoc`

Module and function details are useful reference material to have in documentation, but the process of manually pulling all the necessary details over can become tedious. The Sphinx `autodoc` extension[68] provides the capability to automatically pull in docstrings and module information for Python code.

Prerequisites

To use the Sphinx `autodoc` extension with the Starter Pack, you need:

- Python module files located within the same repository as your documentation

OR

- The code repository added as a Git submodule[69] into the documentation repository

Setup

In the `conf.py` file in your docs directory, update the `sys.path` so that Sphinx can find your module files. At the top of the file, add a `sys.path.insert` that adds your `<code>` directory:

Listing 23: `conf.py`

```
import sys
from pathlib import Path

relative_code_path = Path('..', '<code>')
absolute_code_path = relative_code_path.resolve()
absolute_code_path_str = str(absolute_code_path)
sys.path.insert(0, absolute_code_path_str) # insert at index 0 so it occurs early in the
list
```

Then, further down in the `conf.py`, add `sphinx.ext.autodoc` to the list of extensions:

[68] https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html
[69] https://git-scm.com/book/en/v2/Git-Tools-Submodules

Listing 24: `conf.py`

```
extensions = [
    ...
    "sphinx.ext.autodoc",
]
```

Usage

See Sphinx's `autodoc` instructions[70] for details.

Known issues and limitations

There are a few issues and limitations that should be taken into consideration.

Language

The extension's usage is limited to Python code. There are extensions for some other languages but they have not been tested with the Starter Pack, such as sphinxcontrib-rust[71] for Rust.

Docstring format

The `autodoc` extension pulls the docstrings straight into the the reStructuredText (reST) document, which requires the docstrings to be in reST format. For docstrings in the Numpy or Google style, the napoleon[72] extension can convert the docstrings into reST prior to processing by `autodoc`.

For documentation that is written in MyST Markdown, wrap the `eval-rst` directive around the `autodoc` calls:

````
```{eval-rst}

.. py:currentmodule:: code.<module-name>

.. automodule:: <module-name>
    :members:

```
````

[70] https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html#usage
[71] https://sphinxcontrib-rust.readthedocs.io/en/stable/
[72] https://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html#module-sphinx.ext.napoleon

Canonical examples

Table 1: Canonical autodoc examples

Product	`conf.py`	Raw Doc	Rendered Doc
Jubilant	conf.py[73]	jubilant.rst[74]	jubilant reference page[75]
Ops	conf.py[76]	ops.rst[77]	*ops* reference page[78]

Add OpenAPI integration

Use this guide to add a minimal OpenAPI integration for the starter pack. This implementation uses Swagger UI[79]; it includes a basic specification and an opt-in tag toggle – based on a build-time environment variable – to keep the documentation clean by default.

Locate the specification

The stub OpenAPI document can be found under `docs/how-to/assets/openapi.yaml`. You can use it as a starting point for your API description:

```
openapi: 3.0.3
info:
  title: Hello API
  version: 1.0.0
paths:
  /hello:
    get:
      summary: Say hello
      parameters:
        - in: query
          name: name
          schema:
            type: string
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
```

(continues on next page)

[73] https://github.com/canonical/jubilant/blob/023ce73353352133c43dfb17b4a6cfad0f3e7816/docs/conf.py
[74] https://github.com/canonical/jubilant/blob/023ce73353352133c43dfb17b4a6cfad0f3e7816/docs/reference/jubilant.rst
[75] https://documentation.ubuntu.com/jubilant/reference/jubilant/
[76] https://github.com/canonical/operator/blob/main/docs/conf.py
[77] https://github.com/canonical/operator/blob/main/docs/reference/ops.rst
[78] https://documentation.ubuntu.com/ops/latest/reference/ops/
[79] https://swagger.io/tools/swagger-ui/

(continued from previous page)

```
                    type: string
                    example: "Hello, world!"
```

Enable the viewer

The OpenAPI interactive viewer is disabled by default. Exporting the `OPENAPI` environment variable when building in your terminal makes Sphinx copy the specification to the output directory and add an `openapi` tag that we can use in the documentation source:

```
export OPENAPI=1
```

Then rebuild the docs to pick up the specification:

```
make clean html
```

Tip:

Unset the variable to go back to the default build:

```
unset OPENAPI
```

To enable it for a single command, prefix it like this:

```
OPENAPI=1 make clean html
```

Confirm the viewer works

Serve the documentation locally:

```
make run
```

Navigate to `/how-to/openapi/`. When the feature flag is enabled, the page renders the Swagger UI for the shipped specification right here. The snippet that does this should be included in your documentation source with the `.. raw:: html` directive:

```
   <link rel="stylesheet" type="text/css" href="https://unpkg.com/swagger-ui-dist@5.11.0/
swagger-ui.css" />
   <div id="openapi-hello-swagger"></div>
   <script src="https://unpkg.com/swagger-ui-dist@5.11.0/swagger-ui-bundle.js" charset=
"UTF-8"
       crossorigin="anonymous"></script>
   <script src="https://unpkg.com/swagger-ui-dist@5.11.0/swagger-ui-standalone-preset.js"
charset="UTF-8"
       crossorigin="anonymous"></script>
   <script>
       window.addEventListener('load', function () {
           const specUrl = new URL('/openapi.yaml', window.location.href).toString();
           SwaggerUIBundle({
               url: specUrl,
               dom_id: '#openapi-hello-swagger',
```

(continues on next page)

(continued from previous page)

```
            presets: [SwaggerUIBundle.presets.apis, SwaggerUIStandalonePreset],
            plugins: [],
            deepLinking: true,
            defaultModelsExpandDepth: -1,
            supportedSubmitMethods: []
        });
    });
  </script>
```

Warning:

The interactive viewer is hidden now because `OPENAPI` was not set when this documentation was built. Export `OPENAPI=1` when building the documentation to see it here.

Building on RTD

To enable the OpenAPI viewer on Read the Docs[80] tentatively, define the `OPENAPI` environment variable in your project's admin settings[81].

If you're certain that you want the OpenAPI viewer enabled for all builds, drop all conditional statements from `conf.py` and the reST files. In `conf.py`, replace the following lines:

```
html_extra_path = []

# ...
if os.getenv("OPENAPI", ""):
    tags.add("openapi")
    html_extra_path.append("how-to/assets/openapi.yaml")
```

With this:

```
html_extra_path = ["how-to/assets/openapi.yaml"]
```

Next, remove any tag-based logic from the reST files:

```
.. only:: openapi
```

Troubleshooting

- The `make html` command fails with *unknown tag*: Make sure you exported the environment variable in the same shell session that runs `make`.
- The Swagger UI appears but cannot load the specification: Make sure that `openapi.yaml` exists in the *root* of your build output or adjust the import paths according to your build structure. Also, overriding the `html_extra_path` setting in `conf.py` may interfere with this configuration because it is used to copy the specification to the output directory.

[80] https://readthedocs.org/
[81] https://docs.readthedocs.com/platform/stable/guides/environment-variables.html

Use custom templates

If the default template in the Starter Pack doesn't fully meet your needs – whether you want a unique layout, a custom header or footer, or a specialized sidebar for certain pages – you can create and use a custom template for your Sphinx project.

This guide shows you how to extend or override the default templates in the Starter Pack to tailor the look and structure of your documentation.

Note:

Base template customizations can be made to your documentation. However, they are not officially supported by the team maintaining the starter pack. Use them at your own discretion.

Setup

First, create the `docs/_templates` directory; all your custom templates will need to be stored in this folder.

Then uncomment this line in `docs/conf.py` so your Sphinx project will use local templates (where available):

Listing 25: conf.py

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

In most cases, you will need to copy the default templates from the canonical-sphinx theme[82] as a starting point and edit as needed.

See also

Sphinx uses the Jinja templating engine for its HTML templates; see the Jinja template syntax reference[83] for more details.

Use custom template for all pages

Sphinx looks for a template called `page.html` as the entry point and main page template for documentation pages. To customize your project's look and structure, check this file and determine which parts – such as the header, footer, or sidebars – need to be edited or overridden.

Here are some examples.

[82] https://github.com/canonical/canonical-sphinx/tree/main/canonical_sphinx/theme/templates
[83] https://jinja.palletsprojects.com/en/latest/templates/

Remove on-page TOC

To remove the on-page TOC in the right sidebar, make a copy of page.html[84] in the `docs/_templates` folder, and remove the applicable lines. This will apply to all pages.

Listing 26: page.html

```
{% block right_sidebar %}
<div class="toc-sticky toc-scroll">
  {% if not furo_hide_toc_orig %}
   <div class="toc-title-container">
     <span class="toc-title">
      {{ _("Contents") }}
     </span>
   </div>
   <div class="toc-tree-container">
     <div class="toc-tree">
       {{ toc }}
     </div>
   </div>
  {% endif %}
   {% if meta and ((meta.discourse and discourse_prefix) or meta.relatedlinks) %}
   <div class="relatedlinks-title-container">
     <span class="relatedlinks-title">
```

Add icon for GitHub link in header

To customize the default header by adding an icon for the GitHub link, first make a copy of header.html[85] in the `docs/_templates` folder.

Then modify the conditional statement related to the GitHub URL with your code.

Listing 27: header.html

```
[...]
          {% if github_url %}
          <li>
            <a href="{{ github_url }}" class="p-navigation__sub-link p-dropdown__link">
              <!-- GitHub icon (inline SVG) -->
              <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16" viewBox="0 0
16 16" fill="currentColor" style="vertical-align:middle; margin-right:4px;">
                <path d="M8 0C3.58 0 0 3.58 0 8c0 3.54 2.29 6.53 5.47 7.59.4.07.55-.17.55-
.38
                0-.19-.01-.82-.01-1.49-2.01.37-2.53-.49-2.69-.94-.09-.23-.48-.94-.82-1.13-
.28-.15-.68-.52
                -.01-.53.63-.01 1.08.58 1.23.82.72 1.21 1.87.87 2.33.66.07-.52.28-.87.51-
1.07-1.78-.2-3.64-.89
                -3.64-3.95 0-.87.31-1.59.82-2.15-.08-.2-.36-1.02.08-2.12 0 0 .67-.21 2.2.
82a7.65 7.65 0 012 0c1.53-1.03
                2.2-.82 2.2-.82.44 1.1.16 1.92.08 2.12.51.56.82 1.28.82 2.15 0 3.07-1.87
3.75-3.65 3.95.29.25.54.73.54 1.48
```

(continues on next page)

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

(continued from previous page)

```
                0 1.07-.01 1.93-.01 2.2 0 .21.15.46.55.38A8.013 8.013 0 0016 8c0-4.42-3.
58-8-8-8z"/>
              </svg>
              GitHub
            </a>
          </li>
          {% endif %}
[...]
```

Use custom template for specific pages

If you want to use a custom template for specific pages in your project, you can do so by using conditional logic in `page.html`.

First, create the base template with your modifications (e.g. `special-header.html`, `special-page.html`) and place it in the `docs/_templates` folder.

Next, make a copy of page.html[86].

Partial template changes

To make partial changes (e.g. custom header) to specific pages, modify only the relevant parts of `page.html` where you want the custom layout or behavior to apply. For example, wrap the body block in a conditional statement so the custom header (e.g. `special-header.html`) applies only to the "how-to/custom-templates" and "how-to/build" page.

Listing 28: _templates/page.html

```
{% block body -%}
   {% if pagename in ["how-to/build", "how-to/custom-templates"] %}
       {% include "special-header.html" %}
   {% else %}
       {% include "header.html" %}
   {% endif %}
   {{ super() }}
{%- endblock body %}
```

Whole template changes

To make changes to the whole template (e.g. a custom layout for a landing page or marketing page), modify the `extends` statement in `page.html` to specify the pages that will use different templates. For example, the `special-page.html` template applies only to the "how-to/customise" and "how-to/diagrams-as-code" page.

Listing 29: _templates/page.html

```
{% if pagename in ["how-to/customise", "how-to/diagrams-as-code"] %}
    {% extends "special-page.html" %}
```

(continues on next page)

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

(continued from previous page)

```
{% else %}
    {% extends "furo/page.html" %}
{% endif %}
```

Note:

The pages “how-to/customise” and “how-to/diagrams-as-code” will use `special-page.html` as the base template, but all other blocks (e.g. footer, body, etc) will follow the default `page.html`.

1.3. Reference

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

Also see the following information:

- Example product documentation[87] and Example product documentation repository[88]
- Sphinx documentation starter pack repository[89]

1.3.1. Contents

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[90] to check for accessibility issues in the documentation output.

It is configured to use the Web Content Accessibility Guidelines (WCAG) 2.2[91], requiring Level AA conformance[92].

Note:

This check is only available locally.

[87] https://canonical-example-product-documentation.readthedocs-hosted.com/
[88] https://github.com/canonical/example-product-documentation
[89] https://github.com/canonical/sphinx-docs-starter-pack
[90] https://pa11y.org/
[91] https://www.w3.org/TR/WCAG22/
[92] https://www.w3.org/WAI/WCAG2AA-Conformance

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
```

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[93] on GitHub.

Inclusive language check

The inclusive language check uses Vale[94] to check for violations of inclusive language guidelines.

Run the inclusive language check

Run the following command from within your documentation folder:

```
make woke
```

[93] https://github.com/pa11y/pa11y#command-line-configuration
[94] https://vale.sh/

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.

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[95] 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[96] that will be excluded from the check. To add custom exceptions for your project, add them to the `.custom_wordlist.txt` file.

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

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

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`:

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

Style guide linting

The starter pack includes a method to run the Vale[97] documentation linter configured with the Vale rules for the current style guide[98].

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*
```

[97] https://vale.sh/
[98] https://github.com/canonical/documentation-style-guide

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:

```
.. 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[99].

[99] 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[100]
- 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[101] 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`

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

```
jobs:
documentation-checks:
   uses: canonical/documentation-workflows/.github/workflows/documentation-checks.
yaml@main
   with:
      working-directory: "docs"
      fetch-depth: 0
      runs-on: "ubuntu-22.04"
```

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
```

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.

reStructuredText syntax reference

The documentation files use reStructuredText[102] (reST) syntax.

See the following sections for syntax help and conventions.

Note:

This guide assumes that you are using the Sphinx documentation starter pack[103]. Some of the mentioned syntax requires Sphinx extensions (which are enabled in the starter pack).

[103] https://github.com/canonical/sphinx-docs-starter-pack

For general style conventions, see the Canonical Documentation Style Guide[104].

[102] https://www.sphinx-doc.org/en/master/usage/restructuredtext/index.html
[104] 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[105].

Input	Output
Demonstrate a code block:: code: - example: true	Demonstrate a code block: code: - example: true
.. code:: # Demonstrate a code block code: - example: true	# Demonstrate a code block code: - example: true
.. code:: yaml # Demonstrate a code block code: - example: true	# Demonstrate a code block **code**: - **example**: true

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

Terminal output

A terminal view can be useful to show the output of a specific command, 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 include a terminal view, use the following directive:

Input	Output
`.. terminal::` `    input line 1` `    input line 2` `    output line 1` `    output line 2` `    output line 3`	`user@host:~$ input line 1 > input line 2` `output line 1` `output line 2` `output line 3`

By default, everything before the first blank line in the directive's content is rendered as input, while any content that follows is rendered as output. The terminal directive can only display one input command, but that command can span multiple lines, as in the previous example.

To render only the output of a command, include the `:output-only:` flag in the directive's options:

Input	Output
`.. terminal::` `    :output-only:` `    This is rendered as output.`	`This is rendered as output.`

To customize the prompt (`user@host:~$` by default), specify any of the following options:

- `:user:`
- `:host:`
- `:dir:`

Input	Output
`.. terminal::` `  :user: author` `  :host: canonical` `  :dir: ~/path` `  input` `  output`	`author@canonical:~/path$ input` `output`

The copy button for input commands is **opt-in**. You must include the `:copy:` flag in the directive's options for the button to be displayed.

To make the terminal scroll horizontally instead of wrapping long lines, include the `:scroll:` option.

For more details, refer to the sphinx-terminal README[106].

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[107]
`` :literalref:`ubuntu.com` ``	ubuntu.com[108]
`` :literalref:`xyzcommand <https://example.com>` ``	xyzcommand[109]

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

[106] https://github.com/canonical/sphinx-terminal/blob/main/README.md
[107] https://canonical.com/
[108] https://ubuntu.com
[109] 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[110]	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[111]).

[110] https://canonical.com/
[111] https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-rst_epilog

Listing 30:

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

Input	Output
`Canonical website`_	Canonical website[112]

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/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 `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.
```

[112] https://canonical.com/

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 80)	References a target that has a title.
`:ref:`Link text <a_random_target>``	*Link text* (page 80)	References a target and specifies a title.
`:ref:`starter-pack:home``	Sphinx example[113]	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.

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 65)
`:doc:`Link text <index>``	*Link text* (page 65)
`:doc:`starter-pack:how-to/index``	How-to guides[114]
`:doc:`Link text <starter-pack:how-to/index>``	Link text[115]

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

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[116] 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 `conf.py` file.

[116] 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[117] 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

[117] https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html#table-directives

Grid tables

See grid tables[118] for reference.

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

List tables

See list tables[119] for reference.

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

   * - Header 1
     - Header 2
   * - Cell 1

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

Data tables

The starter pack can render CSV data as tables. See *Create data tables* (page 54).

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

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`
- 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[120] for information about how to word the alt text.

[120] https://abilitynet.org.uk/resources/digital-accessibility/five-golden-rules-compliant-alt-text

Reuse

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

Substitution

To reuse sentences and entire paragraphs that have little markup or special formatting, define substitutions[121] 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[122]):

Listing 31:

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

Listing 32:

```
.. |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 33:

```
.. |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:

[121] https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html#substitutions
[122] 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[123]

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.

Input	Output
`.. include:: index.rst` `  :start-after: Also see the following information:` `  :end-before: Contents`	• Example product documentation[124] and Example product documentation repository[125] • Sphinx documentation starter pack repository[126]

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.

[123] https://example.com
[124] https://canonical-example-product-documentation.readthedocs-hosted.com/
[125] https://github.com/canonical/example-product-documentation
[126] https://github.com/canonical/sphinx-docs-starter-pack

- To make sure inclusions don't break, consider adding 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[127] extension provides.

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[128] 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

[127] https://sphinx-design.readthedocs.io/en/latest/
[128] https://sphinx-tabs.readthedocs.io/en/latest/

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.

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.

MyST syntax guide

The documentation files use a mixture of Markdown[129] and MyST[130] syntax.

See the following sections for syntax help and conventions.

Note:

This guide assumes that you are using the Sphinx documentation starter pack[131]. Some of the mentioned syntax requires Sphinx extensions (which are enabled in the starter pack).

[131] https://github.com/canonical/sphinx-docs-starter-pack

For general style conventions, see the Canonical Documentation Style Guide[132].

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).

[129] https://commonmark.org/
[130] https://myst-parser.readthedocs.io/
[132] https://docs.ubuntu.com/styleguide/en

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

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 ```

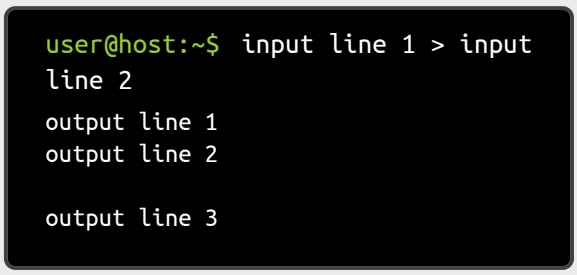
To include back ticks in a code block, increase the number of surrounding back ticks:

Input	Output
`` ``` ````	``` ```

Terminal output

A terminal view can be useful to show the output of a specific command, 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 line 1 input line 2 output line 1 output line 2 output line 3 ```	

By default, everything before the first blank line in the directive's content is rendered as input, while any content that follows is rendered as output. The terminal directive can only display one input command, but that command can span multiple lines, as in the previous example.

To render only the output of a command, include the `:output-only:` flag as a directive option:

Input	Output
``{terminal} :output-only: This is rendered as output. ```	

To customize the prompt (`user@host:~$` by default), specify any of the following options:

- `:user:`
- `:host:`
- `:dir:`

Input	Output
``{terminal} :user: author :host: canonical :dir: ~/path input output ```	

The copy button for input commands is **opt-in**. You must include the `:copy:` flag in the directive's options for the button to be displayed.

To make the terminal scroll horizontally instead of wrapping long lines, include the `:scroll:` option.

For more details, refer to the `sphinx-terminal` README[133].

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[134]
`https://canonical.com`	[135]
`` [`https://canonical.com`](https://canonical.com) ``	`https://canonical.com`

To display a URL as text and prevent it from being linked, add a `<span></span>`:

Input	Output
`https:/<span></span>/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 `[`.

[133] https://github.com/canonical/sphinx-terminal/blob/main/README.md
[134] https://canonical.com
[135] https://canonical.com

To add a link to a Discourse topic, configure the Discourse instance in the `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 97)	{ref}a_section_target_myst	References a target that has a title.
{ref}`link text <a_random_target_myst>`	*link text* (page 98)	{ref}link text <a_random_target_myst>	References a target and specifies a title.
{ref}`starter-pack:home`	Sphinx example[136]	{ref}starter-pack:home	You can also reference targets in other doc sets.
[`xyz`](a_random_target_myst)	*xyz* (page 98)	*xyz* (page 98) (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 65)	{doc}index	Preferred.
[](index)	*Reference* (page 65)		Do not use.
[Index page](index)	*Index page* (page 65)	*Index page* (page 65)	Preferred when overriding the link text.
{doc}`Index page <index>`	*Index page* (page 65)	{doc}Index page <index>	Alternative when overriding the link text.

Adhere to the following conventions:

[136] 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`[137] 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 `conf.py` file.

[137] 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[138] syntax is usually much easier. See the Sphinx documentation[139] 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

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[140] for reference.

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

* - Header 1
  - Header 2
* - Cell 1

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

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

Data tables

The starter pack can render CSV data as tables. See *Create data tables* (page 54).

Notes

Input	Output
	Note: A note.
`` ```{note} A note. ``` ````	
	Tip: A tip.
`` ```{tip} A tip. ``` ````	
	Important: Important information.
`` ```{important} Important information ``` ````	
	Caution: This might damage your hardware!
`` ```{caution} This might damage your hardware! ``` ````	

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[141] for information about how to word the alt text.

[141] https://abilitynet.org.uk/resources/digital-accessibility/five-golden-rules-compliant-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[142].

Substitutions can be defined in the following locations:

- Globally, in a file named `reuse/substitutions.yaml` that is loaded into the `myst_substitutions`[143] variable in `conf.py`:

Listing 34:

```
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 35:

```
# 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:

[142] https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html#substitutions
[143] https://myst-parser.readthedocs.io/en/v0.13.5/using/syntax-optional.html#substitutions-with-jinja2

Input	Output
`{{version_number}}`	0.1.0
`{{formatted_text}}`	*Multi-line* text that uses basic **markup**.
`{{site_link}}`	Website link[144]
`{{advanced_reuse_key}}`	This is a substitution that includes a code block: `code block`

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.

Input	Output
`` % 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." ``` ````	• Do not use consecutive headings without intervening text. • Be consistent with the characters you use for each level.

Adhere to the following convention:

- File inclusion does not work on GitHub. Therefore, always add a comment linking to the included file.

[144] https://example.com

- 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[145] extension provides.

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[146] extension, which is also enabled by default. This was previously recommended due to limitations in Sphinx Design that are now fixed.

[145] https://sphinx-design.readthedocs.io/en/latest/
[146] https://sphinx-tabs.readthedocs.io/en/latest/

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

Collapsible sections

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

Input	Output
<details> <summary>Details</summary> Content </details>	Content

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.

Input	Output
``{glossary} MyST example term Definition of the example term. ```	**MyST example term** Definition of the example term.
{term}`MyST example term`	*MyST example term*

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.

1.4. Explanation

Explore topics about the concepts and ideas in Canonical's Starter Pack.

The starter pack is built using standard Python tools, and is both deep and flexible.

1.4.1. Build

Canonical's Starter Pack uses Make as its build system. Make was chosen because it's well-tested and available on all platforms. The majority of the build configuration is defined in `docs/Makefile`.

Make is also the user interface for operating a project's docs. Authors and developers call all docs actions with `make <action>`.

The docs build depends on environment variables like `BUILDDIR` to locate critical files. They are conditional variables, so other systems can invoke the build at different locations without changing the docs `Makefile`.

Being primarily a Python project, all dependencies are stored in a virtual environment, `docs/.sphinx/venv`. The environment is ephemeral and subject to frequent change. Installing the starter pack initializes it, while cleaning and upgrading tears it down and rebuilds it.

Parent projects and the build

The starter pack is arranged as a standalone project. When it's used in a larger project, the docs are a subsystem among other components.

If the parent project uses a build system, Make or otherwise, the doc build exists in parallel with the parent build. When embedded, the virtual environment and build recipe files would be arranged along these lines:

```
Project
│
├── ...
├── docs
│   ├── ...
│   ├── .sphinx/venv
│   └── Makefile
├── src
│   └── ...
├── <parent build recipes>
└── <parent virtual environment>
```

With embedded docs, the parent build doesn't mesh with the docs build, which can cause difficulty:

- Contributors typically call the build commands from the root of the project. Some will find it inconvenient to run the docs commands by switching to the `docs` directory or manually calling the docs Makefile with `make -C docs <action>`.
- Storing multiple virtual environments bloats the host system. It's reasonable for project maintainers to prefer a shared build environment.

- The starter pack's upgrade process can make changes to many files in the `docs` directory. Upgrading is potentially much simpler if the parent project modifies only a minimum of files in the directory.
- With quality assurance and continuous integration, it's simpler if the project can use the same interface to run local and remote checks. More specifically, the parent build system and CI need a way to call the starter pack's `links`, `spelling`, and `vale` checks.

One possible resolution is for the parent build to manually recreate the docs build, tightly coupling the parent build to the existing docs configuration. But this poses another challenge, because the docs `Makefile` might change during a starter pack update, requiring a rewrite of the parent build recipes.

The solution to these complications is to create a bridge between the two builds, from the parent build to the docs `Makefile`. *Bridge project and docs builds* (page 38) is a guide for how to do this.