MARKUP

Formatting Guide: reStructuredText

Formatting Guide: reStructuredText

This is the formatting guide for reStructuredText as it may be used with the MARKUP theme.

Admonitions

Admonitions are notes and warnings. Use notes for a simple callout and warnings for things that will break if not followed correctly. Use the others sparingly, or at least in a consistent manner.

Attention

Use `.. attention::` as shown here:

```rst
.. attention:: The words for the attention itself.
```

to create an admonition like this:

Attention

The words for the attention itself.

Caution

Use `.. caution::` as shown here:

```rst
.. caution:: The words for the caution itself.
```

to create an admonition like this:

Caution

The words for the caution itself.

Custom Admonitions

This theme uses the default admonition to enable the use of custom titles. The default admonition is styled the same as a note.

For a custom admonition, use `.. admonition:: some string` as shown here:

```rst
.. admonition:: some string

   Contents of custom admonition!
```

Which will appear in the documentation like this:

some string

Contents of custom admonition!

Danger

Use `.. danger::` as shown here:

```rst
.. danger:: The words for the danger itself.
```

to create an admonition like this:

Danger

The words for the danger itself.

Error

Use `.. error::` as shown here:

```rst
.. error:: The words for the error itself.
```

to create an admonition like this:

Error

The words for the error itself.

Hint

Use `.. hint::` as shown here:

```rst
.. hint:: The words for the hint itself.
```

to create an admonition like this:

Hint

The words for the hint itself.

Important

Use `.. important::` as shown here:

```rst
.. important:: The words for the important itself.
```

to create an admonition like this:

Important

The words for the important itself.

Note

Use `.. note::` as shown here:

```rst
.. note:: The words for the note itself.
```

to create an admonition like this:

Note

The words for the note itself.

Tip

Use `.. tip::` as shown here:

```rst
.. tip:: The words for the tip itself.
```

to create an admonition like this:

Tip

The words for the tip itself.

Warning

Use `.. warning::` as shown here:

```rst
.. warning:: The words for the warning itself.
```

to create an admonition like this:

> **Warning**
>
> The words for the warning itself.

Card Walls

> **Warning**
>
> Card walls are not supported in PDF outputs.

Code Blocks

For code samples (Python, YAML, JSON, Jinja, config files, and so on) and for commands run via the command line that appear in the documentation we want to set them in code blocks using variations of the `.. code-block::` directive.

Code blocks are parsed using a tool called Pygments that checks the syntax in the named code block against the lexer in Pygments to help ensure that the structure of the code in the code block, even if it's pseudocode, is formatted correctly.

> **Warning**
>
> Pygments lexers check the code in a code block against a lexer. A lexer checks the structure and syntax of the code in the code block. If this check doesn't pass, the build will fail. For example, if code that contains YAML and Jinja templating is added to a `.. code-block:: yaml` code block, the build will fail because Jinja templating is not YAML. The same will happen if Ruby code is put in a Python code block. And so on. If we need to add a new code block for a particular language, talk to the docs team. In some rare cases, use the `none` code block to work around the problem, as it is much more forgiving.

Line Emphasis

Individual lines in a code block may be emphasized. The presentation is similar to a yellow highlight in a book. The following example shows how to highlight lines 3 and 5 in a code block:

```rst
.. code-block:: python
   :emphasize-lines: 3,5
```

```
   def function(foo):
     if (some_thing):
       return bar
     else:
       return 0
```

will display as:

```python
def function(foo):
  if (some_thing):
    return bar
  else:
    return 0
```

Command Shell

For command shell blocks, assign `console` as the name of the code block:

```rst
.. code-block:: console

   $ cr service stop
```

to create a code block like this:

```shell
$ cr service stop
```

Config File

For generic configuration file blocks, assign `text` as the name of the code block:

```rst
.. code-block:: text

   spark.setting.hours 1h
   spark.setting.option -Duser.timezone=UTC
   spark.setting.memory 20g
```

to create a code block like this:

```
spark.setting.hours 1h
spark.setting.option -Duser.timezone=UTC
spark.setting.memory 20g
```

Note

We're using `text` because there are not specific lexers available for all of the various configuration files. The `text` lexer allows us to style the code block similar to all of the others, but will not apply any highlighting to the formatting within the code block.

CSS

For CSS code blocks, assign `css` as the name of the code block:

```rst
```

```
.. code-block:: css

   ul.tab-selector {
     display: block;
     list-style-type: none;
     margin: 10 0 10px;
     padding: 0;
     line-height: normal;
     overflow: auto;
   }
```

to create a code block like this:

```css
ul.tab-selector {
  display: block;
  list-style-type: none;
  margin: 10 0 10px;
  padding: 0;
  line-height: normal;
  overflow: auto;
}
```

Data Table

Table blocks are used to show the inputs and outputs of processing data, such as with blocks reference documentation. For table code blocks, assign `sql` as the name of the code block:

```
.. code-block:: sql

   --------- ---------
    column1   column2
   --------- ---------
    value     value
    value     value
    value     value
   --------- ---------
```

to create a code block like this:

```
--------- ---------
 column1   column2
--------- ---------
 value     value
 value     value
 value     value
--------- ---------
```

HTML

For HTML code blocks, assign `html` as the name of the code block:

```
.. code-block:: html

   <div class="admonition warning">
     <p class="first admonition-title">Warning</p>
     <p class="last">The text for the warning built from raw HTML.</p>
   </div>
```

to create a code block like this:

```HTML
<div class="admonition warning">
  <p class="first admonition-title">Warning</p>
  <p class="last">The text for the warning built from raw HTML.</p>
</div>
```

JavaScript

For JavaScript code blocks, assign `javascript` as the name of the code block:

```RST
.. code-block:: javascript

   $('div.content-tabs').each(function() {
       var tab_sel = $('<ul />', { class: "tab-selector" });
       var i = 0;

       if ($(this).hasClass('right-col')){
           tab_sel.addClass('in-right-col');
       }

       $('.tab-content', this).each(function() {
           var sel_item = $('<li />', {
               class: $(this).attr('id'),
               text: $(this).find('.tab-title').text()
           });
           $(this).find('.tab-title').remove();
           if (i++) {
               $(this).hide();
           } else {
               sel_item.addClass('selected');
           }
           tab_sel.append(sel_item);
           $(this).addClass('contenttab');
       });

       $('.tab-content', this).eq(0).before(contenttab_sel);
       contenttab_sel = null;
       i = null;
   });
```

to create a code block like this:

```JAVASCRIPT
$('div.content-tabs').each(function() {
    var tab_sel = $('<ul />', { class: "tab-selector" });
    var i = 0;

    if ($(this).hasClass('right-col')){
        tab_sel.addClass('in-right-col');
    }

    $('.tab-content', this).each(function() {
        var sel_item = $('<li />', {
            class: $(this).attr('id'),
            text: $(this).find('.tab-title').text()
        });
        $(this).find('.tab-title').remove();
        if (i++) {
            $(this).hide();
        } else {
            sel_item.addClass('selected');
        }
```

```
        tab_sel.append(sel_item);
        $(this).addClass('contenttab');
    });

    $('.tab-content', this).eq(0).before(contenttab_sel);
    contenttab_sel = null;
    i = null;
});
```

JSON

For JSON code blocks, assign `json` as the name of the code block:

```RST
.. code-block:: json

   {
     "foo": [
       {
         "one": "12345",
         "two": "abcde",
         "three": "words"
       },
     ]
   }
```

to create a code block like this:

```JSON
{
  "foo": [
    {
      "one": "12345",
      "two": "abcde",
      "three": "words"
    },
  ],
}
```

JSON w/Jinja

For JSON code blocks that also embed Jinja templating, such as the nav-docs.html files that are used to build the documentation site's left navigation structures, the standard `.. code-block:: json` block will not work because the code block is not parsable as JSON. Instead, for code blocks that require a mix of JSON and Jinja templating, assign `django` as the name of the code block:

```RST
.. code-block:: django

   {% extends "!nav-docs.html" %}
   {% set some_jinja = "12345" %}
   {% set navItems = [
     {
       "title": "Start Here",
       "iconClass": "fas fa-arrow-alt-circle-right fa-fw",
       "subItems": [
         {
           "title": "Start Here",
           "hasSubItems": false,
           "url": "/some_file.html"
```

```
        },
        {
          "title": "FAQ",
          "hasSubItems": false,
          "url": "/faq.html"
        },
        {
          "title": "Additional Resources",
          "hasSubItems": false,
          "url": "/resources.html"
        },
      ]
    },
  ] -%}
```

to create a code block like this:

JSON w/JINJA

```
{% extends "!nav-docs.html" %}
{% set some_jinja = "12345" %}
{% set navItems = [
  {
    "title": "Start Here",
    "iconClass": "fas fa-arrow-alt-circle-right fa-fw",
    "subItems": [
      {
        "title": "Start Here",
        "hasSubItems": false,
        "url": "/some_file.html"
      },
      {
        "title": "FAQ",
        "hasSubItems": false,
        "url": "/faq.html"
      },
      {
        "title": "Additional Resources",
        "hasSubItems": false,
        "url": "/resources.html"
      },
    ]
  },
] -%}
```

Why django?

Using `django` seems like an odd way to specify a code block that contains both Jinja and JSON.

Django is a site templating language that is part of the Python world. The Sphinx themes are actually built using a combination of Django, Jinja, JSON, and other stuff. The left-side navigation, in particular, is a mix of JSON structure and Jinja variables.

`django` identifies the Pygments lexer that parses a code block that contains both Jinja and JSON.

Lua

For Lua code blocks, assign `lua` as the name of the code block:

RST

```
.. code-block:: lua

   A = class()
   function A:init(x)
     self.x = x
   end
   function A:test()
     print(self.x)
   end
```

to create a code block like this:

```lua
A = class()
function A:init(x)
  self.x = x
end
function A:test()
  print(self.x)
end
```

None

For text that needs to be formatted as if it were a code block, but isn't actually code, assign `none` as the name of the code block:

```
.. code-block:: none

   This is a none block. It's formatted as if it were code, but isn't actually code.

   Can include code-like things:

   function_foo()
     does: something
   end
```

to create a code block like this:

```
This is a none block. It's formatted as if it were code, but isn't actually code.

Can include code-like things:

function_foo()
  does: something
end
```

Python

For Python code blocks, assign `python` as the name of the code block:

```
.. code-block:: python

   def function(foo):
     if (some_thing):
       return bar
     else:
       return 0
```

to create a code block like this:

```python
```

```
def function(foo):
  if (some_thing):
    return bar
  else:
    return 0
```

REST API

For REST API code blocks that show how to use an endpoint, assign `rest` as the name of the code block:

```
.. code-block:: rest

   https://www.yoursite.com/endpoint/{some_endpoint}
```

to create a code block like this:

```
https://www.yoursite.com/endpoint/{some_endpoint}
```

Note

Use the JSON code block style for the JSON request/response part of the REST API.

reStructuredText

For reStructured Text code blocks, assign `rst` as the name of the code block:

```
.. code-block:: rst

   This is some *reStructured* **Text** formatting.

   .. code-block:: none

      that has some(code);
```

to create a code block like this:

```
This is some *reStructured* **Text** formatting.

.. code-block:: none

   that has some(code);
```

Ruby

For Ruby code blocks, assign `ruby` as the name of the code block:

```
.. code-block:: ruby

   items = [ 'one', 1, 'two', 2.0 ]
   for it in items
     print it, " "
   end
```

```
   print "\n"
```

to create a code block like this:

```ruby
items = [ 'one', 1, 'two', 2.0 ]
for it in items
  print it, " "
end

print "\n"
```

Scala

For Scala code blocks, assign `scala` as the name of the code block:

```rst
.. code-block:: scala

   object HelloWorld {
     def main(args: Array[String]) {
       println("Hello, world!")
     }
   }
```

to create a code block like this:

```scala
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Shell Script

For shell script blocks, assign `bash` as the name of the code block:

```rst
.. code-block:: bash

   # The product and version information.
   readonly MARKUP_PRODUCT="markup-app"
   readonly MARKUP_VERSION="1.23.45-6"
   readonly MARKUP_RELEASE_DATE="2019-04-01"
```

to create a code block like this:

```script
# The product and version information.
readonly MARKUP_PRODUCT="markup-app"
readonly MARKUP_VERSION="1.23.45-6"
readonly MARKUP_RELEASE_DATE="2019-04-01"
```

YAML

For YAML code blocks, assign `yaml` as the name of the code block:

```rst
.. code-block:: yaml
```

```
   config:
     - some_setting: 'value'
     - some_other_setting: 12345
```

to create a code block like this:

```yaml
config:
  - some_setting: 'value'
  - some_other_setting: 12345
```

YAML w/Jinja

For YAML code blocks that also embed Jinja templating, the standard `yaml` block will not work because the code block is not parsable as YAML. Instead, these code blocks must be able to parse a mix of YAML and Jinja templating. Assign `salt` as the name of the code block:

```rst
.. code-block:: salt

   {%- set some_jinja = "12345" %}

   config:
     - some_setting: 'value'
     - some_other_setting: {{ some_jinja }}
```

to create a code block like this:

```yaml+jinja
{%- set some_jinja = "12345" %}

config:
  - some_setting: 'value'
  - some_other_setting: {{ some_jinja }}
```

Why salt?

Using `salt` seems like an odd way to specify a code block that contains both Jinja and YAML.

SaltStack is a configuration management tool similar to Ansible, Chef, and Puppet. SaltStack uses a mix of Jinja and YAML to define system states that are to be configured and maintained. The `salt` lexer exists in Pygments originally because of how SaltStack defines system states, their use of Python and documentation built via Sphinx, and the need for a lexer that could parse a file with code samples that contain both Jinja and YAML.

`salt` identifies the Pygments lexer that parses a code block that contains both Jinja and YAML.

Content Tabs

Warning

Content tabs are not supported in PDF formats.

Expandos

Warning

Expandos are not supported in PDF formats.

Font Awesome

Warning

Font Awesome icons may not be visible in PDF outputs.

Header (Level 1)

An H1 header appears in the documentation like this:

```rst
H1 Headers
==========
An H1 header appears in the documentation like this.
```

Which will appear in the documentation like the actual header for this section.

Header (Level 2)

An H2 header appears in the documentation like this:

```rst
H2 Headers
----------
An H2 header appears in the documentation like this.
```

Which will appear in the documentation like the actual header for this section.

Header (Level 3)

An H3 header appears in the documentation like this:

```rst
H3 Headers
++++++++++
An H3 header appears in the documentation like this.
```

Which will appear in the documentation like the actual header for this section.

Header (Level 4)

An H4 header appears in the documentation like this:

```rst
```

```
H4 Headers
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
An H4 header appears in the documentation like this.
```

Which will appear in the documentation like the actual header for this section.

Header Markup Length

Sphinx requires the length of the header to be at least the same length as the content string that defines the header.

Short headers, short title markup. Makes sense!

That said, in large files, it's easier to scan the structure of the content when you can actually see where the headers are. That's why the header markup strings are recommended to be 50 characters long:

```
Short title
==================================================
```

This makes it easier to see the structure of the file when scrolling up and down a long topic page. This is pretty much the only reason to use consistent header markup length. Copy, paste, done.

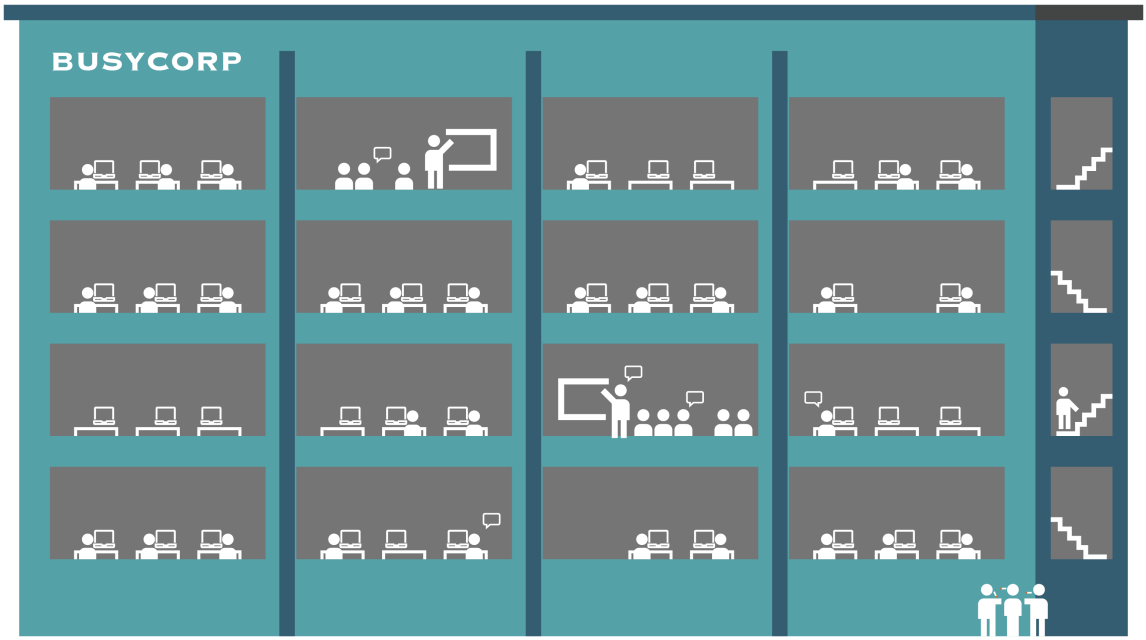
Images

Images may be embedded in the documentation using the `.. image::` directive. For example:

```rst
.. image:: ../../images/busycorp.png
   :width: 600 px
   :align: left
```

with the `:width:` and `:align:` attributes being aligned underneath `image` in the block.

This image will appear in the documentation like this:



Images should be SVG when only HTML output is desired. Printing to PDF from HTML pages requires PNG images.

Includes

Inclusions are a great way to single-source content. Write it in one place, publish it in many. There are two ways to handle inclusions, though both require using the `.. includes::` directive.

1. via File
2. via Snippet

via File

Inclusions may be done from standalone files. These standalone files are typically kept as a standalone file located in a dedicated directory within the docs repository, such as `/shared/some_file.rst`.

The `.. includes::` is used to declare the path to that file. At build time, the contents of the included file are built into the location specified by the `.. includes::` directive.

For example:

```rst
.. includes:: ../../includes/terms.rst
```

will pull in the contents of that file right into the location of the directive.

via Snippet

Inclusions may be done from within existing files as long as the target for that snippet is located in another file in the repository.

Warning

Snippets may not be used within the same file. The “target for that snippet” may not be the same file as the origin. This will cause a rendering issue in the output.

These types of inclusions require two steps:

1. Declare a start and an end for the snippet; this declaration must be unique across the entire documentation repository.

   Tip

   To help ensure unique snippet identifiers are built in the output, ensure that the snippet identifiers are directly assocaited with the name of the source directory and source file. These identifiers don’t have to be long (though they can be), but they must be unique within a doc set.

   For example, a file locatated at `internal_docs/source/tips.rst` should have snippet identifiers like `.. internal-docs-tips-some-identifier-start` or or `.. internal-docs-tips-some-identifier-end`.

2. Specify the `.. includes::` directive, along with the `:start-after:` and `:end-before:` attributes.

   The `:start-after:` and `:end-before:` attributes effectively use a unique code comment located in the file defined by the `.. includes::` directive to know the start and end of the snippet to be included.

For example, a snippet is defined in `docs/source/snippet.rst`:

```rst
This is the file named snippet.rst. It has a few paragraphs and a
reusable snippet.

Paragraph one.

.. docs-snippet-p2-start

Paragraph two.

.. docs-snippet-p2-end

Paragraph three.
```

This snippet can be included in other files like this:

```rst
Some content.

.. include:: ../../docs/source/snippet.rst
   :start-after: .. docs-snippet-p2-start
```

```
   :end-before: .. docs-snippet-p2-end

Some more content.
```

This should result in a file that looks similar to:

```rst
Some content.

Paragraph two.

Some more content.
```

Hint

Snippets may be sourced from large file that contain lists. For example, let's say the docs site has multiple docs collections (by application, by role, by internal vs. external, etc.) and you want each docs collection to have its own dedicated glossary to both enable consistency across doc sets for the same terms, but to also allow specific glossary terms for each doc set.

In this case, all glossary terms can be created and managed from a single file like `shared/terms.rst` in which the snippet start-end pairs are defined and the glossary terms are managed. Then each `glossary.rst` file across the docs set can use the `.. includes::` directive to pull in the terms it needs.

Inline Markup

Use any of these formatting options within paragraphs and lists:

- Bold
- Italics
- Code Strings

Bold

Use two asterisks (`**`) around the word to apply bold formatting: `**bold**`

Italics

Use a single asterisk (`*`) around the word to apply italics formatting: `*italics*`. For example: *this is italicized content*.

Code Strings

Use two backticks around the code string to apply code block formatting. For example:

```rst
``inline code string``
```

renders as shown above at the start of this list item.

Note

An inline code string should only be used within lists and paragraphs for function names, commands for command-line tools, and so on, and only in a way where the contents of that code string reads normally in a sentence. Use the code-block directive for anything else.

Links

There are three types of links:

- External
- Reference
- Topic

External

Ideally, external links should just be the full URL: http://www.yoursite.com, for example.

In some cases the external URL is too big and/or you want to embed the external link naturally within a sentence:

```rst
`some link text here <http://www.yoursite.com>`_
```

For example: some link text here

In some cases, Sphinx will throw warnings and errors if there are too many external links in the docs to the same places, so use a double underscore (__) at the end of the external link to stop those warnings/errors.

Linking topics together:

The `:ref:` links to an anchor on a page in the doc set, as long as the anchor is specified correctly immediately above the header for the section to which you want to link. For example:

```rst
:ref:`format-content-code-block-yaml`
```

will link to the header tagged with:

```rst
.. `_format-content-code-block-yaml`
```

There can be only one header with that name in the entire collection. The link itself resolves to the header string, which is why we want to limit use of this type of linking to only things that resolve exactly, such as function names, commands, settings groups, and so on.

Reference

There are two ways to link to internal headers across the doc set. First, a pre-requisite: the header to which the link is targeted must have an anchor. For example:

```rst
.. _anchor-name:

Internal Reference
------------------------------------------------
There are two ways to link to internal headers across the doc set.
First, a pre-requisite: the header that is the target of the link
must be tagged:
```

where the internal reference is the `.. _anchor-name:`. The string “anchor-name” must be unique across the entire doc set, so the required pattern for these is <file-name-header-name>, like this:

```rst
.. _format-content-code-block-yaml:
```

and then there are two ways to link to that anchor. The first will pull in the header name as the link:

```rst
:ref:`format-content-code-block-yaml`
```

and the second will use the string you put there and will not pull in the header name as the link:

```rst
This links to some information about using
:ref:`YAML code blocks <format-content-code-block-yaml>`
in your documentation.
```

These first example renders like this: YAML. The second example is preferred and looks like the next sentence. This links to some information about using YAML code blocks in your documentation.

Topic

There are two ways to link to internal topics across the doc set. The first will pull in the topic name as the link:

```rst
:doc:`blocks`
```

and the second will use the string you put there and will not pull in the header name as the link:

```rst
This links to some information about using :doc:`blocks </blocks>`
to build a pipeline.
```

Lists

Three types of lists are available:

- Definition List

- Ordered List
- Unordered List

Definition List

A definition list is a specially formatted list that uses whitespace to indent the descriptive text underneath a word or a short phrase. This type of list is used to describe settings—such as command line parameters, API arguments, glossary terms, and so on. For example:

```rst
**list_item_one**
   The description must be indented three spaces.

**list_item_two**
   The description must be indented three spaces.
```

Which will appear in the documentation like this:

list_item_one
: The description must be indented three spaces.

list_item_two
: The description must be indented three spaces.

Note

A definition list may contain a definition list. For example, some configuration settings (already in a definition list) have specific additional settings that must also be in a definition lists. These must be indented and must use the correct amount of white space.

Warning

A definition list title may not contain inline markup.

Ordered List

An ordered list has each list item preceded by an `#.` followed by a space. For example:

```rst
#. one
#. two
#. three
```

Which will appear in the documentation like this:

1. one
2. two
3. three

Unordered List

An unordered list has each list item preceded by an `*` followed by a space. For example:

```rst
* one
* two
* three
```

Which will appear in the documentation like this:

- one
- two
- three

Tables

Tables are always fun! This theme supports the following table types:

- CSV tables
- Grid tables
- List tables
- Simple tables

You can see from the examples below that there are slight differences between how you can set up the tables to get various table structures. Some table types are more fun than others.

There is no right or wrong way to build a table, as long as the build doesn't fail and your readers can see all of the information in the table. Therefore, use the table that's easiest for you. That said, for tables that have a lot of data in them, consider importing that data to a CSV table or use a list table. For tables that contain small amounts of data or that require odd layouts that span columns or rows, grid and simple tables are probably easier.

CSV Table

Tables may be built from a CSV file as long as the CSV file is available to Sphinx at build time.

Tables built from a CSV file use the `.. csv-table::` directive. For example:

```rst
.. csv-table::
   :file: ../../misc/test.csv
   :widths: 30, 70
   :header-rows: 1
```

with the `:widths:` and `:header-rows:` attributes being aligned underneath `csv-table` in the block. The `:file:` must be the path to a CSV file that is available to Sphinx at build time.

A CSV file is similar to:

```rst
Header1,Header2
12345,67890
abcdefghijklmnopqrstuvwxyz,abcdefghijklmnopqrstuvwxyz
```

where the first line in the CSV file is the header row.

This type of table will appear in the documentation like this:

Header1	Header2
12345	67890
abcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz

Grid Table

Grid tables are built by physically spacing out the table in the text file, similar to how it will appear on the page. These are easy when they are small.

```
+------------+------------+-----------+
| Header 1   | Header 2   | Header 3  |
+============+============+===========+
| body row 1 | column 2   | column 3  |
+------------+------------+-----------+
| body row 2 | Cells may span columns.|
+------------+------------+-----------+
| body row 3 | Cells may  | - Cells   |
+------------+ span rows. | - contain |
| body row 4 |            | - blocks. |
+------------+------------+-----------+
```

builds as:

Header 1	Header 2	Header 3
body row 1	column 2	column 3
body row 2	Cells may span columns.	
body row 3 body row 4	Cells may span rows.	• Cells • contain • blocks.

List Table

A list-table is built using the `.. list-table::` directive.

```rst
.. list-table::
   :widths: 200 400
   :header-rows: 1

   * - columnName
     - columnName
   * - **item1**
     - description
   * - **item2**
     - description
```

with the `:widths:` and `:header-rows:` attributes being aligned underneath `list-table` in the block. The number of rows (identified by the dashes (`-`) must equal the number of integers specified by `:widths:`. The integers specified by `:widths:` also specifies the column width, from left to right.

columnName	columnName
item1	description
item2	description

Simple Table

Simple tables are … simple. The markup is focused mostly on the vertical layout. Like grid tables, they are easy when they are small.

```
=====  =====  ======
   Inputs     Output
------------  ------
  A      B    A or B
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
```

builds as:

Inputs		Output
A	**B**	**A or B**
False	False	False
True	False	True
False	True	True
True	True	True

Toctree

A Sphinx project must declare all of the topics that are part of it, which means at least one `toctree` list must be declared.

Note

Because the Market theme doesn't build its left navigation automatically from the header structures in topics and because there's no Previous/Next linking, there's no reason to put a `toctree` on more than one page. Instead, just put the toctree on the root page for the

project (default: `index`) and add to that toctree an alphabetical list of the topics in the collection.

A toctree is similar to:

```
.. Hide the TOC from this file.

.. toctree::
   :hidden:

   test
```

View the source for this file to see the toctree that is used for this document collection.

Tokens

Tokens are defined in the file `names.txt` located at `/tokens`. Each token is defined similar to:

```rst
.. |company_name| replace:: YourCompanyName
```

When used in a sentence, use the `|company_name|` token to replace that with the string that follows `replace::`. For example: MARKUP (is the replacement string for `|theme|`).

Warning

Tokens may not be used in the left-side navigation template (`nav-docs.html`).

The following example tokens exist at `/tokens/names.txt`:

- `|company_name|` => YourCompanyName
- `|theme|` => MARKUP
- `|md|` => Markdown
- `|rst|` => reStructuredText

Use tokens in headers or topic titles carefully. Sphinx will build them correctly in the topic, but anchor references from the left-side navigation will not work unless the anchor reference specifies the token. For example, a token named `|abc|` used for a title must be specified in the left navigation as `"url": "/path/to/file.html#abc"`.

Note

Tokens can really slow the build down if there are too many of them. Sphinx will check each file for tokens, and then check the tokens file to look for matches. Every time this happens, a kitten dies. As such, you should keep the tokens file as small as possible.

Topic Titles

Tokens are defined in the file `names.txt` located at `/tokens`. Each token is defined similar to:

```rst
```

```
.. |company_name| replace:: YourCompanyName
```

When used in a sentence, use the `|company_name|` token to replace that with the string that follows `replace::`. For example: MARKUP (is the replacement string for `|theme|`).

Warning

Tokens may not be used in the left-side navigation template (`nav-docs.html`).

The following example tokens exist at `/tokens/names.txt`:

- `|company_name|` => YourCompanyName
- `|theme|` => MARKUP
- `|md|` => Markdown
- `|rst|` => reStructuredText

Use tokens in headers or topic titles carefully. Sphinx will build them correctly in the topic, but anchor references from the left-side navigation will not work unless the anchor reference specifies the token. For example, a token named `|abc|` used for a title must be specified in the left navigation as `"url": "/path/to/file.html#abc"`.

Note

Tokens can really slow the build down if there are too many of them. Sphinx will check each file for tokens, and then check the tokens file to look for matches. Every time this happens, a kitten dies. As such, you should keep the tokens file as small as possible.

Additional Resources

The following resources may be useful:

- Google Developer Documentation Style Guide