|  |
| --- |
| title: How to do things with reStructuredText and Git author: Peter S. Conrad language: en-US … |
| to do things with ReStructuredText =================================== |
| tructuredText is not just a markup language. It is … |
| h shared storage or source control, ReStructuredText can be a powerful laboration framework, suitable for both technical and non-technical tributors and supported by tools that meet every preference. |
| s guide presents some basic recipes for working with some useful tructuredText tools. Each recipe lists what you’ll need. Read Getting rted <getting-started/getting-started/>\_\_ to learn how to get set |
| you want to know more about the available tools first, you can browse se sections: |
| Source control with Git <../tools/tools-git/>\_\_. |
| sider installing more than one ReStructuredText editor to see which features like. |
| ut the recipes |

Each recipe explains the goals, “ingredients,” and procedures for an interesting way to use ReStructuredText. Just as a lasagna recipe doesn’t contain instructions for preheating (or installing) the oven, the recipes in this guide don’t contain software installation instructions. Git commands are not repeated from recipe to recipe, but documented more centrally to keep the recipes short.

The idea is to help you get started and work around some of the less obvious “gotchas”—not to provide complete documentation. You might find it helpful to open each tool’s website as you try the recipes, in case you get stuck.

Some recipes go together. You might use Centralized Git workflow <../recipes/recipes-centralized-workflow/>\_\_ to manage content you are editing in a ??? Git wiki <../recipes/recipes-git-wiki/>\_\_ with the goal of publishing a website with Hugo <../recipes/recipes-hugo/>**. Or you might take content that you started by taking notes <../recipes/recipes-notes/>** and turn it into a slide presentation <../recipes/recipes-slides-remark/>\_\_.

# Getting started

You can start working with ReStructuredText just by opening your favorite text editor and starting to type—but you probably want to do more than that. For example, you might want to share information on a website or wiki, take notes, or publish an eBook. To get the most out of ReStructuredText, you’ll want to be familiar with some of these tools and topics.

## Ingredients

These are the tools and resources you will use for the recipes in this guide. Each recipe lists the tools you’ll need.

.. raw:: html

.. raw:: html

.. raw:: html

ReStructuredText editor

.. raw:: html

.. raw:: html

A ReStructuredText editor is a specialized text editor that works with ReStructuredText. Different editors have different features—you’ll want to play with more than one to find out which you like.

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Pandoc

.. raw:: html

.. raw:: html

Pandoc enables your ReStructuredText editor to import and export in several formats and also has a few tricks of its own. I recommend Pandoc for anyone who works with ReStructuredText.

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Shared storage

.. raw:: html

.. raw:: html

If you are collaborating with others, you’ll need a shared place to store ReStructuredText files. You might also want web hosting or a blogging platform where you can publish your content when it’s complete.

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Git

.. raw:: html

.. raw:: html

Source control can be very important for collaborating without catastrophe. You’ll need an account with a Git host, a Git client, and Git installed on your computer.

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Publishing tools

.. raw:: html

.. raw:: html

Documentation management tools, static site generators, and even Pandoc can be useful to convert your content to a final format for general consumption.

.. raw:: html

.. raw:: html

.. raw:: html

You can install everything up front, or just install what you need as you go along. If you’re not sure, start with the following steps:

1. Install Pandoc <../../tools/tools-publishing/#pandoc>\_\_.
2. Choose and install a ReStructuredText editor <../../tools/tools-editors/>\_\_.
3. Set up Git <../../tools/tools-git-setup/>\_\_.

!!! hint Some editors detect pandoc, so installing it first is a good idea.

Once you’ve got these tools installed, you can start trying the recipes.

## Things to know

You’ll find it easiest to follow the recipes in this guide if you are comfortable with the following topics:

**ReStructuredText Syntax**  
Although some editors provide a WYSIWYG experience, knowing how to write ReStructuredText <>\_\_ directly will help you work more quickly and easily with a wider variety of tools.

* See the ReStructuredText cheatsheet <>\_\_

**Working with a package manager**  
Installing some tools requires using one of the following common package managers:

* Linux: apt-get <https://help.ubuntu.com/community/AptGet/Howto> **or yum <http://yum.baseurl.org/>**
* macOS: Homebrew <https://brew.sh/>\_\_
* Windows: Chocolatey <https://chocolatey.org/>\_\_

A package manager can make it easy to install several packages at once. For example:

::

sudo apt-get install python3 ghostwriter pandoc mkdocs hugo gftp

**Git**  
Some of the recipes in this guide use Git <https://git-scm.com/>**. Git is very powerful, but doesn’t have to be hard to use. The recipes in this guide emphasize simplicity. If you already know Git, you can choose to do things differently. If you don’t know git, take a look at Git basics <../getting-started-git-basics>**.

**Other documentation tools**

https://gist.github.com/dupuy/1855764

Introduction

https://build-me-the-docs-please.readthedocs.io/en/latest/Using\_Sphinx/OnReStructuredText.html

https://docutils.sourceforge.io/rst.html

A reStructuredText document is simply a plain text file with some markup to specify the format or the semantics of the text.

There are two types of markup:

inline markup: for example, \*the surrounding asterisks would mark this text as italics\*, like this.  
explicit markup: is used for text that need special handling, such as footnotes, tables, or generic directives. Explicit markup blocks always start with .. followed by whitespace.

Paragraphs

The paragraph is the basic block in a reST document.

Paragraphs are simply chunks of text separated by one or more blank lines.

Indentation is significant in reST, so all lines of the same paragraph must be left-aligned to the same level of indentation.

This is a style convention.

Try to keep each line with a maximum of 78 characters. Remember that changing to next line does not create a paragraph, unless the chunks of text is separated by a blank line.

Try to keep each phrase in a different line. It improves readability and facilitates the translation process.

Remember that consecutive blank lines will be ignored in the HTML output. Quoted paragraphs

Quoted paragraphs are created by just indenting them more than the surrounding paragraphs:

Normal paragraph.

Indented paragraph.

This is a style convention.

Each indentation level is created with 3 whitespaces. Do not use tabs. Line breaks

Line blocks are a way of preserving line breaks (the equivalent of using Shift+Enter to break a line in Microsoft Word or LibreOffice Writer):

These lines are  
broken exactly like in  
the source file.

Sections

Section are created by underlining (and optionally overlining) the section title with a punctuation character:

# This is a heading

Any punctuation character can be used to define a section title. The underlining (and overlining) must be at least as long as the text itself. Sections must be properly nested.

This is a style convention.

Use the following punctuation characters in the section titles:

# for Parts  
\* for Chapters  
= for sections (?Heading 1?)  
- for subsections (?Heading 2?)  
^ for subsubsections (?Heading 3?)  
" for paragraphs (?Heading 4?)

Please note that, when converting to HTML format, sections are automatically converted to an appropriate heading tag (for example:

Heading text

).

When converting to ODT or DOCX, an appropriate Heading style is applied. Inline markup Bold, italics, monospace

The markup is quite simple:

use one asterisk for italics: \*text\* (the equivalent of using Ctrl+i in Microsoft Word or LibreOffice Writer),  
use two asterisks for strong emphasis (boldface): \*\*text\*\*  
use backquotes for text literals: ``text``

Be aware of some restrictions:

The markup may not be nested. For example, this markup is wrong: \*italics with \*\*bold\*\* inside\*  
The text content within the markup may not start or end with whitespace. For example, this markup is wrong: \* text\*  
The markup must be separated from surrounding text by non-word characters (whitespace or punctuation). Use a backslash-escaped-space to work around that. For example: thisis\ \*one\*\ word is rendered like thisisoneword.  
If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, they have to be escaped with a backslash.

Subscript and superscript

Subscript is marked with :sub:subscript text. Superscript is marked with :sup:superscript text.

This is a tip.

Whitespace or punctuation is required around interpreted text, but often not desired with subscripts & superscripts. Backslash-escaped whitespace can be used; the whitespace will be removed from the processed document:

The chemical formula for molecular oxygen is O :sub:2.

To improve the readability of the text, the use backslash-escapes is discouraged. If possible, use Substitutions instead:

The chemical formula for pure water is |H2O|.

.. |H2O| replace:: H :sub:2 O

Keep all substitutions together (e.g. at the end of the file). Lists Bulleted lists

List markup is natural: just place an asterisk at the start of a paragraph and indent properly:

* This is a bulleted list.
* It has two items, the second item uses two lines.

Nested lists are possible, but be aware that they must be separated from the parent list items by blank lines:

* this is
* a list
  + with a nested list
  + and some sub-items
* and here the parent list continues

Numbered lists

The same goes for numbered lists; they can also be auto-numbered using a # sign:

1. This is a numbered list.
2. It has two items too.
3. This is a numbered list.
4. It has two items too.

Definition lists

Definition lists are created as follows:

term (up to a line of text) Definition of the term, which must be indented

and can even consist of multiple paragraphs

next term Description.

The Sphinx documentation generator provides a more flexible alternative to definition lists (see Glossaries). Glossaries

The Sphinx ..glossary:: directive contains a reST definition-list-like markup with terms and definitions.

See the following example:

.. glossary::

environment A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

source directory The directory which, including its subdirectories, contains all source files for one Sphinx project.

The definitions will then be used in cross-references with the :term: role. For example:

The :term:source directory for this project is …

In contrast to regular definition lists, a glossary supports multiple terms per entry and inline markup is allowed in terms. You can link to all of the terms. For example:

.. glossary::

term 1 term 2 Definition of both terms.

When the glossary is sorted, the first term determines the sort order.

To automatically sort a glossary, include the following flag:

.. glossary:: :sorted:

Field lists

Field lists are two-column table-like structures resembling database records (label & data pairs). For example:

:Date: 2001-08-16 :Version: 1 :Authors: - Me - Myself - I :Indentation: Since the field marker may be quite long, the second and subsequent lines of the field body do not have to line up with the first line, but they must be indented relative to the field name marker, and they must line up with each other. :Parameter i: integer

Tables

The reStructuredText markup supports two basic types of tables. For grid tables, you have to ?paint? the cell grid yourself. They look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| Header row, column 1 (header rows optional) | Header 2 | Header 3 | Header 4 |
| body row 1, column 1 | column 2 | column 3 | column 4 |
| body row 2 | … | … |  |

Simple tables are easier to write, but limited: they must contain more than one row, and the first column cannot contain multiple lines. They look like this:

===== ===== ======= A B A and B ===== ===== ======= False False False True False False False True False True True True ===== ===== =======

This is a tip.

These are the basic types of tables, which are rather clumsy. Also available (and easier to use) are special tables, namely list-tables and CSV-tables.

An exceltable\_ extension can also be used with Sphinx, which allows the inclusion of XLS spreadsheets, or part of them, into a reST document. Hyperlinks External links

Use link text <http://example.com/>\_ for inline web links. If the link text should be the web address, you don?t need special markup at all, the parser finds links and mail addresses in ordinary text (with no markup).

You can also separate the link and the target definition, like this:

This is a paragraph that contains a link\_.

.. \_a link: http://example.com/

This is a tip.

The use of inline web links is discouraged, to improve the readability of the reST text.

Simple links (e.g. to institutional sites, software sites, and so on) should be kept together at the end of the text file (this is merely a way to simplify the editing procedure, and the update and verification of the links). Internal links

To support cross-referencing to arbitrary locations in any document, the standard reST labels are used. For this to work, the label names must be unique throughout the entire documentation. There are two ways in which you can refer to labels:

If you place a label directly before a section title, you can reference to it with :ref:`label-name`. Example:  
  
.. \_my-label-ref:  
  
Section to cross-reference  
--------------------------  
  
This is the text of the section.  
  
In the end of this phrase is a reference to the section title, see :ref:`my-label-ref`.  
  
The :ref: role would then generate a link to the section, with the link title being ?Section to cross-reference?. This works just as well when section and reference are in different source files.  
  
Automatic labels also work with figures:  
  
.. \_my-figure-ref:  
  
.. figure:: my-image.png  
  
 My figure caption  
  
A reference like :ref:`my-figure-ref` would insert a reference to the figure with link text ?My figure caption?.  
  
The same works for tables that are given an explicit caption using the table directive.  
  
Labels that aren?t placed before a section title can still be referenced to, but you must provide the text for the link, using this syntax: :ref:`Link text <label-name>`.

Using :ref: is advised over standard reStructuredText links to sections (like Section title\_) because it works across files, when section headings are changed, and for all builders that support cross-references. Source Code

Literal code blocks are introduced by ending a paragraph with the special marker ::. The literal block must be indented (and, like all paragraphs, separated from the surrounding ones by blank lines):

This is a normal text paragraph. The next paragraph is a code sample::

It is not processed in any way, except that the indentation is removed.

It can span multiple lines.

This is a normal text paragraph again.

The handling of the :: marker is smart:

If it occurs as a paragraph of its own, that paragraph is completely left out of the document.  
If it is preceded by whitespace, the marker is removed.  
If it is preceded by non-whitespace, the marker is replaced by a single colon.

That way, the second sentence in the above example?s first paragraph would be rendered as ?The next paragraph is a code sample:?. Explicit Markup

Explicit markup is used in reStructuredText for most constructs that need special handling, such as footnotes, specially-highlighted paragraphs, comments, and generic directives.

An explicit markup block begins with a line starting with .. followed by whitespace and is terminated by the next paragraph at the same level of indentation. (There needs to be a blank line between explicit markup and normal paragraphs. This may all sound a bit complicated, but it is intuitive enough when you write it.) Directives

A directive is a generic block of explicit markup.

The directive content follows after a blank line and is indented relative to the directive start.

Basically, a directive consists of a name, arguments, options and content.

Look at this example:

.. contents:: This is my Table of Contents :depth: 2

The directive starts with .. followed by one whitespace. The name of the directive is contents (it creates a table of contents). This directive takes one argument: the table of contents? title (?This is my Table of Contents?). The option depth specifies the number of section levels that are collected in the table of contents.

Options are given in the lines immediately following the arguments and are indicated by the colons. Options must be indented to the same level as the directive content.

Docutils supports the following directives:

Admonitions: attention, caution, danger, error, hint, important, note, tip, warning and the generic admonition. (Most themes style only note and warning specially.)  
Images:  
 image - see the images section;  
 figure - an image with caption and optional legend.  
Additional body elements:  
 contents <table-of-contents> - a local table of contents for the sections in the current file only;  
 rubric - a heading without relation to the document?s sections that won?t be included in any table of contents;  
 topic and sidebar - special highlighted body elements;  
 epigraph - a block quote with optional attribution line;  
 container - a container with a custom class, useful to generate an outer ``<div>`` in HTML output.  
Special tables:  
 table - a table with title;  
 csv-table - a table generated from comma-separated values;  
 list-table - a table generated from a list of lists.  
Special directives:  
 include - include reStructuredText from another file;  
 raw - include raw target-format markup, such as LaTeX;  
 class - assign a class attribute to the next element.

Table of contents

To include a table of contents within a given document, use the directive contents. The following example creates a local table of contents with a maximum of two levels (below the level where it is located):

Part II #######

Chapter 1 \*\*\*\*\*\*\*\*\*

.. contents:: :depth: 2 :local:

# Heading 1

The toctree directive creates a table of contents that collets information from several files. The following example creates a table of contents from the sections of various documents (up to a depth of 3 levels). The :glob: option allows all documents in the ?chapter2? folder to be included (sorted according to their name):

.. toctree:: :glob: :maxdepth: 3

preamble chapter1/part1 chapter1/conclusion chapter2/\* references

Note

When building HTML pages from the default template, a

is created that holds a ?table of contents? with links to the document sections. The number of levels in the sidebar can be controlled. For example, placing :tocdepth: 3 in the beggining of the document restricts the number of levels to 3. Images

reST supports an image directive, used like so:

.. image:: gnu.png (options)

The file name given (here gnu.png) must either be relative to the source file, or absolute (which means that they are relative to the top source directory).

For example, the file sketch/spam.rst could refer to the image images/spam.png as ../images/spam.png or as /images/spam.png.

The image size options (width and height) should be specified in points (pt), as that will best support output to different formats (HTML, LaTeX). Figures

A figure consists of image data (including image options), an optional caption (a single paragraph), and an optional legend (arbitrary body elements):

.. figure:: picture.png :scale: 50 % :alt: map to buried treasure

This is the caption of the figure (a simple paragraph).

The legend consists of all elements after the caption. In this case, the legend consists of this paragraph and the following table:

+———————–+———————–+ | Symbol | Meaning | +=======================+=======================+ | .. image:: tent.png | Campground | +———————–+———————–+ | .. image:: waves.png | Lake | +———————–+———————–+ | .. image:: peak.png | Mountain | +———————–+———————–+

There must be blank lines before the caption paragraph and before the legend. To specify a legend without a caption, use an empty comment (..) in place of the caption. Special tables

The table directive associates a title with the following table:

.. table:: User list

========== ========= First name Last name ========== ========= John Doe Jane Dove ========== =========

A list-table is created from a uniform two-level bullet list:

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

* + First name
  + Last name
  + John
  + Doe
  + Jane
  + Dove

A csv-table is created from comma-separated values (either in the document or in an external file):

.. csv-table:: User list :header:“First name”,“Last name”

“John”,“Doe” “Jane”,“Dove”

Another example of csv-table, using and external file:

.. csv-table:: Table 1 - Legend of the table goes here… :header-rows: 1 :stub-columns: 1 :file: ../tables/table1.csv

An exceltable can also be used:

.. exceltable:: Table 1 - Legend of the table goes here… :file: ../tables/tables.xls :sheet: table1 :selection: A1:C20 :header: 1

Using Excel tables requires an additional module sphinxcontrib.exceltable that is an extension for Sphinx, that adds support for including spreadsheets, or part of them, into Sphinx document. It can be installed using pip:

pip install sphinxcontrib-exceltable

Then the project conf.py file needs to be updated:

# Add sphinxcontrib.exceltable into extension list

extensions = [‘sphinxcontrib.exceltable’]

Another alternative is xmltable (https://pythonhosted.org/rusty/xmltable.html). Footnotes

For footnotes, use [#name]\_ to mark the footnote location, and add the footnote body at the bottom of the document after a ?Footnotes? rubric heading, like so:

Lorem ipsum [#first-footnote-name]\_ dolor sit amet [#second-footnote-name]\_

.. rubric:: Footnotes

.. [#first-footnote-name] Text of the first footnote. .. [#fsecond-footnote-name] Text of the second footnote.

You can also explicitly number the footnotes ([1]*) or use auto-numbered footnotes without names ([#]*).

This is a tip.

To facilitate editing, auto-numbered footnotes should not be used. Instead, use short descriptive names (that simplify cross-referencing). Citations

Standard reST citations are supported:

Lorem ipsum [Ref]\_ dolor sit amet.

.. [Ref] Book or article reference, URL or whatever.

Citation usage is similar to footnote usage, but with a label that is not numeric or begins with #.

When the documentation is built using the Sphinx document generator, the citations are ?global?, meaning that every citation can be referenced from any .rst files. In this case, a separate file may be created (e.g. a references.rst file).

This is a tip.

See Managing bibliographic citations in Sphinx for further information. Substitutions

reST supports ?substitutions?, which are pieces of text and/or markup referred to in the text by |name|. They are defined like footnotes with explicit markup blocks, like this:

.. |name| replace:: replacement *text*

or this:

.. |caution| image:: warning.png :alt: Warning!

If you want to use some substitutions for all documents, put them into a separate file (e.g. substitutions.txt) and include it into all documents you want to use them in, using the include directive.

Be sure to use a file name extension which different from that of other source files, to avoid Sphinx finding it as a standalone document. For example, use the .rst file extension for the source files, and the .txt file extension for the files which are to be included.

This is a tip.

This is useful in technical documentation such as User?s Manuals, where a substitution file can be built for each localised version of the interface elements (menus, messages, etc), guaranteeing the consistency of the document translation with the software?s human user interface.

Warning.

Substitutions do NOT work inside directives (or inside the options of a directive).

Do not try to google for a solution (…been there). It is a design limitation: RST markup can not be nested. Period. Comments

Every explicit markup block which isn?t a valid markup construct is regarded as a comment. For example:

.. This is a comment.

You can indent text after a comment start to form multiline comments:

.. This whole indented block is a comment.

Still in the comment.

This is a style convention.

Comments can also be used as placeholders to mark places within the document. For example:

the .. links-placeholder can mark the place where hyperlinks are kept together at the end of the document;  
the .. metadata-placeholder can mark the place where document metadata (author, date, etc) is kept together at the beginning of the document.

# ReStructuredText editors

The stupid underline has to be as long as the title

http://rst.ninjs.org/#

Editors

reStructuredText documents are text files, and can be edited with any text editor or word processor (provided they are always saved as text files). JEdit

jEdit is a FOSS text editor, written in Java (so it runs in Windows, Mac OS X, Linux, etc.). ReST is among the 211 languages supported natively by jEdit. This document opened in jEdit. Notepad++

Notepad++ is a FOSS text editor for MS Windows OS only.

reStructuredText is not among the languages natively recognised by Notepad++, but it can be added using a User Defined Language File`\_ (see install instructions below the list of available language files).

Follow the link to download the ReST syntax file`\_.

Notepad++ is simpler and more user friendly than jEdit. This document opened in Notepad++. ReText

ReText is a simple editor that reads your text with MarkDown or HTML markup and saves it as plain text, HTML or PDF. It is written in Python using Qt libraries. Visual Studio Code

Visual Studio Code is a FOSS text editor, written in TypeScript (so it runs in Windows, Mac OS X, Linux, etc.). ReST is not among the languages natively supported by Visual Studio Code, but it can be added using an extension from LeXtudio. This document opened in Visual Studio Code.

# Storage

The simplest place to store ReStructuredText is on your own computer, of course. You can merrily save files to a folder deep in My Documents and go on your way, perhaps emailing someone a Pandoc-generated PDF from time to time. But if you want to enjoy the true power of collaborating and publishing with ReStructuredText, you’ll want to share the files you’re working on or publish them in another form.

## A shared server

If you are collaborating on documents or contributing to documentation at work, you might need access to a shared server such as SharePoint or Samba, WebDAV, or online tools such as a Google Drive account. You can store your source files there or upload HTML, Word, PDF, and other documents you create from your ReStructuredText.

## Cloud storage

Cloud storage services like Box <https://www.box.com>**, DropBox <https://www.dropbox.com/>**, and OneDrive <https://onedrive.live.com>\_\_ are a great place to keep documents you want available from everywhere. You can save ReStructuredText files from one computer and have them available on other devices, even sharing them with colleagues.

## Web hosting

If you’re creating online documentation or a website, you’ll need a place to host it. Web hosting is far too big a subject to cover here, but you might need to investigate whether you need any of the following:

* A blog service such as Wordpress <https://wordpress.com/>**, Blogger <https://www.blogger.com/>**, or Medium <https://medium.com/>\_\_ where you can publish writing directly
* A web host, to present your HTML files as web pages
* FTP software to upload HTML to a web host
* Another integrated online publishing service or platform

# Source control with Git

Any time you collaborate with others, source control is important. Although you could just share a folder in the cloud, eventually there will come a time when someone overwrites or deletes something important. That’s where source control comes in. And by “source control,” I mean Git.

With Git, you synchronize files on your computer with files on a remote repository—usually an online Git host.

## Why Git?

Back in the day, there were all kinds of source control systems. The ones I remember were all pretty easy to use, but were all centralized—meaning that only one person could edit any given file at any given time. Worse, it meant that there was one central source of truth; if that were to be corrupted or lost, everything would be gone. Git solves these problems and doesn’t have to be hard to use <https://levelup.gitconnected.com/git-doesnt-have-to-be-hard-e1e115be6668>\_\_.

## Get Git

You’ll need a place to host a Git repository. You might start by signing up with a Git host. Here are a few examples:

* Bitbucket <https://bitbucket.org>\_\_
* GitHub <https://github.com/>\_\_
* GitLab <https://about.gitlab.com/>\_\_

You can use Git on the command line, but it’s even easier if you use a Git client. For example:

* GitHub Desktop <https://desktop.github.com/>\_\_
* Sourcetree <https://www.sourcetreeapp.com/>\_\_

!!! hint While any Git client should work with any Git host, it’s not a bad idea to use the client and host that go together. For example:

::

* Bitbucket and Sourcetree
* GitHub and GitHub Desktop

This guide includes instructions for those two combinations.

## Git concepts

Git is different from older source control systems. A lot of things that used to be intuitive—like the idea that we’re working on *files* and we need to *lock* them so that someone else can’t *check them out* from the *central repository*—have no currency in the Git world:

* There’s no *central repository.* A central repository would represent both a bottleneck and a single point of failure, so why do it?
* No one *checks out* files because there’s no central repository.
* We don’t need to *lock* files because we don’t have to worry about them being *checked out*.

In fact, Git doesn’t care about *files* at all. Git only cares about *changes.*

There’s a lot to know about Git. As a writer, you should be able to decide how much you want to learn. You might just want to learn *exactly* enough to do your writing and keep out of trouble. That’s the goal here. If you want to learn more, check out The Git Book <https://git-scm.com/book/en/v2>\_\_.

What things mean in Git ~~~~~~~~~~~~~~~~~~~~~~~

Here are a few basic Git terms you’ll see in some of the recipes.

**Commit** As you make changes to the files you’re working on, saving periodically, you also tell Git from time to time that you want your changes tracked. This is called *committing* the changes, and is a little like “saving changes to Git.” Git makes it easy by noticing the changes you’ve made so you can review them and make sure you’re not accidentally tracking something irrelevant. When you commit, you type a little note describing the changes so that people know what you did.

A group of changes you’ve committed is also called a *commit.*

**Stage**  
Before you commit changes, you tell Git which changes to track. This is called *staging.* Since changes go with files, sometimes people think of it as staging the files themselves—but it’s really the changes that Git wants to know about. If you delete a file, that’s a change too.

**Repository**  
When you have committed, your changes are stored in the local *repository,* or *repo* for short. It’s just like that old central repository, but it’s on your computer.

**Push**  
If you want other people to be able to work on your files, then you need to put them in an online repository (such as Bitbucket, GitHub, or GitLab). This is not *checking in* since the changes are already committed to your local repository. This is called a *push* to a *remote repository*.

**Clone**  
Once something is in an online repository, an authorized person can *clone* their own copy of the whole repository and work on the files locally.

**Pull**  
As you work on a repository you’ve cloned, you *pull* the latest changes from the remote repo to stay up to date.

**Branch**  
Git lets people work in separate work streams called *branches* so that they don’t interfere with each other’s work. A branch is just a series of commits (and a commit is a group of changes). You’re always working in a branch, even if there’s only one branch. When you have several branches to work in, Git remembers the state of everything in each branch so that when you switch between them everything is just how you expect it.

Creating a new branch is called *branching,* of course. The Git command for creating (or switching to) a branch is called, confusingly, *checkout.*

There are different branching strategies. Some are complicated; others are simple. The recipes in this guide use two simple branching strategies, described in Git basics <../tools-git-basics/>\_\_.

**Merge**  
If there’s more than one branch, there always comes a time to *merge,* which means to add the changes from one branch into another.

Why branching is cool ~~~~~~~~~~~~~~~~~~~~~

Git keeps track of the whole history of all the changes on all the branches. Not only does that mean you can roll back to any point in time, it also means that when you switch branches all your files magically change to match whatever changes you’ve made in that branch.

For example, I created a branch called test-branch based on the master branch. Working in test-branch, I added a file called new-file.md which you can see in the directory:

::

$ ls getting-started img index.md new-file.md

When I switch to master, it’s not there:

::

$ git checkout master Switched to branch ‘master’ Your branch is up to date with ‘origin/master’. $ ls getting-started img index.md

When I switch back to test-branch, it’s there:

::

$ git checkout test-branch Switched to branch ‘test-branch’ $ ls getting-started img index.md new-file.md

Branching is cool.

## Next steps

* Set up Git <../tools-git-setup/>\_\_

# Git setup

If you want to be ready for all the recipes in this guide, follow these steps to install Git and a Git client, sign up for a host, and set up your first repository.

## Install Git

On the command line, check whether Git is installed on your computer by typing:

::

git –version

If Git is not installed, follow these steps.

1. Install Git on your computer:
   * Windows <https://git-scm.com/download/win>\_\_
   * macOS <https://git-scm.com/download/mac>\_\_
   * Linux <https://git-scm.com/download/linux>\_\_
2. Install a git client such as Sourcetree <https://www.sourcetreeapp.com/>\_\_ or GitHub Desktop <https://desktop.github.com/>\_\_.

!!! hint You can use any Git client with any Git host, but some clients work especially well with specific hosts. For example:

::

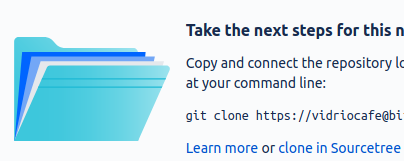
* Sourcetree and Bitbucket
* GitHub Desktop and GitHub

## Set up a repository

A *repository* is where you keep your work. You’ll need a *local repository* where you edit and save files on your computer, and a *remote repository* online that lets people collaborate on the same project. A straightforward way to create both is to set up a repository with an online host and then *clone* it (create a local copy). Your collaborators can also clone the repository to their own computers, so everyone can keep in sync by pushing and pulling changes.

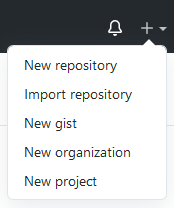
=== “Bitbucket and Sourcetree”

::

1. Sign up with [Bitbucket](https://bitbucket.org/) and log on.
2. Click the new repository button (the plus sign):  
   New repository buttoon
3. Type a repository name, make sure it’s set to be a public repository, and click **Create repository**.
4. Choose or create a directory on your computer where you would like to keep your local copy of the project.
5. Click **Clone in Sourcetree**.  
   
6. Choose a folder on your computer for the local copy of the repository and click **Clone**.

=== “GitHub and GitHub Desktop”

::

1. Sign up with [GitHub](https://bitbucket.org/) and log on.
2. Click the plus sign and select **New repository**:  
   
3. Type a repository name, make sure it’s set to be a public repository, and click **Create repository**.
4. Click **Set up in Desktop** to open the repository in GitHub Desktop:  
   Set up in desktop button
5. Choose a folder on your computer for the local copy of the repository and click **Clone**.

It might not look like much has happened, but you now have:

* Git running on your computer
* A repository at an online Git host
* A local copy of the repository on your computer

## Next steps

* Take a look at Git basics <../getting-started-git-basics/>\_\_
* Try setting up a Git wiki <../../recipes/recipes-git-wiki>\_\_

# Git basics

This page describes the workflows and commands you’ll use in the recipes, and provides a few hints to get you out of trouble.

!!! hint You don’t have to read this page all the way through. Just refer to the parts you need. The recipes link directly to the instructions you need.

## Workflows

There are many possible Git workflows, but only two in this guide:

* Git Centralized Workflow <../../recipes/recipes-centralized-workflow/>\_\_: all work is done in a single branch
* GitHub Flow <../../recipes/recipes-gitflow/>\_\_: different pieces of work are done in different branches

Centralized workflow ~~~~~~~~~~~~~~~~~~~~

In the centralized workflow, everyone works on a single branch (usually called master). Changes are committed to the local repository on each contributor’s own computer. From time to time, each user pushes changes to a remote Git repository.

.. figure:: ../img/git-centralized-workflow.png :alt: Diagram of the Git centralized workflow

Diagram of the Git centralized workflow

Here are the operations a contributor performs when working in the centralized workflow.

–8<– “centralized-workflow-snippet.html”

In the event that two people create conflicting changes, they can be manually resolved and then merged.

Although this workflow is called “centralized,” it doesn’t really resemble the old centralized source control model. The central remote repository is not a single source of truth, because every contributor has their own copy of the entire repository.

The following recipes use the centralized workflow:

* Edit a Git wiki <../../recipes/recipes-git-wiki/>\_\_
* Collaborate using centralized Git workflow <../../recipes/recipes-centralized-workflow/>\_\_

GitHub flow ~~~~~~~~~~~

In the GitHub flow, you start a new branch based on master whenever you start a group of related tasks. How you organize those tasks, and how you decide which ones belong in a branch together, is up to you.

.. figure:: ../img/github-flow.png :alt: Diagram of the GitHub flow

Diagram of the GitHub flow

Just like in the centralized workflow, you commit changes frequently to your own local repository. When you push to the remote repo, you create a copy of your branch there so that others can review your changes before they are merged back into master.

Here’s how your day looks in the GitHub flow.

–8<– “github-flow-snippet.html”

Merge conflicts in your working branches are less likely, because own your branches and other people don’t necessarily work in them with you. Conflicts are more likely to happen between your working branch and the master branch on the remote repo. Before you push your branch to the remote repo, you can pull <#pull>\_\_ from master again and merge any conflicts locally.

!!! hint After you’ve pushed, approved, and merged your work, you can delete your working branch or keep it around for further work. When you start work on a different part of the project, remember to switch to master and pull again before creating a new branch.

The following recipe uses the GitHub flow:

* Managing docs with GitHub flow </recipes/recipes-gitflow/>\_\_

## How to Git

These are the commands that make up the steps in the Git workflows.

Pull ~~~~

The Git pull command fetches and downloads content from your remote repository, automatically merges the changes with your local repository, and updates everything so that your repository matches the latest version of everything on the remote. It’s a good idea to pull after making sure you’re on the right branch and before starting to work on the content.

=== “Sourcetree”

::

1. Make sure you’re on the right branch in the correct repository:
   * The bold text under **Branches** tells you the branch
   * The tab at the top of the screen tells you the repository
2. Select **Repository > Pull** or click the **Pull** button.  
   

=== “GitHub Desktop”

::

In GitHub Desktop, you can *fetch* and *pull* separately. Fetch gets the latest updates from origin but doesn’t update your local working copy with the changes. After you click **Fetch origin**, the button changes to **Pull Origin**.

1. Make sure you’re on the right branch in the correct repository:
   * The bold text under **Current branch** tells you the branch
   * The bold text under **Current repository** tells you the repository
2. Select **Repository > Pull** or:
   1. Click the **Fetch origin** button.  
      Fetch origin button
   2. Click the **Pull origin** button.  
      Pull origin button

=== “Linux command line” 1. Make sure you’re on the right branch in the correct repository: $ git branch \* master $ git remote -v origin https://github.com/pconrad-fb/markdown.git (fetch) origin https://github.com/pconrad-fb/markdown.git (push) 1. Type the git pull command: $ git pull

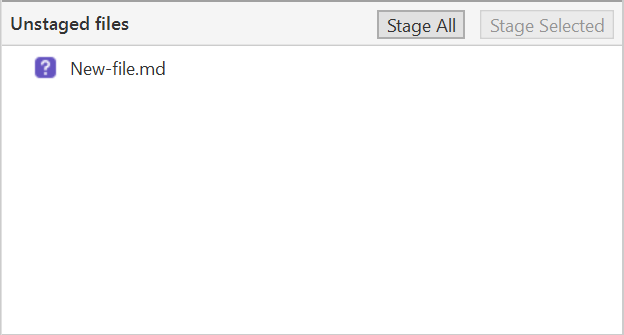
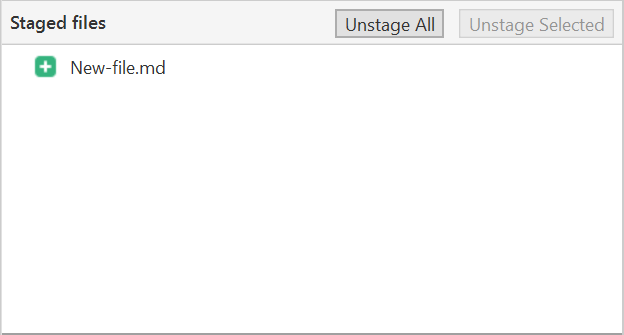
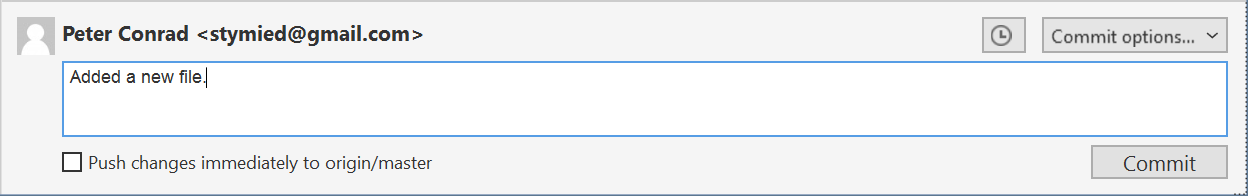
Stage and commit ~~~~~~~~~~~~~~~~

Git knows when you make changes to your files. When you want to save those changes to Git, you must do two things:

* *stage* them, which tells Git which changes you intend to keep
* *commit* them, which saves the changes.

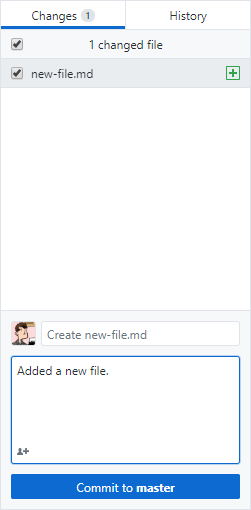
=== “Sourcetree” In Sourcetree, you stage and commit your files in two operations.

::

1. Make sure you’re on the right branch in the correct repository.
2. Look for the files you changed in the Unstaged files pane. Select the files you want to stage—in most cases, you can just click **Stage All.**  
   
3. Make sure you see the right files in the Staged files pane.  
   
4. Type a short commit message and click **Commit**.  
   

=== “GitHub Desktop” In GitHub Desktop, you can stage and commit your files in one step.

::

1. Make sure you’re on the right branch in the correct repository.
2. Look for the files you changed in the Changes tab. Unselect any files you don’t want to change—most of the time, you can leave all the checkboxes checked.  
   
3. Type a short commit message.
4. Make sure the **Commit** button refers to the correct branch (“Commit to master,” for example).
5. Click **Commit to [branch]**.

=== “Linux command line”

::

1. Make sure you’re on the right branch in the correct repository.
2. Use git status to see what changes are not yet staged.
3. Stage any changes you plan to commit. In many cases, you can stage all the changes at once like this:

* $ git add \*

1. Commit the changes, adding a descriptive message:

* $ git commit -m "Type your descriptive message here."

!!! hint If you are changing files but not adding or deleting any files, you can often stage and commit all in one line with commit -am like so: $ git commit -am "Type your descriptive message here."

Push ~~~~

=== “Sourcetree”

::

1. Make sure you’re on the right branch in the correct repository.
2. Select **Repository > Push** or click the **Push** button.  
   

=== “GitHub Desktop”

::

1. Make sure you’re on the right branch in the correct repository.
2. Select **Repository > Push** or click the **Push origin** button.  
   Push origin button

=== “Linux command line”

::

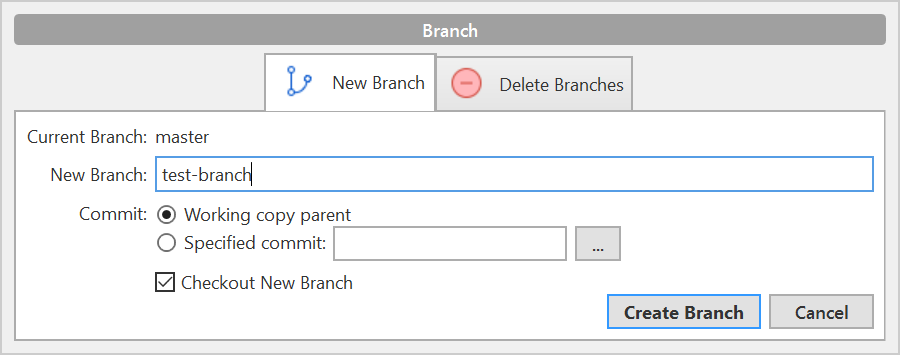
1. Make sure you’re on the right branch in the correct repository.
2. Push, specifying the remote (usually origin) and the branch. For the recipes where you are working on the master branch, the command looks like this:

* $ git push origin master
* Of course, since git knows what branch you’re on and where your remote is, you can sometimes just type git push.

Create a branch ~~~~~~~~~~~~~~~

=== “Sourcetree”

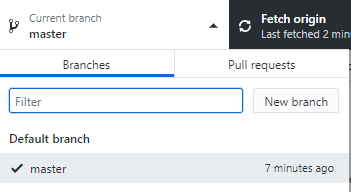
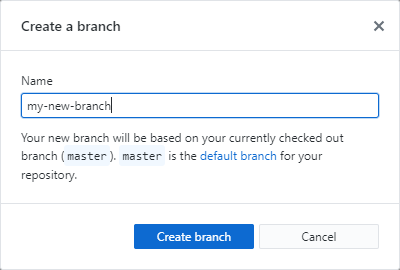
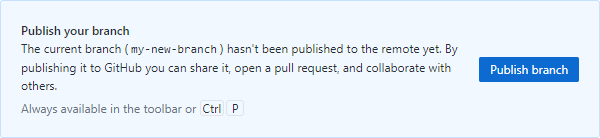
::

1. [**Pull**](../../getting-started-getting-started-git-basics/#pull) from master.
2. Click the **Branch** button:  
   
3. Type a descriptive name and click **Create Branch**.  
   
4. Look under **Branches** to see that you’re on the new branch.

You can switch to a different branch by clicking it in the list of branches.

=== “GitHub Desktop”

::

1. [**Pull**](../../getting-started-getting-started-git-basics/#pull) from master.
2. Click the **Current branch** tab and click **New branch**:  
   
3. Type a descriptive name and click **Create branch**:  
   
4. Click **Publish branch**:  
   
5. Look under **Branches** to see that you’re on the new branch.

You can switch to a different branch by clicking it in the list of branches.

=== “Linux command line”

::

1. Pull from master to make sure you have the latest changes: ``` $ git checkout master Already on ‘master’ Your branch is up to date with ‘origin/master’. $ git pull Already up to date.

* ```

1. Create a new branch and switch to it with git checkout -b. Example: $ git checkout -b test-branch Switched to a new branch 'test-branch'

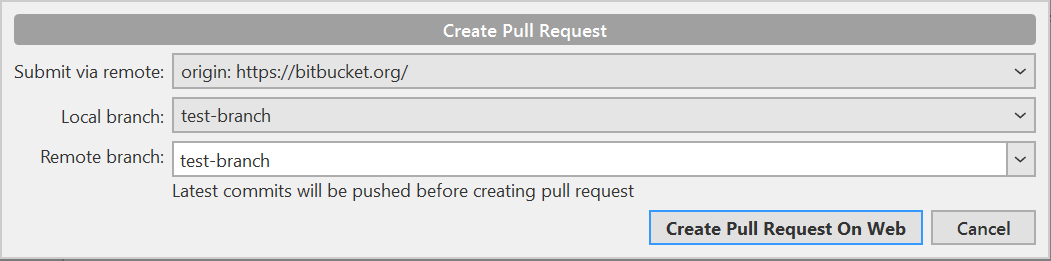
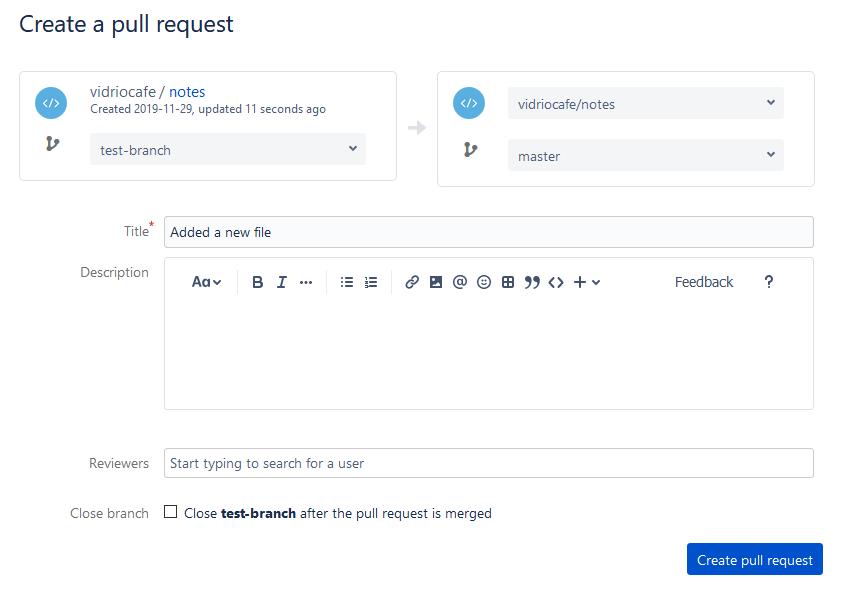
You can switch to any existing branch by typing git checkout <branch-name> without the -b. Example: $ git checkout test-branch Switched to branch 'test-branch'

!!! hint You can’t switch branches with uncommitted changes. You have to commit before switching to a new branch.

Create a pull request ~~~~~~~~~~~~~~~~~~~~~

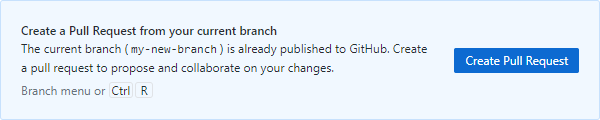
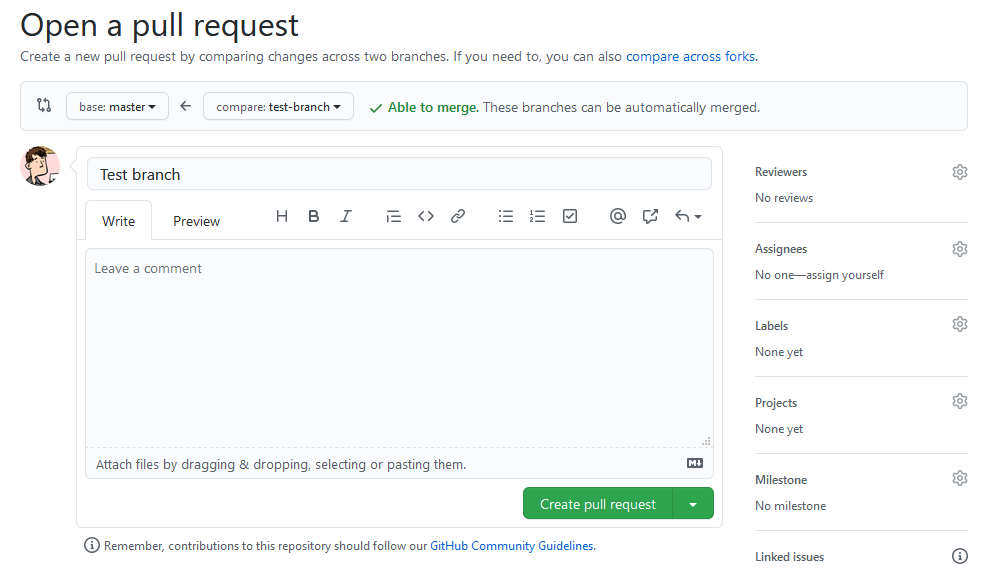
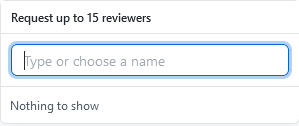
=== “Bitbucket and Sourcetree”

::

1. Click **Repository > Create pull request**.
2. In the dialog that appears, click **Create Pull Request On Web**:  
   
3. Type a description, add reviewers, and click **Create pull request**:  
   

=== “GitHub and GitHub Desktop”

::

1. When you push, the banner with the Push button changes to read “Create a pull request from your current branch.” Click **Create Pull Request**:  
   
2. The browser opens a page with a form for creating a pull request: 
3. Click the gear next to **Reviewers** to add reviewers:  
   
4. Click **Create pull request**.

=== “Linux command line”

::

1. Take a look at the output from the push command:

$ git push origin test-branch   
Enumerating objects: 14, done.  
Counting objects: 100% (14/14), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (10/10), done.  
Writing objects: 100% (10/10), 4.39 KiB | 1.10 MiB/s, done.  
Total 10 (delta 2), reused 0 (delta 0)  
remote: Resolving deltas: 100% (2/2), completed with 1 local object.  
remote:   
remote: Create a pull request for 'test-branch' on GitHub by visiting:  
remote: https://github.com/pconrad-fb/markdown/pull/new/test-branch  
remote:   
To https://github.com/pconrad-fb/markdown.git  
 \* [new branch] test-branch -> test-branch

1. Copy the URL from the line after Create a pull request into a browser.
2. Follow the instructions on screen. If you get stuck, you can see some hints in the instructions for Bitbucket or GitHub.

Approve and merge ~~~~~~~~~~~~~~~~~

Merge your branch online in the web interface of your Git host.

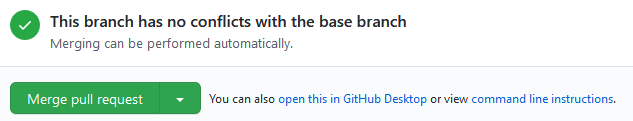
=== “Bitbucket”

::

1. When your pull request is sufficiently approved, click **Merge**: Merge button

=== “GitHub”

::

1. When your pull request is sufficiently approved, click **Merge**:  
   

## Getting out of trouble

If you get out into the woods with Git, there’s usually a way to get back—but for this kind of magic, you have to go to the command line.

Working in the wrong branch ~~~~~~~~~~~~~~~~~~~~~~~~~~~

You’ve edited a file in the wrong branch. What you’d like to be able to do is undo those changes, switch branches, then re-do them. Actually, it would be even better to lift those changes off of the wrong branch, laying them gently on top of the branch you meant to be in. Fortunately, Git provides a command called stash that does exactly that.

1. Make sure you’re in the right directory.
2. Use git status to check what branch you’re on and what changes Git knows about.
3. Make sure you’re in the branch where you were erroneously working. For example:

* ::
* $ git checkout the-wrong-branch Switched to branch ‘the-wrong-branch’

1. Stash your uncommitted changes:

* ::
* $ git stash

1. Switch to the branch you wish you had been working in:

* ::
* $ git checkout -b the-wrong-branch Switched to branch ‘the-wrong-branch’

1. Use stash to apply the changes there:

* ::
* $ git stash apply

Editing the wrong file ~~~~~~~~~~~~~~~~~~~~~~

You opened a file to look at it, but then your cat walked across the keyboard. You’re not sure what was added or deleted. You just want to go back to the way things were at the last commit. For this, use checkout—it’s not just for switching branches, but also for fixing changes to files.

1. Make sure you’re in the right directory.
2. Use git status to check what branch you’re on and what changes Git knows about.
3. If necessary, switch to the correct branch. For example:

* ::
* $ git checkout the-branch Switched to branch ‘the-branch’

1. Use git status to see what files were accidentally modified. For example:

* ::
* $ git status On branch master Changes not staged for commit: (use “git add …” to update what will be committed) (use “git checkout – …” to discard changes in working directory)
* modified: dont-change-this.md

1. Use git checkout -- <file> to undo the changes. For example:

* ::
* git checkout – dont-change-this.md

!!! hint The output of the git status command tells you how to use git checkout this way.

Staged too soon ~~~~~~~~~~~~~~~

You edited the right file the right way, but then you added it to the staging area too hastily. You don’t want to undo your changes to the file, but you would like to remove it from the next commit. This is one of the uses of reset. You can also use reset to do more drastic rollbacks—you can undo entire commits if needed.

1. Make sure you’re in the right directory.
2. Use git status to check what branch you’re on and what changes Git knows about.
3. If necessary, switch to the correct branch. For example:

* ::
* $ git checkout the-branch Switched to branch ‘the-branch’

1. Use git status to see what files were accidentally modified. For example:

* ::
* $ git status On branch master Changes to be committed: (use “git reset HEAD …” to unstage)
* renamed: README.md -> README  
   modified: dont-commit-this.md

1. Use git reset to remove the file from the next commit. For example:

* ::
* git reset HEAD dont-commit-this.md

!!! hint The output of the git status command tells you how to use reset to unstage changes.

Merge conflicts ~~~~~~~~~~~~~~~

When Git is unable to automatically merge, it means that two changes happened in the same place in the same file. That means you need to edit the file and decide which of the two changes to keep.

When you open the file, the merge conflict looks like this:

::

<<<<<<< HEAD Some content that was changed by one person ======= Other content that someone else changed >>>>>>> 9af9d3b

HEAD is a pointer to the most recent commit in the branch you’re on. The other label can be another branch name or a number representing another commit.

All you need to do is decide which version of the content you want to keep and then delete the merge conflict markers (<<<<<<<, =======, >>>>>>>).

After you’ve resolved all the changes in that way, just commit <#stage-and-commit>\_\_ again.

More trouble ~~~~~~~~~~~~

For more help and advice, check out Dangit, Git!?! <https://dangitgit.com/>\_\_

# Publishing tools

https://people.mbi.ucla.edu/leec/docs/gitpublish/intro.html

Sphinx https://www.sphinx-doc.org/en/master/

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license.

It was originally created for the Python documentation, and it has excellent facilities for the documentation of software projects in a range of languages. Of course, this site is also created from reStructuredText sources using Sphinx! The following features should be highlighted:

Output formats: HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text  
Extensive cross-references: semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information  
Hierarchical structure: easy definition of a document tree, with automatic links to siblings, parents and children  
Automatic indices: general index as well as a language-specific module indices  
Code handling: automatic highlighting using the Pygments highlighter  
Extensions: automatic testing of code snippets, inclusion of docstrings from Python modules (API docs), and more  
Contributed extensions: more than 50 extensions contributed by users in a second repository; most of them installable from PyPI

Sphinx uses reStructuredText as its markup language, and many of its strengths come from the power and straightforwardness of reStructuredText and its parsing and translating suite, the Docutils.

Sphinx is a Python documentation generator.

It requires Python, which is installed by default in Linux and Mac OS X systems. For Microsoft Windows systems, see Installing Python on Windows if you need help installing Python and two useful installation utilities (easy\_install and pip).

After you have Python installed, simply use the following command (in a command window):

easy\_install -U Sphinx

Elevated privileges (i.e. administration rights) should not be required.

The Sphinx builder can produce a number of output formats (e.g. HTML, PDF). PDF files can be produced using the LaTeX builder (more complicated) or using the a direct PDF builder called rst2pdf (see below). Rst2Pdf

rst2pdf is a tool for transforming reStructuredText to PDF using ReportLab. To install rst2pdf on Windows you also need Python because rst2pdf is coded in python.

Rst2pdf uses ReportLab, which can be installed using:

easy\_install reportlab

Again, in Windows, there may be a problem with the required Microsoft Visual Studio version. While running setup.py for package installations, Python 2.7 searches for an installed Visual Studio 2008. The solution is to define VS90COMNTOOLS variable to point to Tools directory of Visual Studio:

SET VS90COMNTOOLS=%VS100COMNTOOLS%

How to install rst2pdf on Windows?

Download rst2pdf source from https://code.google.com/p/rst2pdf/downloads/list  
Unzip the file to an rst2pdf folder.  
Goto the the rst2pdf folder which contains setup.py file.  
Run python setup.py install command and it will be installed.  
To convert any .rst file to PDF file Run rst2pdf myfile.rst command and you are done.  
  
http://rst2pdf.ralsina.me/  
User Manual: http://ralsina.me/static/manual.pdf

Pandoc Integrated solutions ReST Editor for Eclipse

The ReST editor for Eclipse is a plug-in for the Eclipse IDE. If Sphinx is installed, it can also be used to create (and build) Sphinx projects from within Eclipse. The following presentation documents the use of the editor.

This ReST editor has several advantages, namely:

integrated spell-checking using Hunspell4Eclipse  
contextual ReST syntax help  
sections outline rearrangement

This document opened in the Eclipse ReST Editor.

# Take notes

http://blog.getreu.net/projects/tp-note/tp-note–manual.html

# Edit a Git wiki

\*\* Does this work? Try it \*\*

A Git repository comes with a wiki, where people can read and collaboratively edit documentation. You can create a wiki to document projects or code stored in the repository, or you can just use a repository for its wiki capability.

!!! note A Git wiki is a second repository attached to your repository. You clone, pull, and push to the main repository and the wiki separately.

A Git wiki uses Git wiki structure <../../tools/tools-publishing#git-wiki-structure>\_\_ and it’s one way to develop content for publishing in

## Ingredients

.. raw:: html

.. raw:: html

.. raw:: html

Markdown editor

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Git

.. raw:: html

.. raw:: html

.. raw:: html

## Set up a wiki on your hosted repository

The easiest way to set up the wiki is by logging onto your Git host and adding it there.

=== “Bitbucket and Sourcetree” 1. In a browser, log on to Bitbucket. 1. Under “Repository settings” look for “Features” and click **Wiki**. 1. Select **Public wiki** and save.  
|Wiki dialog|

=== “GitHub and GitHub Desktop” The GitHub documentation on Wikis <https://docs.gitlab.com/ee/user/project/wiki/>\_\_ is helpful. Here are the basic steps:

1. In a browser, log on to GitHub.
   1. Click the **Settings** button: |Settings button|
   2. Scroll down to “Features” and select **Wikis**: |Features section of settings|

## Edit content on the host

If you just want to add a few pages to the wiki online, there’s no more setup to do! Just go to your repository, click **Wiki**, and you’ll see buttons for creating and editing pages.

!!! hint To add a page in a new folder, make the folder part of the new filename. For example: morestuff/newpage.md adds newpage.md in a folder called morestuff.

## Clone the wiki to a local repository

There are advantages to working with wiki files locally, on your computer:

* It’s much easier to add folders and move files around
* You can work on it even when you’re offline
* You can use whatever Markdown editor you want
* Others can collaborate with you

To work on the wiki locally, clone the wiki to a local repository:

=== “Bitbucket and Sourcetree” 1. Go to your online repository and click **Wiki**. 2. Click **Clone wiki** then **Clone in Sourcetree**.  
|Clone in Sourcetree button| 1. Make sure the local path shows the directory where you want to clone the repository, and click **Clone**.

=== “GitHub and GitHub Desktop” 1. Go to your online repository and click the **Wiki** button:  
|Wiki button| 1. Copy the **Clone this wiki locally** URL. 1. In GitHub Desktop, click **File > Clone repository**. 1. Paste the URL, make sure the local path shows the directory where you want to clone the repository, and click **Clone**.

=== “Linux command line” 1. Go to your online repository and click **Wiki**. 1. Copy the URL (or command and URL) to clone the wiki repository. 1. On the command line, navigate to the directory where you want to clone the repository. 1. Use git clone and the URL to clone the repository. Example:  
git clone https://my\_name@bitbucket.org/my\_name/markdown-stuff.git/wiki

## Work with the content locally

On your computer, go to the directory where you cloned the wiki. There should be a directory called wiki containing a file called Home.md containing the Markdown source for the welcome page the wiki displayed when you created it online.

You can now work with the wiki using the Git centralized workflow <../../tools/tools-git-basics#centralized-workflow>\_\_ or the GitHub flow <../../tools/tools-git-basics#github-flow>\_\_, using the familiar cycle of pull, edit, commit, push.

## Tutorial

Here’s a quick tutorial that shows how to organize pages in the wiki.

Try creating some content ~~~~~~~~~~~~~~~~~~~~~~~~~

Try making the following changes:

* Add a folder called stuff.
* Using your favorite Markdown editor, make a file called something.md inside stuff, with the following contents:
* ::
* # Something
* Yes, there’s *something* here! Now go [home](../Home).

You should now have a directory structure that looks like this:

::

Home.md stuff/ something.md

Take a look ~~~~~~~~~~~

After you commit and push the changes, take a look in your online wiki:

1. Go to your online repository and click **Wiki**.
2. View the page tree of the wiki. For example:
   * In Bitbucket, click the name of the wiki.
   * In GitHub, click **Pages**.
3. Navigate to the page you created. |Screenshot of hte wiki page|
4. Try the home link.

## Next steps

Try the following recipe, which is very similar:

* Collaborate using the Git centralized workflow <../recipes-centralized-workflow/>\_\_

.. |Wiki dialog| image:: ../img/recipes-git-wiki-create.png .. |Settings button| image:: ../img/github-settings.png .. |Features section of settings| image:: ../img/git-wikis.png .. |Clone in Sourcetree button| image:: ../img/recipes-git-repo-bb-clone-wiki.png .. |Wiki button| image:: ../img/github-wiki-button.png .. |Screenshot of hte wiki page| image:: ../img/recipes-git-wiki-bb-something.png

# Collaborate using centralized Git workflow

You can use Git to help a team collaborate on informal or internal documents in a very simple way.

The centralized workflow can be an effective way for content creators, managers, and engineers to collaborate on non-production content such as specifications, planning documents, newsletters, internal documentation, and the like.

.. figure:: ../img/git-centralized-workflow.png :alt: Diagram of centralized Git workflow

Diagram of centralized Git workflow

In this workflow, everyone works on the same branch. This makes things easy for simple collaboration.

!!! Note If you’re publishing a large website or documentation set in a production environment with with a substantial number of collaborators, you should consider using GitHub Flow <../recipes-gitflow/>\_\_ instead.

## Ingredients

.. raw:: html

.. raw:: html

.. raw:: html

reStructuredText editor

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Git

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Publishing tools (optional)

.. raw:: html

.. raw:: html

.. raw:: html

!!! hint Plan ahead based on the publishing tool you want to use. If you’re planning to use a wiki, MkDocs, or Hugo, organize your docs in Git wiki structure <../../tools/tools-publishing#git-wiki-structure>\_\_ and add YAML frontmatter from the start. If you’re creating large documents with Pandoc, think about whether you need to work on content in pieces and then use Pandoc to assemble everything.

## Working with content

This recipe uses the centralized Git workflow. Pick an editor <../../tools/tools-editors/>\_\_, then make sure you have Git set up and try each step below. The steps work together like a heartbeat that keeps content safe and synchronized for all collaborators.

–8<– “centralized-workflow-snippet.html”

If you need to Publish content <../../tools/tools-publishing/>\_\_, you can use one of these recipes:

* Publish documentation with MkDocs <../recipes-mkdocs/>\_\_
* Publish a website with Hugo <../recipes-mkdocs/>\_\_
* Create a document with Pandoc <../recipes-pandoc/>\_\_

!!! hint Remember not to publish content into a Git repository. The repo is only for storing your reStructuredText and other source files.

## Next steps

Check out the following recipes :

* Manage docs with Gitflow <../recipes-gitflow/>\_\_
* Create an eBook <../recipes-pandoc-ebook/>\_\_
* Write a Word doc <../recipes-pandoc-word/>\_\_
* Publish a PDF <../recipes-pandoc-pdf/>\_\_
* Present slides <../recipes-slides/>\_\_

# Manage docs with GitHub Flow

Once your team or project reaches a certain size, branching is a good way to keep people from accidentally interfering with each other’s work. It’s also a great way to stage and test code or content before migrating to production. There are many branching strategies, some of them quite complicated. The one I like, and that I have used for documentation in the past, is called GitHub Flow <https://scottchacon.com/2011/08/31/github-flow.html>\_\_. It’s simple but effective. The **Pull-Work-Commit-Push** steps happen within a working branch.

.. figure:: ../img/github-flow.png :alt: Diagram of GitHub flow

Diagram of GitHub flow

## What you need

.. raw:: html

.. raw:: html

.. raw:: html

reStructuredText editor

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Git

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Publishing tools

.. raw:: html

.. raw:: html

.. raw:: html

!!! hint Plan ahead based on the publishing tool you want to use. If you’re planning to use a wiki, MkDocs, or Hugo, organize your docs in Git wiki structure <../../tools/tools-publishing#git-wiki-structure>\_\_ and add YAML frontmatter from the start. If you’re creating large documents with Pandoc, think about whether you need to work on content in pieces and then use Pandoc to assemble everything.

## Working with content

Pick an editor <../../tools/tools-editors/>\_\_, then make sure you have Git set up and try each step below. The steps work together like a heartbeat that keeps content safe and synchronized for all collaborators. –8<– “github-flow-snippet.html”

!!! hint After you’re done with that part of the project, you can delete your working branch or keep it around for further work. Before starting work on a different part of the project, remember to switch to master and pull again.

If you need to Publish content <../../tools/tools-publishing/>\_\_, you can use one of these recipes:

* Publish documentation with MkDocs <../recipes-mkdocs/>\_\_
* Publish a website with Hugo <../recipes-mkdocs/>\_\_
* Create a document with Pandoc <../recipes-pandoc/>\_\_

!!! hint Remember not to publish content into a Git repository. The repo is only for storing your reStructuredText and other source files.

## Next steps

Check out the following recipes :

* Manage docs with Gitflow <../recipes-gitflow/>\_\_
* Create an eBook <../recipes-pandoc-ebook/>\_\_
* Write a Word doc <../recipes-pandoc-word/>\_\_
* Publish a PDF <../recipes-pandoc-pdf/>\_\_
* Present slides <../recipes-slides/>\_\_

# Publish a website with Hugo

https://github.com/fisodd/hugo-restructured

Hugo <https://gohugo.io/>\_\_ is a very powerful open-source static site generator that includes tools for organizing content, adding extensions, and even creating dynamic logic. There is way too much to document here, so the focus of this recipe will be getting started and a few basics.

To install Hugo, use your operating system’s package manager:

* Linux: apt-get <https://help.ubuntu.com/community/AptGet/Howto> **or yum <http://yum.baseurl.org/>**
* macOS: Homebrew <https://brew.sh/>\_\_
* Windows: Chocolatey <https://chocolatey.org/>\_\_

## Ingredients

.. raw:: html

.. raw:: html

.. raw:: html

reStructuredText editor

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Hugo

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Git (optional)

.. raw:: html

.. raw:: html

.. raw:: html

!!! hint Hugo works best with reStructuredText files that are organized in Git wiki structure <../../tools/tools-publishing#git-wiki-structure>**. You can even use Git wiki <../recipes-git-wiki/>** to develop the content, so long as you add YAML frontmatter as you go.

This recipe goes well with:

* Centralized Git workflow <../recipes-centralized-workflow/>\_\_
* GitHub flow <../recipes-gitflow/>\_\_

## Setting up your site

Once Hugo is installed, you can create a new site by typing hugo new site my-project, which provides some instructions when it runs:

::

$ hugo new site my-project Congratulations! Your new Hugo site is created in /home/pconrad/git/my-project.

Just a few more steps and you’re ready to go:

1. Download a theme into the same-named folder. Choose a theme from https://themes.gohugo.io/ or create your own with the “hugo new theme ” command.
2. Perhaps you want to add some content. You can add single files with “hugo new /.”.
3. Start the built-in live server via “hugo server”.

Visit https://gohugo.io/ for quickstart guide and full documentation.

!!! Hint For source control and collaboration, create the project in a Git repo.

## Adding a theme

To add a theme, you use the git submodule add command from within your site directory. Each theme in Hugo’s official Complete List <https://themes.gohugo.io/>\_\_ includes instructions for adding the submodule from the correct Git repository. Example:

::

$ git submodule add https://github.com/budparr/gohugo-theme-ananke.git themes/ananke Cloning into ‘/home/pconrad/git/my-project/themes/ananke’… remote: Enumerating objects: 8, done. remote: Counting objects: 100% (8/8), done. remote: Compressing objects: 100% (8/8), done. remote: Total 1839 (delta 2), reused 1 (delta 0), pack-reused 1831 Receiving objects: 100% (1839/1839), 4.33 MiB | 1.26 MiB/s, done. Resolving deltas: 100% (1022/1022), done.

After you’ve installed the theme, add it to the configuration file config.toml. Example:

::

$ echo ‘theme = “ananke”’ >> config.toml

Edit the config.toml file to change other things about the site, such as the site title or the base URL.

## Working with content

You can create a new page with hugo new <path>. For example, to create a new post in the posts directory, type:

::

hugo new posts/my-first-post.md

When it’s created, all this page contains is YAML frontmatter:

::

|  |
| --- |
| title: “My First Post” |
| date: 2020-08-28T18:42:02-07:00 |
| draft: true |

You must add reStructuredText content before you can preview the page. A page in Hugo must contain both frontmatter and reStructuredText content. If either is missing, the page shows 404 page not found when you try to preview it.

Frontmatter can signal content status, including publish and expiry dates, and can contain variables. You can use the variables in templates and in content by creating your own Hugo shortcodes. Variables and shortcodes are outside the scope of this recipe, but the Hugo website has a lot of documentation.

If you are migrating content into Hugo from a Git wiki, MkDocs site, or some other source, you will need to add frontmatter to any pages that don’t already have it.

## Local preview

To preview your content, start the Hugo server:

::

$ hugo server -D

The -D option tells Hugo to include draft pages in the preview. If you omit this option, any page with draft:true in the frontmatter is ignored.

.. figure:: ../img/hugo-preview.png :alt: Screenshot of Hugo live preview

Screenshot of Hugo live preview

Hugo is a bit finicky about previewing and building content, and there are a few reasons why pages might not show up in the preview. A first troubleshooting step is to make sure each file contains both frontmatter and reStructuredText, and that it doesn’t have a publish date in the future or expiry date in the past.

Images ~~~~~~

Images in Hugo normally go in the static directory. Here, I’ve added a subdirectory called images and when I add the image in my reStructuredText editor it looks like this:

.. figure:: ../../static/images/whatever.png :alt: An image

An image

That makes sense, since that is the correct relative path to the file. When the site is built for local preview, however, the result is this:

Unfortunately, that won’t work. The actual relative path to the image is: ../../images/whatever.png instead.

In other words, when you add an image whose relative path is correct with regard to the reStructuredText file, you must remove static/ from the path to make it work in the preview and the built site. If you are using an editor that lets you preview the images as you work, then you have to break all the images to get them to work in Hugo.

One solution might be to upload all your images to the web first, then use absolute paths—but this will make it more difficult to move the website or change the paths later.

## Building and publishing

The hugo command builds the website in a directory called public. To publish the site, use FTP to transfer the contents of that directory to a folder on a webserver.

!!! hint To prevent Git from tracking changes to the public directory, create a file called .gitignore at the top level directory of the Hugo project with the following contents:

public/

If you use git add to add your .gitignore file to change tracking, then it will apply to anyone who clones the repo—meaning that no one will add built HTML pages to Git.

# Overview

You can use Pandoc to create a Word document that other people can work on, for upload to Google Drive, or for use with other publishing tools.

## Ingredients

.. raw:: html

.. raw:: html

.. raw:: html

reStructuredText editor

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Pandoc

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Word, Google Drive, or LibreOffice

.. raw:: html

.. raw:: html

.. raw:: html

## Creating the Word document

The command for creating the document is simple. With a single reStructuredText file, it looks like this:

::

pandoc -o my\_document.docx reStructuredText.rst

Special Pandoc formatting ~~~~~~~~~~~~~~~~~~~~~~~~~

Pandoc includes a number of formatting tricks that you might find useful. One of the most useful is fenced div syntax, which uses groups of colons.

You can use fenced div syntax to create columns using nested divs without writing <div> tags in HTML. Take a look at this example:

::

:::::::::::::: {.columns} ::: {.column width=“50%”}

Left column:

* Bullet
* Bullet
* Bullet

::: ::: {.column width=“50%”}

::: ::::::::::::::

That translates to a <div class="columns"> containing two <div class="column"> tags. Pandoc uses these to create two columns in the Word doc. Each div can be signified with as few as three colons in a row; in the example, the outer div uses more colons for readability.

You can use curly braces to define attributes such as identifiers, classes, and key/value pairs on headers, images, and a few other elements in Pandoc. If you’re using Pandoc to create long Word documents, this is handy because you can set anchors on headings and link to them internally.

You can also use an attribute to scale an image:

::

Alt text

Alt text

When Pandoc renders the image, it is scaled to a percentage of the container where it resides (a column, for example). The alt text is used for a caption.

Concatenating multiple files ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

You can specify as many files as you like. For example:

::

pandoc -o my\_document.docx chapter\_1.md chapter\_2.md

Title file ~~~~~~~~~~

You can specify a title, author, licensing, and other information about the book in a file called title.txt at the front of the book, containing YAML that Pandoc uses when it generates the file.

Example:

::

title: How to do things with reStructuredText and Git author: Peter S. Conrad language: en-US …

Creating a Word doc with a title file and multiple reStructuredText files looks like this:

::

pandoc -o my\_document.docx title.txt chapter\_1.md chapter\_2.md

Images ~~~~~~

When your reStructuredText includes images, use relative paths. For example:

::

An image

An image

In the above example, the images directory is at the same level as the file containing the reStructuredText file; the relative path goes up a directory from the reStructuredText file and then down into the images directory to find the image.

When Pandoc follows these relative links, it starts from the directory where you typed the pandoc command. If you want Pandoc to find your images, either run the command from a directory where the relative links to the images make sense, or copy the images to a place where the relative links can find them.

Using a reference document ~~~~~~~~~~~~~~~~~~~~~~~~~~

When you use Pandoc to convert reStructuredText to Word, you can apply the theme and styles from another Word document called a *reference document.* For example:

::

pandoc –reference-doc another.docx -o my\_document.docx title.txt chapter\_1.md chapter\_2.md

# Create an ePub book

You can use Pandoc to create an ePub book suitable for tablets and phones or for self-publishing on Amazon <https://kdp.amazon.com/en\_US/bookshelf>\_\_.

# Overview

You can use Pandoc to create a PDF, which is a pleasant way to share content that you don’t want other people to edit along the way.

# Grab a web page with Pandoc

Converting a web page to reStructuredText on the fly is a small task that can be very helpful if you need to grab content for editing or add information from the web to your notes.

rst2html5slides - Presentations from restructuredtext files … rst2html5slides.readthedocs.io They have new and great visual effects such as slide transitions and positional effects; A web presentation is portable. It runs on any modern browser. On the other …

Easy Slide Shows With reST & S5 - Docutils - SourceForge docutils.sourceforge.io ? docs ? user ? slide-shows Mar 30, 2006 ? Uses normal reStructuredText as input. One section per slide. Each first-level section is converted into a single slide. XHTML output.

hovercraft/presentations.rst at master · regebro/hovercraft … github.com ? regebro ? hovercraft ? blob ? master ? docs Calling them “slides” is not really relevant in an impress.js context, as they can overlap and doesn’t necessarily slide. The name “steps” is better, but it’s also more …

darkslide · PyPI pypi.org ? project ? darkslide ReStructuredText. Your ReST source files must be suffixed by .rst or .rest (.txt is not supported); Use headings for slide titles; Separate your slides using an …

hieroglyph: Easy, Beautiful Slides with Restructured Text … www.yergler.net ? 2012/03/13 ? hieroglyph Mar 13, 2012 ? hieroglyph: Easy, Beautiful Slides with Restructured Text. I was happy to have my talk proposal accepted for PyCon this year, and happy with …

Using rst for presentations - MorganGoose.com morgangoose.com ? blog ? 2010/09/12 ? using-rst-for-… Sep 12, 2010 ? S5 is a slide show format based entirely on XHTML, CSS, and JavaScript. And rst2s5 takes a reStructuredText document and complies it into the …

Hieroglyph Documentation - Read the Docs readthedocs.org ? hieroglyph ? downloads ? pdf ? latest PDF Jul 4, 2020 ? In addition to mapping ReStructured Text sections to slides, you can create a slide at any point in your document using the slide directive. The …

Creating presentations using restructured text | Ralsina.Me ralsina.me ? stories ? BBS52 May 21, 2009 ? So, how do you create slides using rst2pdf? It’s very simple. So simple, I will use an example file. This is my first slide ——– … Create a DZSlides presentation ==============================

You can use Pandoc to create a standalone slide presentation in DZSlides format, which is useful for presenting when you aren’t sure what software will be available. DZSlides creates bold, attractive slides that discourage the use of too much text.

## Ingredients

.. raw:: html

.. raw:: html

.. raw:: html

Markdown editor

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

Pandoc

.. raw:: html

.. raw:: html

.. raw:: html

.. raw:: html

A browser

.. raw:: html

.. raw:: html

.. raw:: html

## Markdown for DZSlides

Start your Markdown file with YAML metadata or a simple block like this:

::

% Title % Author Name % Date

DZSlides uses the horizontal rule (---) as a separator between slides. A level one or level two heading is a section title.

Here’s some sample Markdown:

::

# H1 or H2 is a Section Title

Normal Text or Slide Title

* Bullet
* Bullet

Normal text is big enough for a title or announcement on a slide.

.. figure:: ../img/slides-dzslides.png :alt: Screenshot of a DZSlides slide

Screenshot of a DZSlides slide

The layout of slides in DZslides is simple; you probably won’t find yourself using tables or columns a lot.

Images ~~~~~~

When your Markdown includes images, use relative paths. For example:

::

An image

An image

In the above example, the images directory is at the same level as the file containing the Markdown file; the relative path goes up a directory from the Markdown file and then down into the images directory to find the image.

When Pandoc follows these relative links, it starts from the directory where you typed the pandoc command. If you want Pandoc to find your images, either run the command from a directory where the relative links to the images make sense, or copy the images to a place where the relative links can find them.

After you create the presentation, which is an HTML file, you need to keep the images and the presentation together. If you copy your presentation to a thumb drive without the images, the images won’t work.

!!! hint It’s a good idea to create the Markdown presentation in a folder with all the images it needs, then use Pandoc to build the presentation in the same folder. You can copy the entire folder wherever you need it, knowing that all the images for the presentation will work.

Pandoc provides syntax for scaling an image:

::

An image

An image

.. figure:: ../img/slides-dzslides-images.png :alt: Screenshot of another DZSlides slide

Screenshot of another DZSlides slide

When Pandoc renders the image, it is scaled to a percentage of the container where it resides (the slide, in most cases). The alt text is used for a caption.

## Creating the presentation

The command for creating the document is simple. With a single Markdown file, it looks like this:

::

pandoc -t dzslides -s myslides.md -o myslides.htm

The -s option tells Pandoc to create a standalone presentation, including all the CSS, HTML, and JavaScript needed to display it. You can view the presentation by opening the resulting HTML file in a browser.

## Next steps

* Check out the DZSlides template <../../resources/templates/#dzslides>\_\_

Examples of reST markup Headers

# Section Header

## Subsection Header

Lists

* A bullet list item
* Second item
  + A sub item
* Spacing between items separates list items
* Different bullet symbols create separate lists
* Third item

1. An enumerated list item
2. Second item
   1. Sub item that goes on at length and thus needs to be wrapped. Note the indentation that must match the beginning of the text, not the enumerator.
      1. List items can even include
      * paragraph breaks.
3. Third item
4. Another enumerated list item
5. Second item

Images

.. image:: /path/to/image.jpg

Named links

A sentence with links to Wikipedia\_ and the Linux kernel archive\_.

.. \_Wikipedia: https://www.wikipedia.org/ .. \_Linux kernel archive: https://www.kernel.org/

Anonymous links

Another sentence with an anonymous link to the Python website\_\_.

\_\_ https://www.python.org/

N.B.: named links and anonymous links are enclosed in grave accents (`), and not in apostrophes (’). Literal blocks

::

some literal text

This may also be used inline at the end of a paragraph, like so::

some more literal text

.. code:: python

print(“A literal block directive explicitly marked as python code”)

# Templates

Use these templates as starting points for slide presentations.

# Glossary

admonition A note, warning, or other call-out that draws attention to a block of content. Bash A Unix/Linux shell and language that lets users execute commands and programs. branch A series of commits representing changes to one or more files in a Git repository. change In Git, a modification, creation, or deletion of a file. checkout In Git, to switch to a different branch or restore a file. chmod A shell command that changes file permissions. client Hardware or software that accesses a *service*. A web browser is a client to a webserver, and a Git client accesses a service provided by a Git host. clone In Git, to make a complete local copy of a *remote repository* so you can work with the files on your computer. cloud Someone else’s computer. commit In Git, to save your changes to the *local repository*. A group of changes saved together using the commit command. CSS Cascading Style Sheets, a style sheet language for defining the look and feel of a document written in HTML or another markup language. div A division or section in an HTML document, specified with a <div> tag. dynamic site A site that is generated or modified at the time it is displayed. See *static site.* fenced Delineated with a series of characters. For example, a *fenced code block* is marked with three backticks (``) at the top and bottom. frontmatter Metadata at the start of a file, often including information such as the title, author, and date. FTP File Transfer Protocol, a way of exchanging files between your computer and a server. Git A distributed source control system. Git wiki An additional repository, attached to a Git repository, for the purpose of displaying and managing content (often, content about the Git repository). Git wiki structure A content structure in which the display paths or URLs to content pages are defined by the directory paths of the files that make up the content. host A server, often on the web. A \*Git host\* provides access to Git repositories, a \*web host\* provides access to websites, and so on. HTML HyperText Markup Language, the standard markup language for creating web pages. JavaScript A programming language that enables the creation of interactive features on web pages. JSON JavaScript Object Notation, a format for storing and transporting data. LaTeX A document preparation system for high-quality typesetting. Linux A family of Unix-like operating systems first designed by Linus Torvalds in 1991. local On your own computer. Markdown A simple \*markup language\* originally designed as an easy way to write HTML pages. markup language A way of indicating display formatting and other information within a document. merge In Git, to combine two sets of changes into one branch. merge conflict In Git, a merge that cannot be completed automatically because the same parts of the files have been modified in both sets of changes. metadata Information about the content in a file, or about the file itself. package manager A tool for installing software. SeeGetting started <../../getting-started/>`\_\_. PDF Portable Document Format, a file format developed by Adobe in 1993 to present documents consistently across software, hardware, and operating systems. permissions Settings that specify what actions can be taken and by whom. For example, file permissions can specify who can read, write, or execute the file. pull In Git, to fetch and merge changes from a *remote* to your *local repository*. pull request In Git, a set of proposed changes to be approved and then merged into a branch. push In Git, to upload changes from your local *repository* to a *remote*. Python A popular programming language. Python Markdown extensions A set of additional features and syntax provided with the Python implementation of Markdown. recursion See *recursion*. remote A *remote repository*. remote repository A version of your project that is hosted on the network or online rather than on your computer. repo Repository. repository In Git, a collection of files and the entire history of all changes made to them. reStructuredText XXX Definition goes here Samba Open source software that runs on Unix or Linux to enable communication with Windows clients over a network. script A computer program that automates the execution of commands or tasks. server A computer or application that provides a service for other programs or devices, which in turn are called *clients*. Sharepoint A web-based collaboratoin platform that integrates with Microsoft Office and is often used to manage and store documents. shell A program that lets users type commands for the operating system to execute. source control A way of tracking and managing changes to code or other content. stage In Git, to specify which changes to save in the next *commit.* stash In Git, to record the current state of the working directory and revert the working directory to the previous *commit.* static site A site composed of HTML pages or other documents that are made available exactly as stored, as opposed to a *dynamic site* whose pages are rendered on the fly when they are requested. A static site often performs better and can be more secure, but lacks some of the capabilities of a *dynamic site.* static site generator A tool that builds a *static site.* TOML Tom’s Obvious, Minimal Language, a text format for configuration files or metadata. Unix A family of operating systems designed at Bell Labs in the 1970s, that Linux is like. unstage In Git, to remove previously *staged* changes from the upcoming *commit.* WebDAV Web Distributed Authoring and Versioning, an HTTP extension that lets clients perform remote operations on content. wiki A structured HTML site, often edited and managed by the readers themselves, that collects information about a particular topic. working branch In Git, a temporary branch created for working on a particular set of content or code changes. working directory The folder on your local computer where you store the content you are editing. WYSIWYG What You See Is What You Get, an editing experience that mimics the appearance of the document in its final form. YAML Yaml Ain’t Markup Language, a text format for configuration files or metadata.

# Links

Editors

* Atom <https://atom.io/>\_\_
* BbEdit <https://www.barebones.com/products/bbedit/bb>\_\_
* Emacs <https://www.gnu.org/software/emacs/>\_\_
* Sublime Text <https://www.sublimetext.com/>\_\_
* Vim <https://www.vim.org/>\_\_
* Visual Studio Code <https://code.visualstudio.com/>\_\_

Git

* Bitbucket <https://bitbucket.org>\_\_
* Centralized Git Workflow <../recipes/recipes-centralized-workflow/>\_\_
* Dangit, Git!?! <https://dangitgit.com/>\_\_
* Git <https://git-scm.com>\_\_
* Git doesn’t have to be hard <https://levelup.gitconnected.com/git-doesnt-have-to-be-hard-e1e115be6668>\_\_
* GitHub <https://bitbucket.org/>\_\_
* GitHub Desktop <https://desktop.github.com/>\_\_
* GitHub Flow <https://scottchacon.com/2011/08/31/github-flow.html>\_\_
* GitLab <https://about.gitlab.com/>\_\_
* Git wiki structure <../../tools/tools-publishing#git-wiki-structure>\_\_
* Git Wikis <https://docs.gitlab.com/ee/user/project/wiki/>\_\_
* Sourcetree <https://www.sourcetreeapp.com/>\_\_
* The Git Book <https://git-scm.com/book/en/v2>\_\_