Use Git and GitHub

- Set up Git on your Mac
- Git/GitHub Glossary and Illustrated Reference
- Enable a virtual machine
- Create a repository clone on your local machine
- Daily update and rebase the master branch on your local repository
- Create and checkout a new feature branch
- Checkout an existing feature branch
- Verify your work with the make docs command

Set up Git on your Mac

When to do this:	Once, to get Git set up and configured on your Mac. Mac comes with Apple's Git version. This sucks. Get the real Git.
Prerequisites:	A Mac.

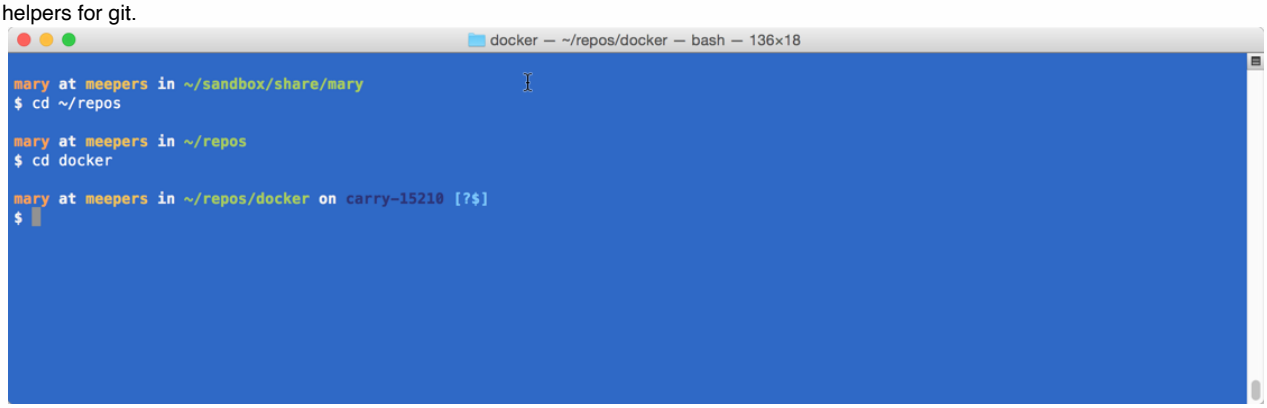
1. Create a GitHub account.
2. Install Git correctly.
3. Install XCode on your Mac.

    > We used to do these in reverse order, but installing Git first prevents you from having to clobber the potentially-bad version of Git that ships bundled with XCode.)

4. Generate an SSH key and add it to GitHub ad it to both your GitHub profile and the `ssh-agent`.
5. Change to your /tmp directory.

    ```
    $ cd /tmp
    ```

    We all use a profile which originates from Jessie Frazelle one of Docker's top developers. This profile includes some cool prompt helpers for git.

    We recommend you get this profile.

6. Clone the profile set from moxiegirl.

```
$ git clone git@github.com:moxiegirl/my_profile.git
```

7. Change into the `my_profile` directory.

```
$ cd my_profile
```

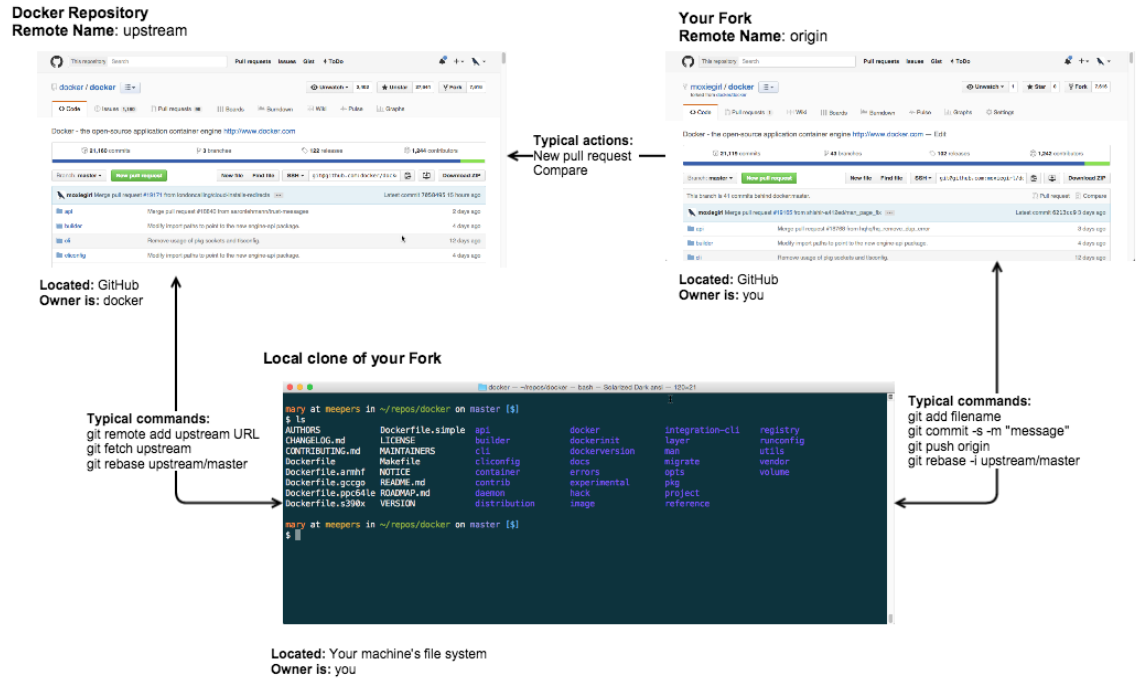
8. Copy the profile files to yours.
   This will overwrite any existing profile files you have.

```
$ cp .* ~/
```

9. Close any open terminals and reopen them.

Git/GitHub Glossary and Illustrated Reference

You need this if you are new to Git or just for refresher.



Glossary

Term	Description
clone	*n.* A clone is a copy of a repository that lives on your computer instead of in GitHub. *v.* The act of copying a repository with the Git `clone` command. With your clone you can edit the files in your preferred editor and use Git to keep track of your changes without having to be online. It is, however, connected to the remote version so that changes can be synced between the two. You can push your local changes to the remote to keep them synced when you're online.

branch	A branch is a version of a repository code. It does not affect the primary or `master` branch allowing you to work freely without disrupting the "live" version. When you've made the changes you want to make, you can merge your branch back into the `master` branch to publish your changes. Also, a Git command for managing branches.
checkout	
commit	*n.*, A commit, or "revision", is an individual change to a file (or set of files). A Git command that saves a change in a repository It's like when you *save* a file, except with Git, every time you save it creates a unique ID (a.k.a. the "SHA" or "hash") that allows you to keep record of what changes were made when and by who. Commits usually contain a commit message which is a brief description of what changes were made.
diff	*n.*, The *difference* in changes between two commits. Also, a Git command that shows changes between commits, commit and working tree, etc
fetch	A git command to get the latest changes from an online repository (like GitHub.com) without merging them in. Once these changes are fetched you can compare them to your branches in your local clone and merge these changes.
fork	A fork is a personal copy of another user's repository that lives on your GitHub account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes. You can also keep your fork up to date by pulling in updates from the original.
Git	Git is an open source program for tracking changes in text files. It was written by the author of the Linux operating system.
GitHub	A social coding cloud application is built on top of Git.
hash	See SHA or commit.
HEAD	A reference to the currently checked out commit. In normal states, it's actually a symbolic reference to the branch you have checked out - if you look at the contents of `.git/HEAD` you'll see something like "`ref: refs/heads/master`". The branch itself is a reference to the commit at the tip of the branch
LGTM	Looks Good to Me – a short hand way for a pull request reviewer to indicate approval of a change. Documentation changes require the approval of two doc maintainers in addition to technical reviewers.
master	The main branch. Analogous to Subversion's *trunk*.
merge	A Git command that takes the changes from one branch (in the same repository or from a fork), and applies them into another. The GitHub interface also has a **Merge** action which you can perform once your pull request is fully reviewed and approved.
pull	A Git command for fetching *in* changes *and* merging them. For instance, if someone has edited the remote file you're both working on, you'll want to *pull* in those changes to your local copy so that it's up to date. Writers normally don't do use this command.
pull request	A GitHub feature that allows you to submit changes from your fork back to the original repository owner. Pull requests allow contributors to review, comment, and suggest changes before a merge.
push	Pushing refers to sending your committed changes to a remote repository such as GitHub.com. For instance, if you change something locally, you'd want to then *push* those changes so that others may access them.
rebase	A merge without the annoying merge commit. This is how SVN always works! I still can't figure out whether rebase is something that everyone should use all the time or that advanced users should use for special situations. Also, a Git command to forward-port local commits to the updated upstream head.
ref	A reference to a single commit. This is a pointer. If you think of the commit history like a graph, then this points to a single node in that graph. It could be a tag, or it could be the tip of a branch, or it could be *HEAD*, the current state of your repository. In actual fact, a branch is simply a pointer and nothing more. The actual tree structure that represents the "branch" is the commit graph, and the branch itself is just a pointer into that graph. Each repository has its own set of refs which are not necessarily shared with other repositories.
remote	An instance of a repository. You can pull changes from or push changes to remotes. origin is the your fork (owned by you) that lives on GitHub and allows you to work without impeding with others. When you clone your fork, Git automatically creates a remote for it called `origin` in the `.git/config` file. `upstream` is the "source of truth" the repository owned by the Docker on GitHub.
repository	A repository contains all of the project source files (including documentation), and stores each file's revision history. Repositories can have multiple collaborators and can be either public or private.

SHA	A unique ID (a.k.a. the "SHA" or "hash") for a commit.
ssh key	SSH keys are a way to identify yourself to an online server, using an encrypted message. It's as if your computer has its own unique password to another service. GitHub uses SSH keys to securely transfer information from GitHub.com to your computer.
tip	The last commit on a branch, i.e. the most recent commit, is referred to as the *tip* of that branch, or sometimes the *head*. This is a leaf in the commit graph, if you like to think in terms of graphs.
upstream	When talking about a fork, the original repository is often referred to as the "upstream", since that is the main place that other changes will come in from. The fork you are working on is then called the "downstream".

Enable a virtual machine

When to do this:	As needed. A virtual machine allows you to run Docker CLI commands on your local system.
Prerequisites:	• Running on a Mac or Windows machine. • Docker Machine is installed either through Docker Toolbox or directly.

1. List the available machines.

```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     ...
default            virtualbox   Stopped   ...
```

   If you need to create a new machine:

```
$ docker-machine create --driver virtualbox default
Creating VirtualBox VM...
Creating SSH key...
Starting VirtualBox VM...
Starting VM...
To see how to connect Docker to this machine, run: docker-machine env
default
```

   If you had to create a machine, you can skip the next step as your machine is already started.
2. Start a machine.

```
$ docker-machine start default
(default) Starting VM...
Started machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

3. Get the environment configuration for the machine you started.

```
$ docker-machine env moxie
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/mary/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval "$(docker-machine env default)"
```

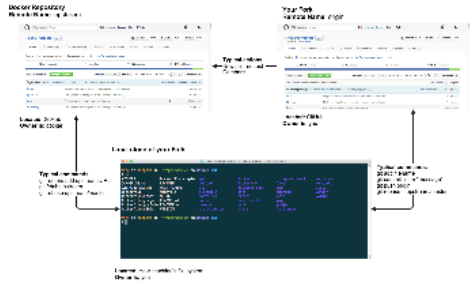
4. Set the environment configuration for the machine you started.

```
$ eval "$(docker-machine env default)"
```

Create a repository clone on your local machine

When to do this:	Infrequently. When you first start working on a repo or if you need to start over with a “fresh” repo
Prerequisites:	A GitHub account.

Click to enlarge



The procedure below assumes you are cloning the `docker/toolbox` repository. **So, make sure you sub in your repo for `toolbox` in the examples below.**

1. Go to the Docker repository on Github.
2. Fork the `docker/repository` to your GitHub user account.
3. In a shell back on your system, change directory (`cd`) into your `~/repos` directory.

```
$ cd ~/repos
```

If you don't have a repos directory, make one.

```
$ mkdir ~/repos
```

4. Clone your fork to your `repos` directory.
In the repo URL, your GitHub username should appear, in this example the username is `moxiegirl`.

```
$ git clone git@github.com:moxiegirl/toolbox.git
Cloning into 'toolbox'...
<snip>
Checking connectivity... done.
```

5. Change directory into your new repository.

```
$ cd toolbox
```

6. Create an upstream remote that points to the original source repository.

```
$ git remote add upstream git@github.com:docker/toolbox.git
```

7. Set `no_push` on the `upstream` remote.

```
$ git remote set-url --push upstream no_push
```

8. Check the cat `.git/config` contains both remotes.

```
$ cat .git/config
[core]
 repositoryformatversion = 0
 filemode = true
<snip>
[remote "upstream"]
 url = git@github.com:docker/toolbox.git
 fetch = +refs/heads/*:refs/remotes/upstream/*
 pushurl = no_push
```

9. Edit the `.git/config` file in your favorite editor.
10. Add a fetch for pull requests:

```
 [remote "upstream"]
 url = git@github.com:docker/toolbox.git
 fetch = +refs/heads/*:refs/remotes/upstream/*
 fetch = +refs/pull/*/head:refs/remotes/upstream/pull/*
 pushurl = no_push
```

Line 4 is what allows you to pull another user's code branch.

11. Initialize the local repository by running a `fetch` of references for the `upstream master`, followed by a `rebase`, and `push`.

To do this, follow the steps in the next procedure, Daily update and rebase the master branch on your local repository (also shown below, in short form).

```
$ git fetch upstream
From github.com:docker/toolbox
 * [new branch]      buc-branch -> upstream/buc-branch
 * [new branch]      build-test -> upstream/build-test
 * [new branch]      master     -> upstream/master

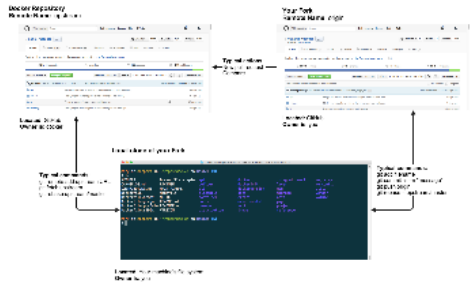
$ git rebase upstream/master
Current branch master is up to date.

$ git push origin
Everything up-to-date
```

Daily update and rebase the master branch on your local repository

When to do this:	**Daily or before you create a pull request**. This ensures your local master and fork are even with the upstream/master. If you have a long-lived feature branch, you should rebase your feature branch frequently.
Prerequisites:	• A repository clone on your local machine. • An upstream remote defined.

Click to enlarge



This procedure gets the latest changes from the `docker/repository master` branch and updates your local clone with them. Then, you push the changes to the `master` branch on your fork. Why do this? This ensures your local clone and fork are even with `docker/repository master` branch. Remember, while your working on your feature branch, other people are working on theirs and merging changes into master. If your feature branch gets too far behind, you'll have a more complex merge to do when you are ready to send your changes with a pull request.

1. Go to the root of your repository.

```
$ cd ~/repos/toolbox
```

2. Verify the branch you are on with a `git status` command.

```
 $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

   You should also have nothing to commit at this point. Remember, ***never*** work directly in your `master` branch.

3. If you are not on the `master` branch, checkout the `master` branch.

```
 $ git checkout master
```

4. Fetch the latest references from the upstream remote.
   The references detail any changes that went into the repository since you last did a `fetch`.

```
 $ git fetch upstream
remote: Counting objects: 1986, done.
remote: Compressing objects: 100% (37/37), done.
remote: Total 1986 (delta 110), reused 93 (delta 93), pack-reused
1855
Receiving objects: 100% (1986/1986), 8.42 MiB | 1.63 MiB/s, done.
Resolving deltas: 100% (1064/1064), completed with 26 local
objects.
From github.com:docker/toolbox
 * [new branch]      dockercon-demopack ->
upstream/dockercon-demopack
 * [new branch]      master     -> upstream/master
 * [new tag]         v1.7.0     -> v1.7.0
 * [new tag]         v1.7.1     -> v1.7.1
 * [new tag]         v1.8.0-rc1 -> v1.8.0-rc1
 * [new tag]         v1.9.1h    -> v1.9.1h
<snip>
```

5. Rebase your local master branch.

```
$ git rebase upstream/master
First, rewinding head to replay your work on top of it...
Fast-forwarded master to upstream/master.
```

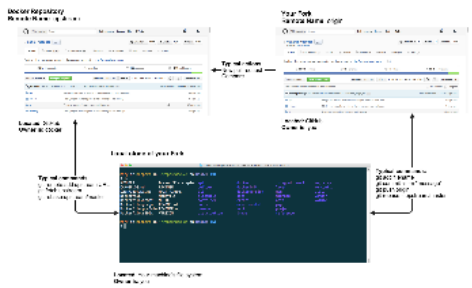
6. Push your changes to your fork.

```
$ git push origin
Counting objects: 1936, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (675/675), done.
Writing objects: 100% (1936/1936), 2.78 MiB | 1.08 MiB/s, done.
Total 1936 (delta 1057), reused 1908 (delta 1031)
To git@github.com:moxiegirl/toolbox.git
   5c0db16..cbd98e3  master -> master
```

Create and checkout a new feature branch

When to do this:	Before you start working and usually after you have updated your master branch.
Prerequisites:	A repository clone on your local machine.

Click to enlarge



1. Change to the root of your repository directory.

```
$ cd ~/repos/toolbox
```

2. Checkout the `master` branch.
   You can create a new branch from any other branch. It is a good idea to start from an updated `master` branch because you'll be sure to have latest changes.

```
$ git checkout master
```

3. Update and rebased your master branch.
   You should do this every day.
4. Create a new code branch for your feature.

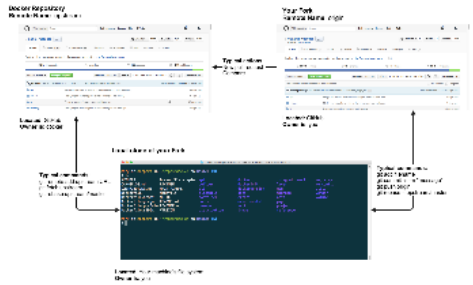
```
$ git checkout -b my-new-feature-branch
Switched to a new branch 'my-new-feature-branch'
```

   The `-b` flag is what creates the new branch. Git also switches you into that branch.

Checkout an existing feature branch

When to do this:	Whenever you need to. Typically, before you start working and usually after you have updated your master bran ch
Prerequisites:	An existing feature branch in your repository.

Click to enlarge:



1. Change directory to the root of your repository.

```
$ cd ~/repos/toolbox
```

2. Do a git status to see what branch you are currently on.
   You may be on the branch you want already.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

3. If you forgot your branch name, list your branches.

```
$ git branch -l
* master
  my-new-feature-branch
```

Git places an * (asterisk) beside your current feature branch.

4. Checkout the branch you want to work on.

```
$ git checkout my-new-feature-branch
Switched to branch 'my-new-feature-branch'
```

Verify your work with the make docs command

When to do this:	As you work to check the layout.
Prerequisites:	• A feature branch with some changes in it. • You should be in the `docs` directory of your repository.

You have to periodically display your work in a browser to check the menus and layouts are correct.

1. Change to the root of your repository.

```
$ cd ~/repos/machine
```

2. Checkout an existing or make a new branch.

```
$ git checkout my-new-feature-branch
```

3. Modify or add files and folders.
4. Enable a VM if you haven't already.

Useful commands to know

```
docker-machine ls
docker-machine start VM_NAME
docker-machine env VM_NAME
eval "$(docker-machine env VM_NAME)"
docker-machine IP VM_NAME
```

5. Change to the docs directory.

```
$ cd docs
```

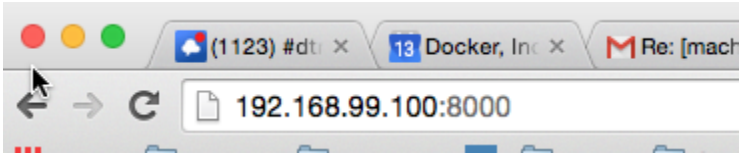
6. Get the VM of the machine.

```
$ docker-machine ip moxie
192.168.99.100
```

7. Make the documentation.

```
$ make docs
docker build -t "docs-base:flickerbox-test" .
Sending build context to Docker daemon 647.2 kB
Step 1 : FROM docs/base:latest
 ---> 688fea15ab55
Step 2 : MAINTAINER Mary Anthony <mary@docker.com> (@moxiegirl)
 ---> Using cache
 ---> d343f6510c26
Step 3 : RUN svn checkout
https://github.com/docker/docker/trunk/docs /docs/content/engine
 ---> Running in 3679eaaef763
A    content/engine/.gitignore
<snip>
338 pages created
287 non-page files copied
0 paginator pages created
0 tags created
0 categories created
in 1821 ms
Serving pages from /docs/public
Web Server is available at http://192.168.99.100:8000/ (bind
address 0.0.0.0)
Press Ctrl+C to stop
```

8. View the documentation by entering the VM"s IP address plus the port in your browser's window.



9. Check your changes in the browser.
10. Iterate until perfect or you are satisfied.

What to look for when checking your local build

Check List

In the source code, visually spot check for:

- [ ] Every line should wrap at 80 characters
- [ ] Run a spell check and verify there are no spelling errors.
- [ ] Make sure your Markdown is correct
- [ ] Sentences end in a period

Markdown Tips

Element	

If the change is more than paragraphs or it removes entire files:

- [ ] Does the build report any errors that exist in the source you are building
- [ ] Does the content on the page look correct
- [ ] Are the head levels correct meaning H1 the page title, and everything else H2 or lower
- [ ] Are notes in blue, indented, with a bold **Note**
- [ ] Code is in code font inline and in code blocks
- [ ] Images are displaying properly
- [ ] Images are optimized

If you are adding entirely new files with a change, make sure the metadata at the top of the file is correct.

Example of metadata

```
<!--[metadata]>
+++
title = "Plugins API"
description = "How to write
Docker plugins extensions "
keywords = ["API, Usage,
plugins, documentation,
developer"]
[menu.main]
parent = "mn_extend"
weight=1
+++
<![end-metadata]-->
```

- [ ] The metadata must be commented out correctly
- [ ] The `title` value should be unique or the file should have an `identifier` value
- [ ] The `keywords` are appropriate to the content
- [ ] The `weight` value places the page correctly in the documentation menu
- [ ]

Note	`> **Note**`
	` > It's important that you choose a partitioning tool that is available as an ISO so`
	` > that the Boot2Docker VM can be booted with it.`

	`# Page title`
	` ## Head 2`
	` ### Head 3`
	` #### Head 4`
	
	Try not to go beyond 3 unless you absolutely have to

How to check your work after a make docs

Check List

In the source code, visually spot check for:

- [ ] Every line should wrap at 80 characters
- [ ] Run a spell check and verify there are no spelling errors.
- [ ] Make sure your Markdown is correct
- [ ] Sentences end in a period

If the change is more than paragraphs or it removes entire files:

Markdown Tips

Element	

- [ ] Does the build report any errors that exist in the source you are building
- [ ] Does the content on the page look correct
- [ ] Are the head levels correct meaning H1 the page title, and everything else H2 or lower
- [ ] Are notes in blue, indented, with a bold **Note**
- [ ] Code is in code font inline and in code blocks
- [ ] Images are displaying properly
- [ ] Images are optimized

If you are adding entirely new files with a change, make sure the metadata at the top of the file is correct.

Example of metadata

```
<!--[metadata]>
+++
title = "Plugins API"
description = "How to write
Docker plugins extensions "
keywords = ["API, Usage,
plugins, documentation,
developer"]
[menu.main]
parent = "mn_extend"
weight=1
+++
<![end-metadata]-->
```

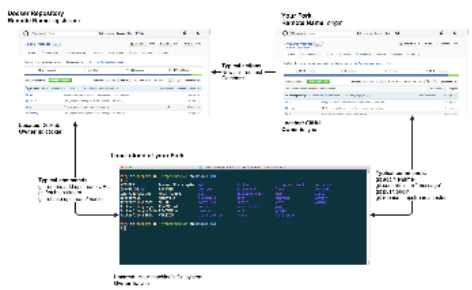
- [ ] The metadata must be commented out correctly
- [ ] The `title` value should be unique or the file should have an `identifier` value
- [ ] The `keywords` are appropriate to the content
- [ ] The `weight` value places the page correctly in the documentation menu
- [ ]

Note	`> **Note** > It's important that you choose a partitioning tool that is available as an ISO so > that the Boot2Docker VM can be booted with it.`
	`# Page title ## Head 2 ### Head 3 #### Head 4` Try not to go beyond 3 unless you absolutely have to

Add a new file or folder to Git tracking

When to do this:	Periodically as you work, commit your change. You should create one commit for each unit of work on a feature. For example, adding a new page or drafting a procedure.
Prerequisites:	• a clone on your local machine • a feature branch in that clone

Click to enlarge:



This example shows you how to add a directory to the `docs` subdirectory in a branch. Each directory should have an `index.md` file, so this

example illustrates that as well.

1. Verify you are working in the correct branch.

```
$ git status
```

   If you aren't in the right branch,check it out.

```
$ git checkout my-new-feature-branch
```

2. Change to the directory in your repository where you want to make the change.

```
$ cd docs
```

3. Add your directory to the file system.

```
$ mkdir cooldir
```

4. Check your status.

```
$ git status
On branch my-new-feature-branch
nothing to commit, working directory clean
```

   At this point your directory is empty. Git will only track a directory if it contains files.
5. Change to your new directory.
6. Add an `index.md` file to it.

```
$ touch index.md
```

   The `touch` command is a simple Linux command for creating an empty file. You can also create one with your favorite text editor. At this point git knows you have a new directory. It doesn't list the files only the directory because the directory and everything it contains is untracked.
7. Check your status.

```
$ git status
On branch my-new-feature-branch
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 ./
nothing added to commit but untracked files present (use "git add"
to track)
$ cd ..
$ git status
On branch my-new-feature-branch
Untracked files:
  (use "git add <file>..." to include in what will be committed)
 cooldir/
nothing added to commit but untracked files present (use "git add"
to track)
```

Notice your status depends on where you are in the filesystem.

8. Add the directory to Git tracking.

```
$ git add ~/repos/machine/docs/cooldir
```

9. Check your status.

```
$ git status
On branch my-new-feature-branch
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
 new file:   cooldir/index.md
```

10. Commit the change to your fork.

```
$ git commit -s -m "Adding a new directory"
```

11. Push your changes
    Your fork is on GitHub and is known to your local Git as the `origin` remote.

```
$ git push origin
fatal: The current branch my-new-feature-branch has no upstream
branch.
To push the current branch and set the remote as upstream, use
    git push --set-upstream origin my-new-feature-branch
```

You have already noticed that Git tries to prevent you from breaking things by sending you messages as you work with the command line. If this is the first time you've pushed this branch, then you have to set the upstream origin for the new branch. Git tells you this. So, set the `upstream` for this branch as your fork.

```
$ git push --set-upstream origin my-new-feature-branch
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:moxiegirl/machine.git
 * [new branch]      my-new-feature-branch -> my-new-feature-branch
Branch my-new-feature-branch set up to track remote branch
my-new-feature-branch from origin.
```

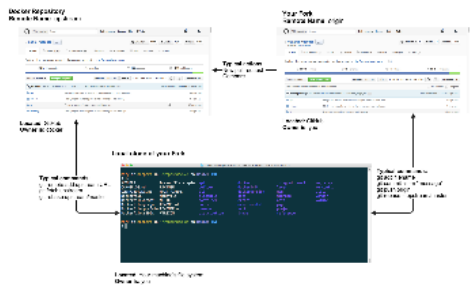
The next change you push accepts a simple command without the `--set-upstream` flag.

```
$ git push origin
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 404 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To git@github.com:moxiegirl/machine.git
   36abac3..2920840  my-new-feature-branch -> my-new-feature-branch
```

Push changes to a fork

When to do this:	Periodically as you work, commit your change. You should create one commit for each unit of work on a feature. For example, adding a new page or drafting a procedure.
Prerequisites:	• a clone on your local machine • a feature branch in that clone

Click to enlarge:



This example shows you how to add a directory to the `docs` subdirectory in a branch. Each directory should have an `index.md` file, so this example illustrates that as well.

1. Verify you are working in the correct branch and list unstaged changes.

```
$ git status
On branch carry-1830
Your branch is up-to-date with 'origin/carry-1830'.
Changes not staged for commit:
  <snip>
 modified:   install-machine.md
```

2. Stage your changes by adding them.

```
$ git add install-machine.md
```

3. Commit your change locally.

```
$ git commit -s -m "Incorporate comments"
[carry-1830 7bb79dc] Incorporate comments
 1 file changed, 20 insertions(+), 8 deletions(-)
```

4. Push your changes
   Your fork is on GitHub and is known to your local Git as the `origin` remote.

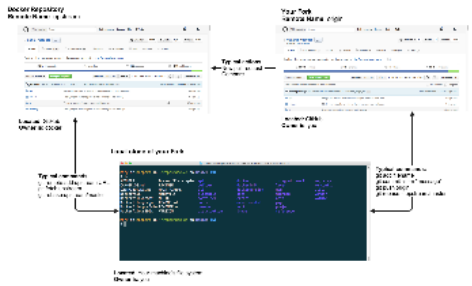
```
$ git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 849 bytes | 0 bytes/s, done.
Total 4 (delta 3), reused 0 (delta 0)
To git@github.com:moxiegirl/machine.git
   211f80a..7bb79dc  carry-1830 -> carry-1830
```

5. Go to your fork on GitHub.
6. Look for the pushed code.

Rebase, squash, and push to your fork

When to do this:	• **Daily**. This ensures your local master and fork are even with the upstream/master. If you have a long-lived feature branch, you should rebase your feature branch frequently. • Before you make a pull request to ensure your changes consist of single commit, not several. • If you have an existing pull request you need to update.
Prerequisites:	• One or more changes on a feature branch.

Click to enlarge



This procedure gets the latest changes from the `docker/repository master` branch and merges them to your feature branch. Then, you push the changes to the `master` branch on your fork. Then, you rebase your feature branch to ensure it has all the changes from `master` ***before*** you make a pull request. Why a single commit? It makes backing out a change easier.

1. Daily update and rebase the master branch on your local repository.
2. Checkout your feature branch.

```
$ git checkout my-new-feature-branch
```

3. Make sure you have pushed all your changes to your fork.
   You want to make sure your changes are in the fork in case you mess up your rebase somehow. Having the changes in the fork already make it easier to recover.

Do	Description
`git status`	Tells you if you have unstaged work or commits you haven't pushed.

`git checkout — path_to_file`	To unstage work you don't want in the branch. This removes all your work in a file.
`git commit -s -m "message" filename`	To commit work you want in your fork.
`git push origin`	Push your changes to your fork.

4. Interactively rebase your feature branch.
   This command says to interactively take the changes from the upstream/master and merge them into your branch. Interactive rebase allows you to fix (resolve) any conflicts with the changes in your feature branch.

```
$ git rebase -i upstream/master
```

   When you issue this command, Git sends you into a rebase workflow. Make sure read the information Git gives you. Remember, Git is a version control system, so you can recover from most any problem you may cause.

   a. Git opens an editor listing all the commits in this branch.
   b. Make sure only the first commit says pick; change the other `pick` instances to `squash`.

```
pick 2920840 adding thig
squash efa047f Adding in the metadata
# Rebase 36abac3..efa047f onto 36abac3 (2 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log
message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to
bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be
aborted.
#
# Note that empty commits are commented out
```

   c. Save and close the file.
   d. Git opens another editor listing all the commit messages.

```
# This is a combination of 2 commits.
# The first commit's message is:
adding thig
Signed-off-by: Mary Anthony <mary@docker.com>
# This is the 2nd commit message:
Adding in the metadata
Signed-off-by: Mary Anthony <mary@docker.com>
# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
#
# Date:      Sun Jan 10 19:22:03 2016 -0800
#
# interactive rebase in progress; onto 36abac3
# Last commands done (2 commands done):
#    pick 2920840 adding thig
#    squash efa047f Adding in the metadata
# No commands remaining.
# You are currently editing a commit while rebasing branch
'my-new-feature-branch' on '36abac3'.
#
# Changes to be committed:
# new file:   docs/cooldir/index.md
#
```

e. Leave all the messages but only one signature.

```
# This is a combination of 2 commits.
adding thig
Adding in the metadata
Signed-off-by: Mary Anthony <mary@docker.com>
# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
#
# Date:      Sun Jan 10 19:22:03 2016 -0800
#
# interactive rebase in progress; onto 36abac3
# Last commands done (2 commands done):
#    pick 2920840 adding thig
#    squash efa047f Adding in the metadata
# No commands remaining.
# You are currently editing a commit while rebasing branch
'my-new-feature-branch' on '36abac3'.
#
# Changes to be committed:
# new file:   docs/cooldir/index.md
#
```

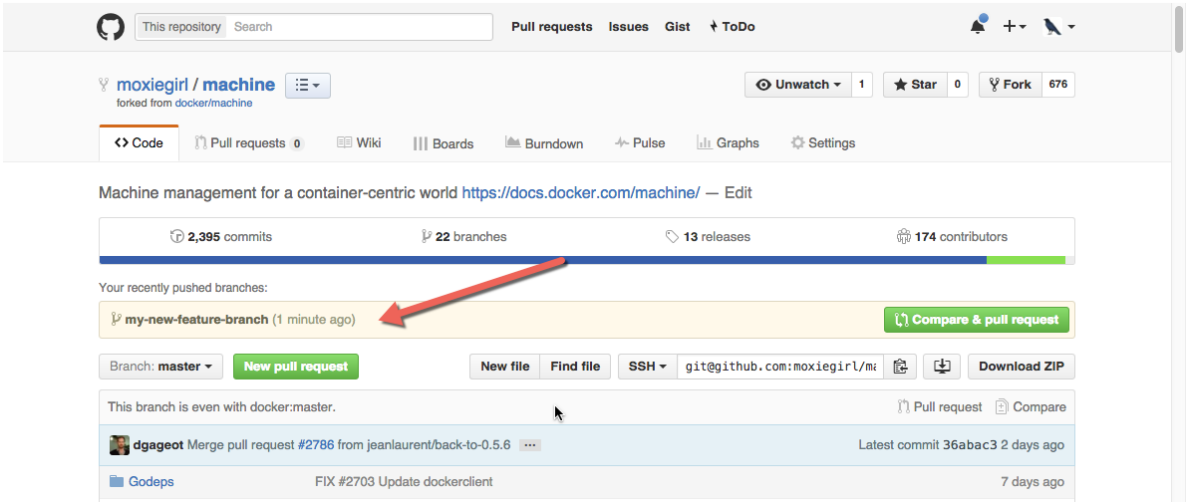
f. Save and close the file.
g. Git gives you the status of the rebase.

```
$ git rebase -i upstream/master
[detached HEAD 7f94622] adding thig
 Date: Sun Jan 10 19:22:03 2016 -0800
 1 file changed, 1 insertion(+)
 create mode 100644 docs/cooldir/index.md
Successfully rebased and updated
refs/heads/my-new-feature-branch.
```

5. Force push the updated branch to your fork.

```
$ git push -f origin
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 439 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To git@github.com:moxiegirl/machine.git
 + efa047f...7f94622 my-new-feature-branch -> my-new-feature-branch
(forced update)
```

6. Go up to your fork on GitHub and make sure the pushed code is there.



Fixing merge conflicts while rebasing

On this page:

The strange part is, `docs-alpha7-a` seemed to be okay.

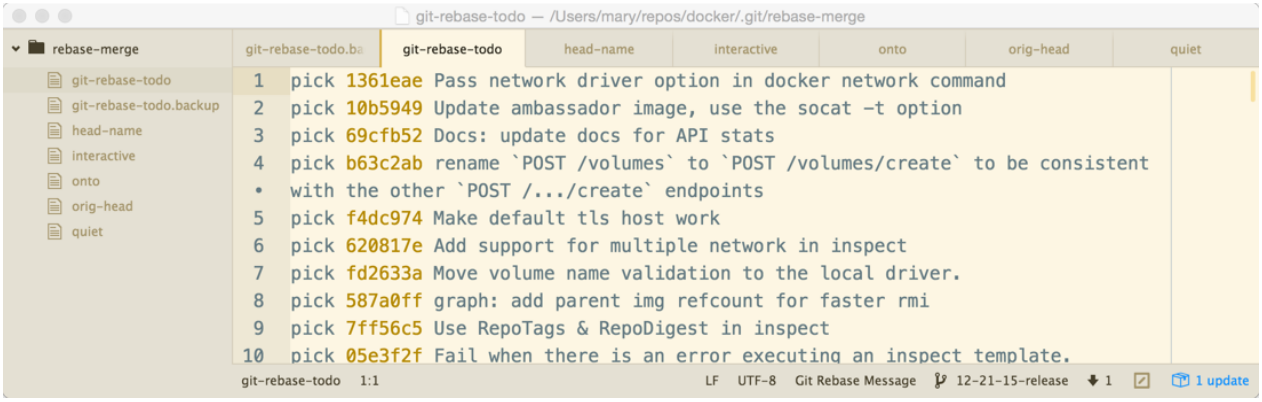
-- Vickley

Sometimes, when you rebase everything seems like it is ok. But then, you hit something you haven't seen in a while. A merge conflict. And it isn't that much fun anymore. (Except for the Alien quotes...see if you can spot them all)

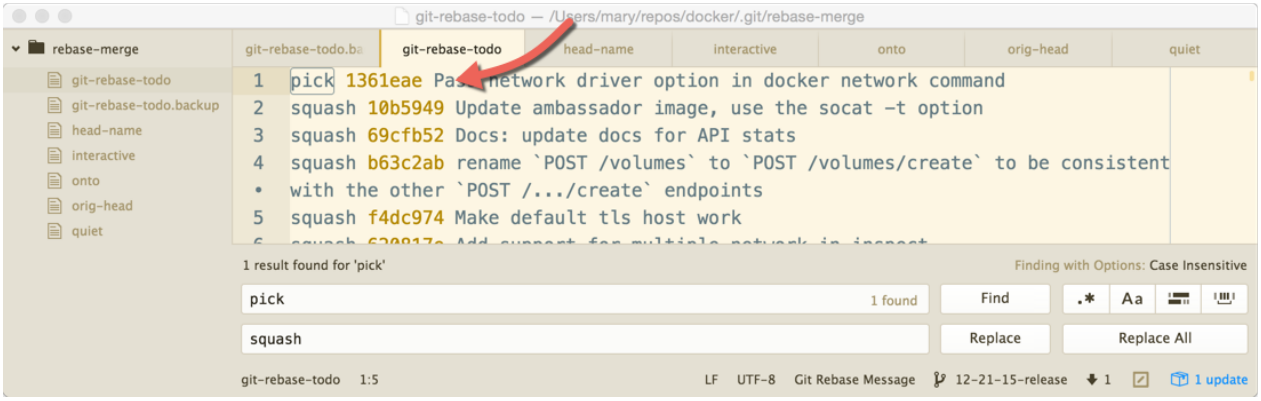
1. Daily update and rebase the master branch on your local repository.
2. Switch to the local feature branch that contains the changes you want to merge.

```
$ git checkout docs-alpha7-a
```

3. Start an interactive rebase.
4. Git prompts you to edit the "todo" list.



5. Change all "pick" to "squash" except for the first pick which should stay pick.



6. Save and close the release-merge.
   Git starts the rebase and then it says:

```
$ git rebase -i upstream/master
error: could not apply 1361eae... Pass network driver option in
docker network command

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase
--abort".
Could not apply 1361eaef7c796911b58002c248bcd158ad1272fd... Pass
network driver option in docker network command
```

At this point, you suddenly think I'm on an express elevator to hell, going down! Because you know you have a merge conflict and your rebase is stopped until you resolve it.

7. Use the git status command to see where the conflict is.

```
$ git status
interactive rebase in progress; onto 9ae51b3
Last command done (1 command done):
   pick 1361eae Pass network driver option in docker network
command
Next commands to do (49 remaining commands):
   squash 10b5949 Update ambassador image, use the socat -t option
   squash 69cfb52 Docs: update docs for API stats
  (use "git rebase --edit-todo" to view and edit)
You are currently rebasing branch 'docs-alpha7-a' on '9ae51b3'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

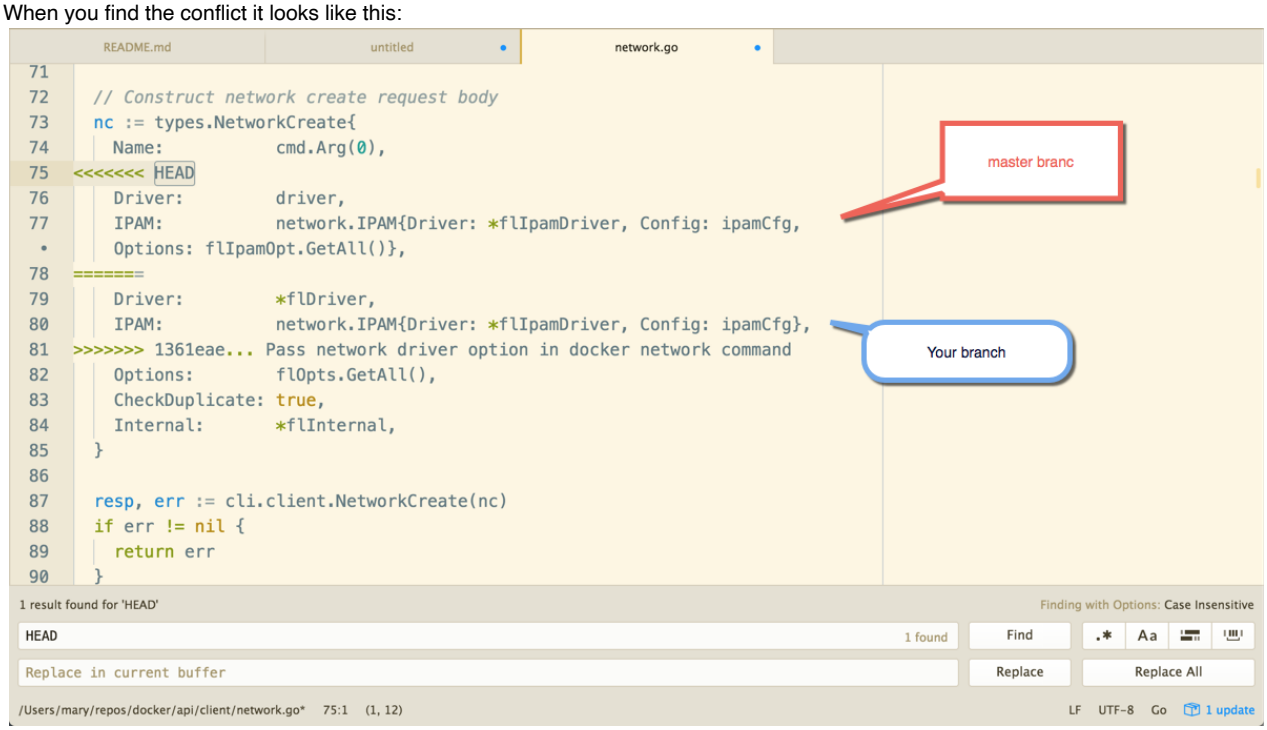
Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

 both modified:   api/client/network.go
 both modified:   api/server/router/network/network_routes.go
 both modified:   daemon/network.go
 both modified:   integration-cli/docker_cli_network_unix_test.go
 both modified:
vendor/src/github.com/docker/engine-api/types/types.go

no changes added to commit (use "git add" and/or "git commit -a")
```

It's very pretty, you think, but what are we looking for? You are looking for the `Unmerged paths`. You modified those files but so did someone else on the master branch you are rebasing against.

8. Open the first modified file for editing.
9. Search the file for <<<<<<< `HEAD` or just `HEAD`.
   When you find the conflict it looks like this:



10. Remove the markers and the content you don't want.
    Your browser does not support the HTML5 video element
11. Repeat for each remaining HEAD conflict.
12. Save and close the file.
13. Do a git status again to see what happened.

```
$ git status
interactive rebase in progress; onto 9ae51b3
Last command done (1 command done):
   pick 1361eae Pass network driver option in docker network
command
Next commands to do (49 remaining commands):
   squash 10b5949 Update ambassador image, use the socat -t option
   squash 69cfb52 Docs: update docs for API stats
  (use "git rebase --edit-todo" to view and edit)
You are currently rebasing branch 'docs-alpha7-a' on '9ae51b3'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

 both modified:   api/client/network.go
 both modified:   api/server/router/network/network_routes.go
 both modified:   daemon/network.go
 both modified:   integration-cli/docker_cli_network_unix_test.go
 both modified:
vendor/src/github.com/docker/engine-api/types/types.go

no changes added to commit (use "git add" and/or "git commit -a")
```

    The file you changed still shows as modified.
14. Use the `git add` command to mark the file's conflicts resolved.

```
$ git add api/client/network.go
```

> *Vickley*, what are you *doing*? Do not use the git commit message during a rebase! Git does the commits for you automatically.

15. Repeat step 7 through 14 until all the conflicts are resolved.
16. After you have added all the files, do a git status again.

```
$ git status
interactive rebase in progress; onto 9ae51b3
Last command done (1 command done):
   pick 1361eae Pass network driver option in docker network
command
Next commands to do (49 remaining commands):
   squash 10b5949 Update ambassador image, use the socat -t option
   squash 69cfb52 Docs: update docs for API stats
  (use "git rebase --edit-todo" to view and edit)
You are currently rebasing branch 'docs-alpha7-a' on '9ae51b3'.
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

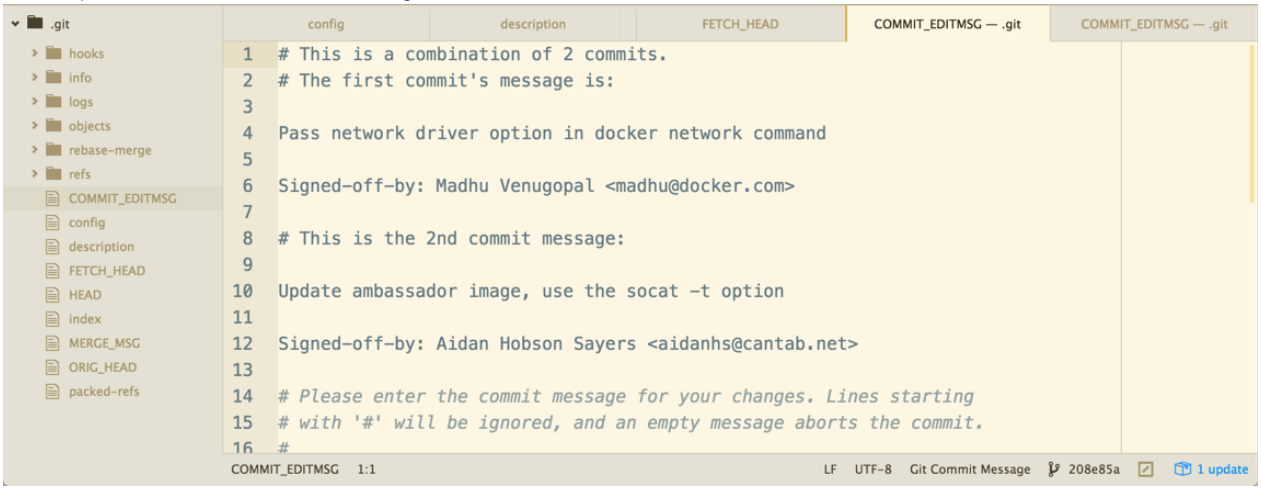
 modified:   api/client/network.go
 modified:   api/server/router/network/network_routes.go
 modified:   daemon/network.go
 modified:   integration-cli/docker_cli_network_unix_test.go
 modified:   vendor/src/github.com/docker/engine-api/types/types.go
```

As you can see, we're in the pipe five-by-five. Everything is ready to continue.

17. Tell Git to continue the rebase.

```
$ git rebase --continue
```

18. Git asks you to edit the final commit message.



19. Save and close the file to continue.
    You may have to resolve multiple conflicts. Why is that? A rebase is Git "Playing" the commits from master over your branch.
20. Continue until Git tells you it succeeded.

```
$ git rebase --continue
[detached HEAD b2f92e6] Fixed path to docker.log from Finder, added
what's new item re: com.docker.driver.amd64-linux binary, format
copyedits
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/docs-alpha7-a.
```

Aye-firmative. Ready to push to your fork.

21. Push the rebased code to your fork.

```
$ git push -f origin
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 632 bytes | 0 bytes/s, done.
Total 7 (delta 5), reused 0 (delta 0)
To git@github.com:londoncalling/pinata.git

 + b73399a...b2f92e6 docs-alpha7-a -> docs-alpha7-a (forced update)
```

Now would be a time to have a strong mocha with whipped cream, and consider this: don't try to `commit` when you are in the middle of a rebase.

MARINES. We are LEAVING! (Or how to get out of the middle of a rebase)

What do you do if you run into a problem or simply get confused in the middle of a rebase? Up to the point the Git rebase announces it is successful, you can get the heck out of an interactive rebase at any time by doing this:

```
$ git rebase --abort
```

This stops the rebase and returns you to exactly the point you were at when you started the rebase. It is as if it never happened. Now, you can start again or ask for help.

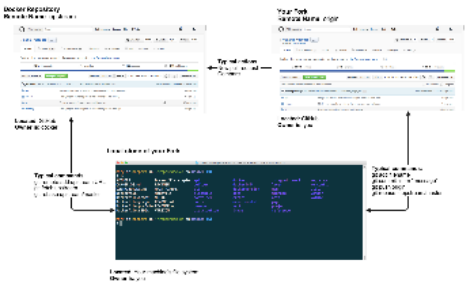
We got nukes, we got knives, we got sharp sticks (or other ways to do this)

There are lots of ways to do a rebase and lots more to Git merge than this one page teaches you. So, do some reading and this article on Git Conflict Resolution is really good.

Create a pull request

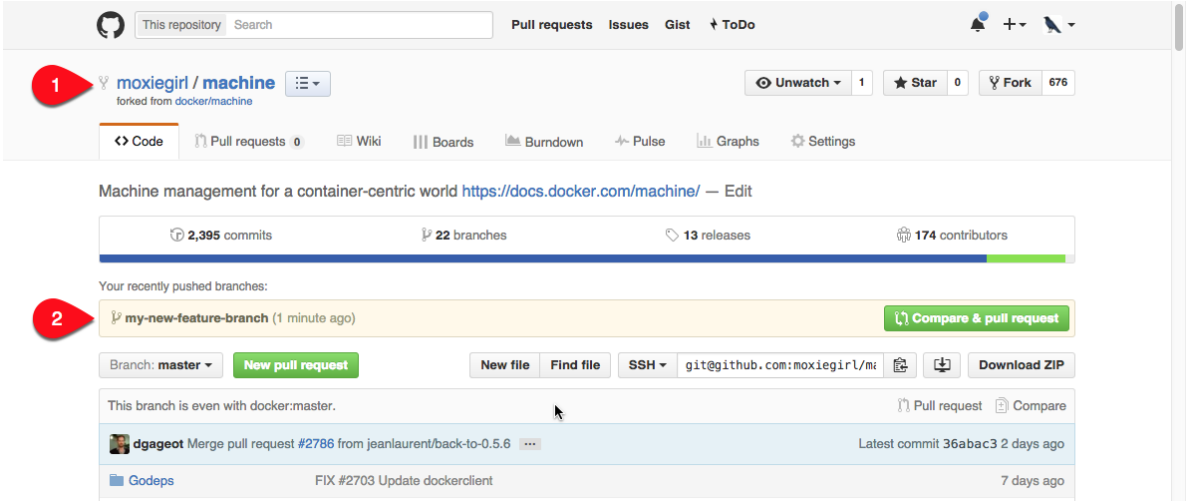
When to do this:	• You are ready to have your work reviewed by others. • You want to create a WIP pull request so people can see your changes as you work.
Prerequisites:	• Feature work done or in progress on a branch.

Click to enlarge:

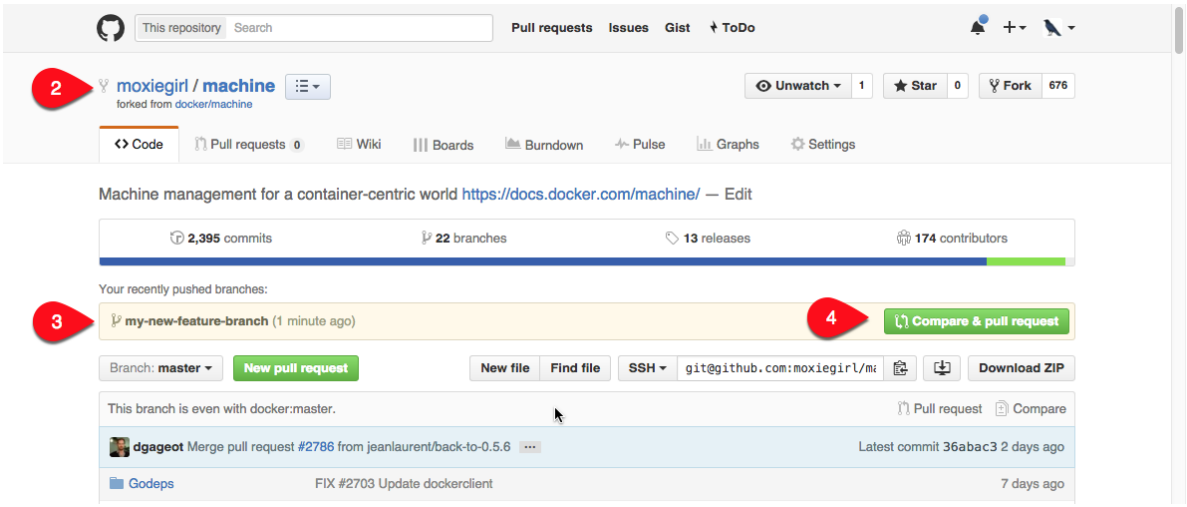


Introductory sentence

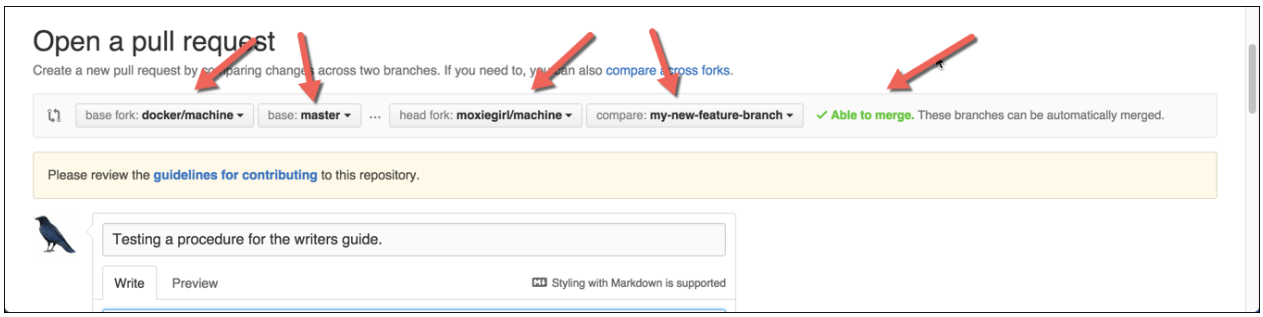
1. Rebase, squash, and push to your fork.
2. Go to your fork on GitHub.
3. Make sure your feature branch is pushed.



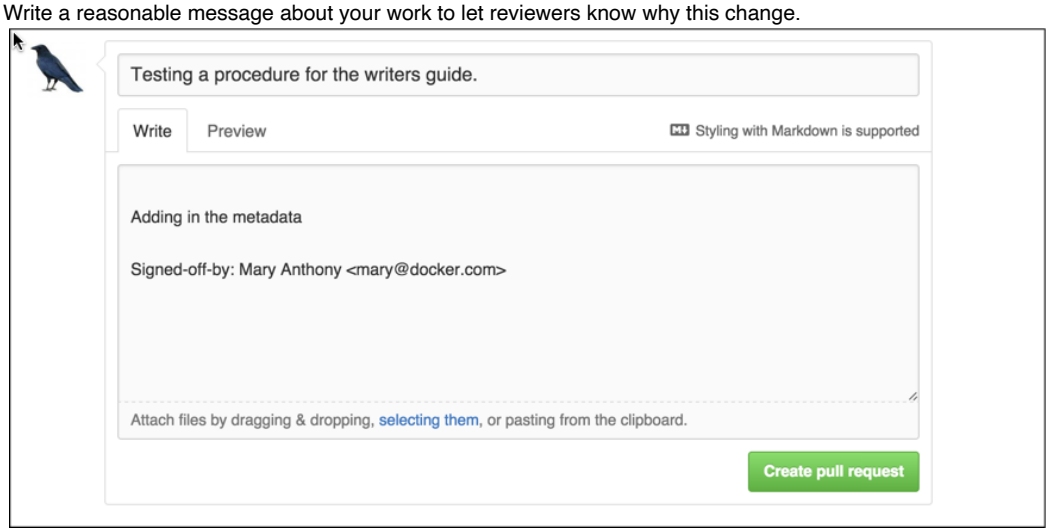
4. Press **Compare & pull request**.



5. Make sure the pull request is to the docker/repository master and from your/repository feature branch.

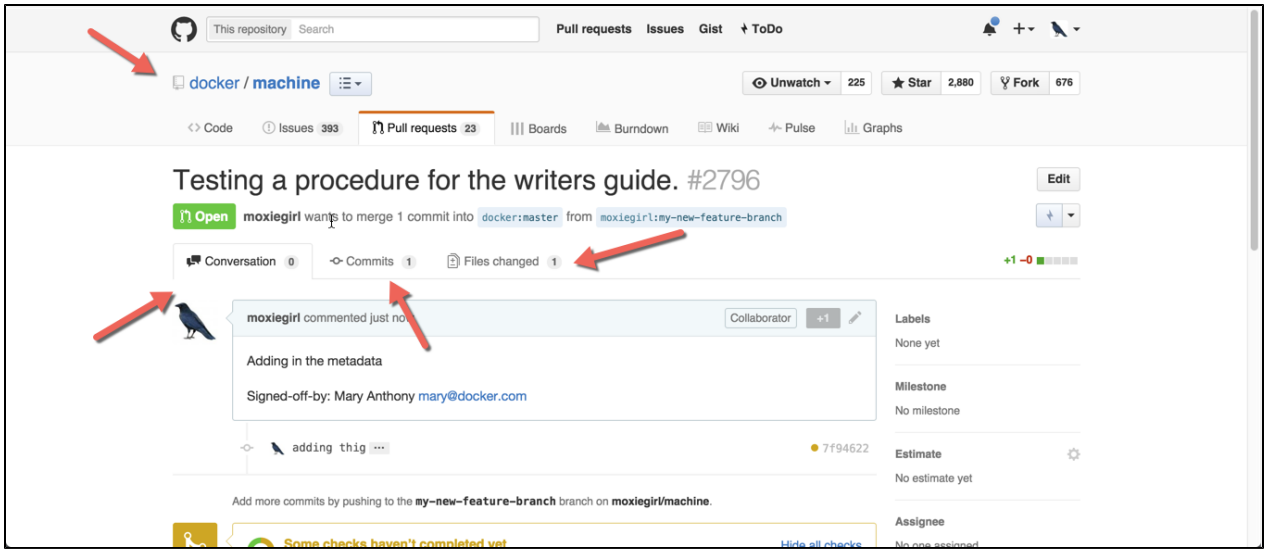


6. Write a reasonable message about your work to let reviewers know why this change.



7. Check the content of the pull request.
   The number of files, the names of the files, and the changes should all be checked.

8. Press **Create pull request**.
   GitHub creates the pull request on the original repository. Take some time to navigate among the tabs on the pull request.



Participate a pull request review

Completed work goes through a peer review. The mechanism for reviewing work in GitHub is a pull request. This page contains information specific to reviewing pull request for Docker repositories.

Make sure you read through or review the GitHub help for pull requests.

Terminology

Term	Definition
open source software (OSS)	OSS is computer software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose. Open-source software may be developed in a collaborative public manner.
maintainer	A person responsible for code in a Docker OSS GitHub repository. This person is approved by Docker and has permissions to merge PRs on our projects.
developer	A person responsible for code in a Docker commercial GitHub repository.
reviewer	OSS repositories are public, this means anyone can review a
LGTM	LGTM is an acronym that stands for **L**ooks **G**ood **T**o **M**e and, like it sounds, it means the person who wrote it approves of your work. People will sometimes give you a **–LGTM** which means I don't approve this.

Workflow for getting a pull request reviewed

1. From GitHub, create a pull request from your fork to a branch on the Docker-owned repository.
   Typically, the pull request will be from your fork's feature branch to the master branch on the Docker repository.
2. Make sure your pull requests references any related issues it is fixing or closing.
   There are a couple of ways to do this. You can add an issue number in a Git comment or you can add a link reference in the PR description.
3. Mention or ping the reviewers you want.
4. Change the label on the PR to "ready for a review."
5. Respond to the reviewers in a timely manner.
6. If you are asked to change something, you make the change in the same feature branch where the PR originated.
   There is a one-to-one relationship between feature branch and pull requests. You cannot have to PR which both originate with the same feature branch.
7. Commit and push your change as you normally would.
8. Then, rebase and squash to ensure your PR has a single commit.

9. Add a comment to the PR to notify the reviewers you have updated the work.
   This comment causes GitHub to notify your reviewers you've updated your work. If you forget this, the reviewers will not come back.
10. Ask for a review if you don't have enough LGTMs.
11. Merge when you have met all the requirements.

When is an LGTM not an LGTM?

People can qualify an LGTM for example:

```
LGTM with the changes noted in my review
```

If you get a qualified LGTM, then make the change and ask for another check from the commenter. Only then can you accept their LGTM as legit.

So, that means you must work with the review to address his or her concerns.

Requirements (rules) for merging

When you merge a pull request, you move code from your fork's feature branch into Docker's code base. There are the import rules for merging. You should always follow these rules. There are some projects and repositories where there are exceptions to these rules. This page explains both the rules and the acceptable exceptions.

Open source (public) merge rules

These apply to all Docker OSS projects:

- You must have at least two reviewers who are project maintainers give you an LGTM.
- If you are making a significant change, you must get a review from another doc-maintainer.
  - If you are moving or removing pages
  - If you are adding or removing more than 1 paragraph
- Do not merge your own PR: someone else must merge it for you.
- Do not merge if there is a broken Continuous Integration (CI) test on your PR.
  - Re-run the test to see if it clears
  - If the test continues to fail, get a comment a comment on the PR from a maintainer on the project which says:

    ```
    Good to merge with broken test; not a doc impact
    ```

Commercial source (private) merge rules

These apply to all Docke commercial projects:

- You must have at least one reviewer who is a developer give you an LGTM.
- If you are making a significant change, get an LGTM from a documentation team member.
- Do not merge if there is a broken Continuous Integration (CI) test on your PR.
  - Re-run the test to see if it clears
  - If the test continues to fail, get a comment on the PR from a developer on the project which says:

    ```
    Good to merge with broken test; not a doc impact
    ```

Exceptions in open source

Project	Exception
`docker/docker`	The project allows you to merge your own pull request provided all the other conditions are met.
`docker/docs-base`	• Mary reviews each of Sven's PRs • Sven or Seb reviews each of Mary's PRs • Mary and Sven reviews any other contribution

Exceptions in commercial source

- None at this time.

`docker/docs.docker.com`	• Mary reviews each of Sven's PRs • Sven or Seb reviews each of Mary's PRs • Mary and Sven reviews any other contribution

Test or carry another user's code branch

Sometimes, you need to build the code on another user's pull request. For example, if you want to check the document layout or see how the page flows. A pull request's code is accessible through the upstream remote. You can check it out like any other branch, provided you have configured your repository `.git/config` properly.

Configure the .git/config for the repository

1. Change to the root of your repository.
2. Edit the `.git/config` file in your favorite editor.
3. Add a fetch for pull requests:

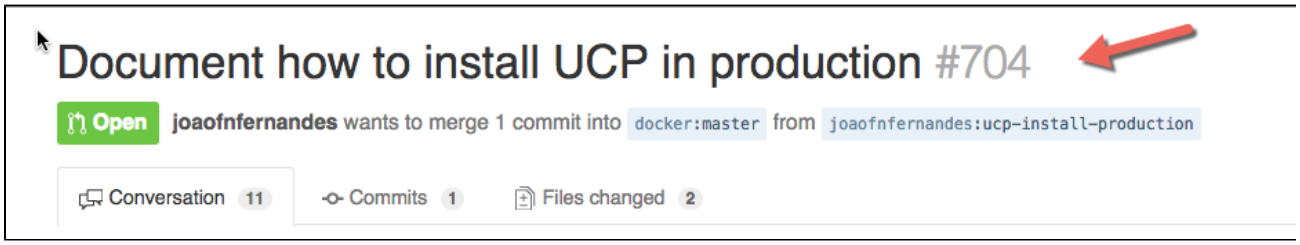
```
[remote "upstream"]
url = git@github.com:docker/orca.git
fetch = +refs/heads/*:refs/remotes/upstream/*
fetch = +refs/pull/*/head:refs/remotes/upstream/pull/*
pushurl = no_push
```

   Line 4 is what allows you to pull another user's code branch.
4. Save and close the file.
5. Do a `git fetch` to download the refs for all the pull requests to your local system.

```
$ git fetch upstream
Saving password to keychain failed
Identity added: /Users/victoriabialas/.ssh/id_rsa
(/Users/victoriabialas/.ssh/id_rsa)
remote: Counting objects: 4518, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 4518 (delta 2570), reused 2568 (delta 2568), pack-reused
1929
Receiving objects: 100% (4518/4518), 5.93 MiB | 3.32 MiB/s, done.
Resolving deltas: 100% (3114/3114), completed with 984 local objects.
From github.com:docker/compose
 * [new ref]         refs/pull/10/head -> upstream/pull/10
 * [new ref]         refs/pull/100/head -> upstream/pull/100
 * [new ref]         refs/pull/1001/head -> upstream/pull/1001
 * [new ref]         refs/pull/1002/head -> upstream/pull/1002
 * [new ref]         refs/pull/1005/head -> upstream/pull/1005
 * [new ref]         refs/pull/1006/head -> upstream/pull/1006
 * [new ref]         refs/pull/1007/head -> upstream/pull/1007
 ...
```

How to checkout another user's code branch

1. In GitHub, locate the number of the pull request you want to checkout.



2. Make sure you are in your repo.
3. Use the checkout command like this:

```
$ git checkout upstream/pull/704
Note: checking out 'upstream/pull/704'.

You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you make in
this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you
may
do so (now or later) by using -b with the checkout command again.
Example:

  git checkout -b <new-branch-name>

HEAD is now at 4cb5a79... Document how to install UCP in production
```

4. Read the message Git gives you.
   In this branch, you can make changes but you can't push back your changes. If you want to do that, branch off the branch as Git instructs.
5. Once you review how the code in the PR behaves, you can make comments about it on GitHub.

Carrying a PR on a new branch

You can ***carry*** a PR for another contributor or person. A carry allows you to make any needed code changes yourself. You carry a PR in these situations:

- the contributor has started but can't complete a change
- a contributor abandoned a change
- a contributor or another Docker maintainer asks you to in the course of your work

If you decide to carry, do the following:

1. Add a comment *on the existing PR* telling the requestor that you will carry.
2. Branch from the detached HEAD that you created with the upstream pull on the PR.

```
$ git checkout -b carry-pr-704
```

   This creates a new, standard, local branch that points to HEAD. From here, you can work with the files and branch as you normally would.
3. When you are ready to push, follow the same procedure as you typically do, described in Rebase, squash, and push to your fork, with these few exceptions:
   a. In the commit message, indicate that the commit "carries and closes #<PR>".

```
$ git commit -s -m "Carries and closes #3047"
```

The #mention is a GitHub feature. It causes GitHub to automatically close the original pull request when your new request is merged.

b. Provide the option to open a new PR with your latest changes, and carry along the original commits and conversation thr
c. When you do the interactive rebase (`$ git rebase -i upstream/master`), the commit messages will include those from the initiator of the original PR, along with their signature.
   Be sure to keep those in the message along with your messages and signature when you save and close the file.
d. After pushing the changes (`$ git push -f origin`), go to your fork and make sure that the changes show up there, then continue with Create a pull request. On the branch you were working in, you'll get the option to open a new PR with your latest changes, and carry along the original commits and conversation thread.

Understand when and how to merge a pull request

When you merge a pull request, you move code from your fork's feature branch into Docker's code base. There are the import rules for merging. You should always follow these rules. There are some projects and repositories where there are exceptions to these rules. This page explains both the rules and the acceptable exceptions.

Open source (public) merge rules

These apply to all Docker OSS projects:

- You must have at least two reviewers who are project maintainers give you an LGTM.
- If you are making a significant change, you must get a review from another doc-maintainer.
    - If you are moving or removing pages
    - If you are adding or removing more than 1 paragraph
- Do not merge your own PR: someone else must merge it for you.
- Do not merge if there is a broken Continuous Integration (CI) test on your PR.
    - Re-run the test to see if it clears
    - If the test continues to fail, get a comment a comment on the PR from a maintainer on the project which says:

      ```
      Good to merge with broken test; not a doc impact
      ```

Commercial source (private) merge rules

These apply to all Docke commercial projects:

- You must have at least one reviewer who is a developer give you an LGTM.
- If you are making a significant change, get an LGTM from a documentation team member.
- Do not merge if there is a broken Continuous Integration (CI) test on your PR.
    - Re-run the test to see if it clears
    - If the test continues to fail, get a comment on the PR from a developer on the project which says:

      ```
      Good to merge with broken test; not a doc impact
      ```

Exceptions in open source

Project	Exception
`docker/docker`	The project allows you to merge your own pull request provided all the other conditions are met.
`docker/docs-base`	• Mary reviews each of Sven's PRs • Sven or Seb reviews each of Mary's PRs • Mary and Sven reviews any other contribution

Exceptions in commercial source

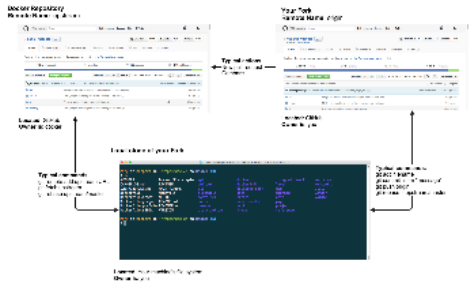
- None at this time.

`docker/docs.docker.com`	• Mary reviews each of Sven's PRs • Sven or Seb reviews each of Mary's PRs • Mary and Sven reviews any other contribution

Backing out changes you already pushed to GitHub

When to do this:	There may be times when you have files pushed to GitHub and you need to back some of the files out and keep others. This procedure removes the unwanted files.
Prerequisites:	A file already committed to your branch.

Click to enlarge:



1. Check out your branch and check your status.
   You don't want to have changes pending or staged.
2. List the commits on your branch that aren't on `upstream/master`.

```
$ git log upstream/master..my-new-feature-branch
```

3. Reset the branch to the last committed change.

```
$ git reset --soft HEAD^
```

   The single ^ caret removes the last commit. If you had three commits in this branch, you could have typed: `git reset --soft HEAD ^^^` If you had five commits you could have used five carrots or: `git reset --soft HEAD ~5`
4. Do another git status.
   You should see all the changes in your branch.

```
carolfager-higgins at merlot in ~/repos/dhe-deploy/docs on post-dtr142
$ git reset --soft HEAD^

carolfager-higgins at merlot in ~/repos/dhe-deploy/docs on post-dtr142 [+]
$ git status
On branch post-dtr142
Your branch is behind 'origin/post-dtr142' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   assets/gc1.png
        modified:   assets/gc3.png
        modified:   install/install-csengine.md
        modified:   install/upgrade.md
        modified:   soft-garbage.md
```

   All your files are now uncommitted. You can unstage them.
5. Once you are satisfied, `git add` the files you want to commit.
6. Push your changes to your fork on GitHub.

7. Go to GitHub and verify your changes.

How to cherry-pick a commit for release

Periodically, the docs build engineer updates docs.docker.com between official releases of Docker. They do this by:

- cherry-picking commits from a `master` branch
- merging them into the `docs` branch
- publishing the result from the `docs` branch

Create a cherry-pick based off of the upstream/docs branch

1. Go to your `docker/docker` fork and get the latest from master.

```
$ git fetch upstream
```

2. Checkout a new branch based on `upstream/docs`.
   You should give your new branch a descriptive name.

```
$ git checkout -b post-1.2.0-docs-update-1 upstream/docs
```

Find the commits you want to cherry pick

Cherry-pick commits that have ***only .md files*** in them. These are documentation changes. If you cherry-pick commits with code files in them, chances are you are cherry picking functional product changes intended for a future release.

1. In a browser window, open [https://github.com/docker/docker/commits/master].
2. Locate the merges you want to publish.
   You should only cherry-pick individual commits; do not cherry-pick merge commits. To minimize merge conflicts, start with the oldest commit and work your way forward in time.
3. Copy the commit SHA from GitHub.
4. Cherry-pick the commit.

```
$ git cherry-pick -x fe845c4
```

5. Repeat until you have cherry-picked everything you want to merge.
6. Push your changes to your fork.

```
$ git push origin post-1.2.0-docs-update-1
```

Git Going Cheesheet

This page has some quick reference commands for using Git with the Writer workflow. They are categorized by specific activities. Where there is a detailed procedure. That is noted.

Set Global Git Configuration

Typically, you only do this one per machine.

Command	Notes
`git config --global user.name "FirstName LastName"`	You need to do this before creating a pull request.
`git config --global user.email "emailname@mycompany.com"`	
`git config --global core.editor "atom --wait"`	This assumes Atom is your editor. You might want to use another editor so change the command accordingly.

Getting a repo to your local machine

See Create a repository clone on your local machine for the full procedure.

Command or Action	Notes
Fork the original repo on GitHub	Copies the repo to your account.
`git clone url-to-your-forked-repo`	Clones the repo to your local machine. Usually you should use the SSH protocol.
`git remote add upstream url-to-the-original-repo`	Do this from the within your fork directory. This sets as upstream the Docker repository you forked.
`git remote set-url --push REMOTE_NAME no_push`	This command prevents the accidental push to a repository.

Commonly used commands

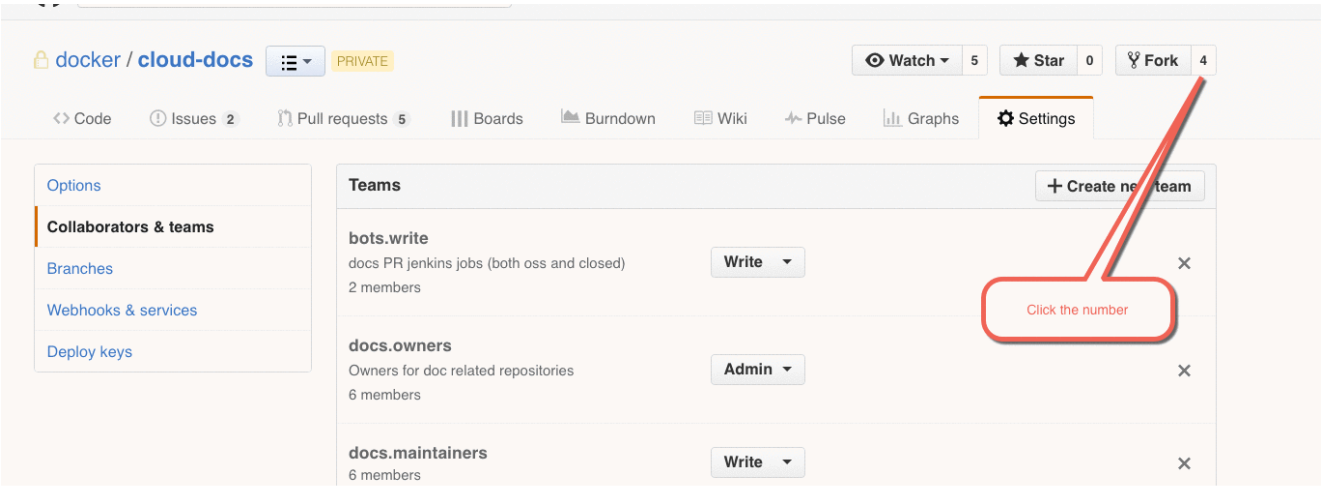
Most of these commands you use every day.

Command	Notes
`git status`	Used on the file system within a repo. Informs you about the current status.
`git checkout BRANCH_NAME`	Checks out an existing branch --- basically switches you to the branch
`git fetch REMOTE_NAME`	Fetches the latest references (committed changes) from a remote. If you use the , called upstream remote --- if you are following along with this page, that is the original repo
`git rebase REMOTE_NAME/BRANCH_NAME`	Does a fast-forward merge. Dangerous if you don't know what you are doing. See Daily update and rebase the master branch on your local repository and Rebase, squash, and push to your fork.
`git add filename`	Stages a file for commit. See Add a new file or folder to Git tracking
`git commit -s -m "Message for commit"`	Commits a change. See Add a new file or folder to Git tracking
`git push REMOTE_NAME`	Pushes changes to a remote repository. See Add a new file or folder to Git tracking

Undoing Git things

Command	Notes
`git checkout BRANCH_NAME — FILE_PATH`	Checkout a file from specific branch. For example, if you want to back out a file change you made in a feature branch, you can checkout the file from the `upstream/master` branch. `git checkout upstream/master -- docs/multi-manager-setup.md`
`git checkout -- FILE_NAME`	Discards a file that you modified but did not commit yet, and reverts it to its state before you made the changes. Examples:
`git reset --hard HEAD~2`	Removes the last two commits. Increase the number to remove even more commits.
`git reset HEAD^`	"Uncommits" the commits (removes them from the branch and the index), but retains the changes in the working tree for re-working.
`git branch -b BRANCH_NAME`	Run this in an existing feature branch when you want to save current changes to a different branch.

How to get to another users' fork when you don't have the URL



Other helpful commands

Command	Notes
`git config --global core.excludesfile 'file-type-or-path'`	Prevents Git from including files or folders in a commit.

`git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)%Creset'`
 	
`--abbrev-commit --date=relative --since=2015-08-11 branchname..master docs`
	Example of a fancy comparison.

Non-Git commands you need to run in the course of your work

Command	Notes

`ssh-add ~/.ssh/filename`	Adds an identity to the ssh-agent on your Mac. See Correct a Permission Denied (publickey) error
`docker rm $(docker ps -a -q)`	Removes all the containers.
`docker rmi -f $(docker images -q -a -f dangling=true)`	This will remove untagged images, that are the leaves of the images tree (not intermediary layers). If you have a container that is using the image it is not removed.
`docker rmi -f $(docker images -q -a )`	Forcefully removes all images.
`docker-machine create -d virtualbox --virtualbox-disk-size "20000" default`	Creates machine called `default` with the size of 20G. Useful if you run out of space on your machine.

FAQ about writers and Git

Why not just use the Edit command on a file in GitHub?

The edit command doesn't allow a user to enter a signature that our CI system can recognize. It also doesn't allow the contributor to easily update the change.

Why don't the writers just use X simple GUI for Git or do Y instead of X with Git?

Writers use the Git command line and use the same, identical workflows because:

- Writers use the same process and tools we teach our OSS contributors: this allows writers to help contributors on the OSS project
- GUI tools or other tools are just one more thing to learn.
- If each writer chooses a tool they like, then whoever is helping them is stuck learning ***N*** number of tools. Not practical.
- If writers use the Git command line and no other writer is around to help them, then an engineer is likely to know Git and can help.
- Using the command line forces writers into simple developer workflow. It is the workflow our contributors use. Adding to a writers knowledge in this way makes them more effective in working with their audience.
- Git is very flexible, there are a millions of ways to do each thing. There are millions of situations they can get into. If writers stick to the same proven workflow, particularly when learning Git/GitHub, they are less likely to get into trouble.

Troubleshooting Git

Correct a Permission Denied (publickey) error

If you get a public key error interacting with a remote repository. Make sure you add your public key to your SSH agent.

```
$ git fetch upstream
Permission denied (publickey).
fatal: Could not read from remote repository.
Please make sure you have the correct access rights
and the repository exists.
$ ssh-add ~/.ssh/mma-docker
Enter passphrase for /Users/mary/.ssh/mma-docker:
Identity added: /Users/mary/.ssh/mma-docker (/Users/mary/.ssh/mma-docker)
```

Correct a TLS-enabled daemon error

TLS-enabled daemon error

```
$ make docs
docker build -t "docs-base:fixes-15790" .
Post
http:///var/run/docker.sock/v1.20/build?cgroupparent=&cpuperiod=0&cpuquota
=0&cpusetcpus=&cpusetmems=&cpu<snip>: no such file or directory.
* Are you trying to connect to a TLS-enabled daemon without TLS?
* Is your docker daemon up and running?
make: *** [docs-build] Error 1
```

1. Check and see if your machine is running.

```
$ docker-machine ls
NAME      ACTIVE   DRIVER
default     *      virtualbox
```

2. If it is running, get the environment config for your machine.

```
$ docker-machine env default
```

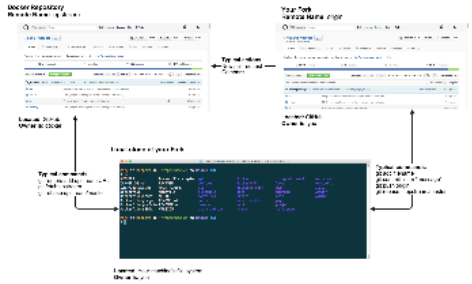
3. Set the environment config for your machine.

```
$ eval "$(docker-machine env default)"
```

4. Try the `make docs` command again.

Checkout a remote branch from GitHub

When to do this:	You might want to do this if you accidentally remove or destroy your local repository.
Prerequisites:	A clone of your fork.



The branches on your GitHub fork are remote from your local repository. When you clone a fork, the `clone` operation has the references for all the remote branches, but only creates a `master` branch locally. To list all the branches Git knows about both local and remote, use the `git branch -la` command. The output shows the branches that are local and the remote ones.

```
$ git branch -la
* fix-904
  fix-release-notes
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/add-engine-discovery
  remotes/origin/block-out-docs
  remotes/origin/carry-443
  remotes/origin/deploy-app
  remotes/origin/docs-beta-7
```

To checkout a remote branch, select the branch you want to checkout and then use the `-t` flag when you check it out. For example:

```
git checkout -t origin/branch-name
```

Git checks out and creates a local branch that has the name as the remote and that *tracks* the remote branch at `origin/branch-name`.

You can use `git branch` with the `-t` (`–track`) flag to create the branch without switching to it.

Guidance and Style

Page construction and format

Metadata

Each page starts with metadata in TOML format. Hugo calls this area Frontmatter.

```
 <!--[metadata]>
+++
draft=true|false
title = "Title as it appears on a menu"
description = "Description of page"
keywords = ["appear as metatags in a generated HTML document"]
[menu.main]
identifier="Page identifier"
parent = "Parent page identifier"
weight = "negative or positive integer value"
+++
<![end-metadata]-->
```

Lines	
1 and 12	Comments. They are there to hide the metadata when the page is displayed in GitHub. GitHub displays pages but this is display, not a web; don't confuse it with one. Even so, developers like the page display to "work" in GitHub so we facilitate for that if we can.
2 and 11	TOML format designators.
3	This is an optional component. When true, the page is not generated for the web. The default false ensures it is.
7 thru 10	• Metadata • Block TBD and TODO in Comments • Code and command line examples Specifies where and in which in the menu this page appears. If you don't specify an identifier, the page uses the title as the identifier. This can cause problems, say between the `docker help` command the `docker-machine help`. For reference material, include an identifier to prevent name collisions such as: `identifier="eng_info"` You can have multiple menu configurations; which means you can place the page in any menu in our system. The `parent` value says which menu this page hangs off of. No parent? You have a top level page.
10	A word about the `weight` value. This sorts a page *within* a menu. If you don't specify a weight, the menu sorts alphabetically. Negative weighted pages are placed higher than positive. Try to add this value infrequently. If you have to add, use blocks of 10 you can always add pages easily. If you start with single digits, you have to reshuffle.

Block TBD and TODO in Comments

If you want to leave a TBD, TODO, image placeholder, or other production comment in a in a page, put it in comment blocks.

```
<!--[metadata]>
Need an image here.
<![end-metadata]-->
```

Code and command line examples

Code can appear in three kinds places,

- as part of sentence to indicate a file or command name
- within the body of the page text
- as part of ordered or unordered list

Within the documentation generation system, we use Highlight JS to create syntax highlighting in our docs. Our system supports highlighting for dozens of languages (and not-really-languages, like diffs and HTTP headers); to see the complete list, and how to write the language names, see the highlight.js demo page.

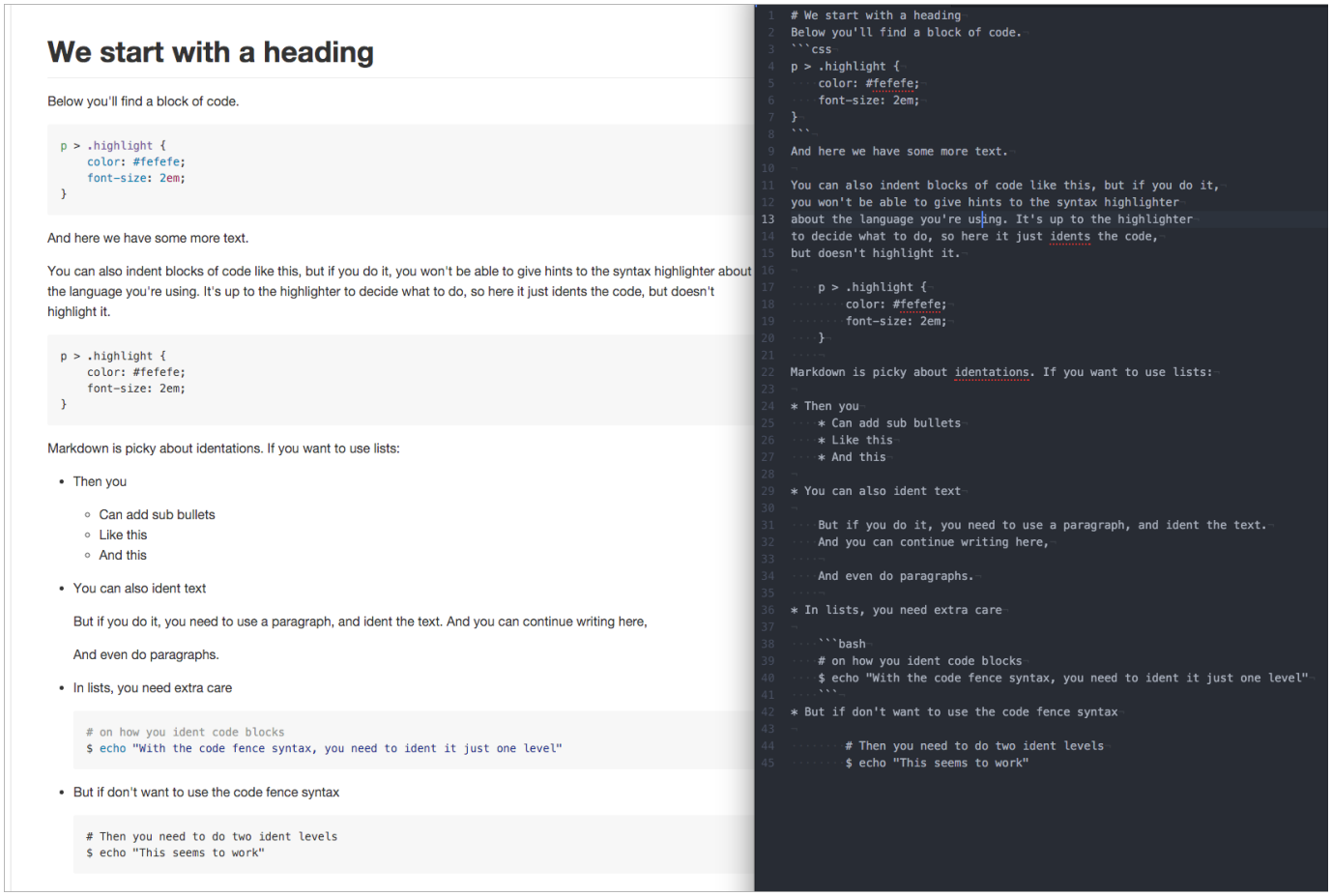
Inline `code` has `` `back-ticks around` `` it.

Within the body of the page text, fence code blocks with three back-ticks ``` and then follow the tics with a language indicator.

```
```css
@font-face {
  font-family: Chunkfive; src: url('Chunkfive.otf');
}
```
Or this:

```javascript
var s = "JavaScript syntax highlighting";
alert(s);
```
```

To get code blocks right, you need to be careful with the indentation you're using. Check this sample to learn more.



Style Guide

We pick an industry style guide to follow and document only those things that are contrary to the book.

Internal Style Guide

Read Me First! A Style Guide for the Computer Industry
Third Edition (3rd Edition)

by Sun Technical Publications
Link: http://amzn.com/0137058268

Terminology

This page describes how to use special Docker terms in your writing.

Docker alone

Docker refers to the family of OSS projects and commercial products which includes Engine, Machine, Swarm, Docker Hub, Registry etc.

Docker resource and or objects

Some things are objects or resources used within the Docker ecosystem. In these cases, where we are trying to "own" a branded flavor of these resources of objects, the product designator should be dropped.

- Docker container
- Docker volume
- Docker container network
- Docker image

Docker Engine

Docker Engine is the core product which provides image and container functionality.

First use on a page within text, not menu titles.

- Docker Engine
- Docker Engine daemon

- Docker Engine CLI
- a Docker Engine client
- Docker Engine host
- Docker Engine Remote API

Subsequent references in text

- Engine daemon
- Engine CLI
- Engine CLI commands
- an Engine command
- an Engine client
- Engine Remote API
- Engine host
- Engine

Docker Machine

> *Victoria Bialas maybe you want to expand Machine terms here?*

First use on a page within text, not menu navigation.

- Docker Machine
- Docker Machine CLI
- host (***machine***) - we just need to clarify what lower-case "machine" is (I've done so in latest draft of "overview")
- Dockerized host (not specific to Machine topics, though)

Subsequent references in text

- Machine
- Machine CLI

Docker Compose

First use on a page within text, not menu navigation.

- Docker Compose
- Docker Compose CLI

Subsequent references in text

- Compose
- Compose CLI

Docker Swarm

First use on a page within text, not menu navigation.

- Docker Swarm
- Docker Swarm CLI

Subsequent references in text

- Swarm
- Swarm CLI

Related terms:

- Swarm cluster
- cluster
- high availability (HA) first use
- HA subsequent use
- *node* is a system belonging to a Swarm cluster. This system (VM or iron) is running Docker Engine
- *manager* - a node running the Swarm manager container
- *primary manage*r - currently active manager in a cluster with multiple managers
- *secondary manager* - a node capable of replacing the primary manager should it fail

Docker Notary

Notary is a component project that stands alone. It is also embedded into some of our products to provide trust to our image content.

Embedded in product

First use on a page within text, not menu navigation.

- Docker content trust

Subsequent references in text

- content trust

Component Projects (Notary and Registry)

First use on a page within text, not menu navigation.

- Docker Notary
- Docker Notary CLI
- Docker Notary API

Subsequent references in text

- Notary
- Notary CLI

Docker Trusted Registry

First use on a page within text, not menu navigation.

- Docker Trusted Registry

Subsequent references in text

- Trusted Registry

Docker Commercially Supported Engine

First use on a page within text, not menu navigation.

- Docker Commercially Supported Engine

Subsequent references in text

- CS Engine

Universal Control Plane (UCP)

First use on a page within text, not menu navigation.

- Docker Universal Control Plane

Subsequent references in text

- UCP

Related terms:

- See the Docker Swarm section above.

- UCP cluster
- cluster
- high availability (HA) first use
- HA subsequent use
- *node* is a system belonging to a UCP installation and the underlying Swarm cluster. This system (VM or iron) is running Docker Engine
- *controller* - a node running the UCP controller processes
- *primary controller* - currently active controller in a cluster with multiple controllers
- *replica* - a node capable of replacing the primary controller should it fail

- `ucp` tool
- `ucp` tool's `XXX` subcommand such as `engine-discovery`, `install`, etc.
- the "hamburger" menu pop-out — menu

Guidelines for screenshots and illustrations

WIP COMMENT

THIS IS A DRAFT PLEASE COMMENT AWAY

When to do this:	Refer to these guidelines when using screenshots and illustrations in your documentation. The goal of this page is to help you create meaningful graphics that quickly convey important information.
Prerequisites:	Snagit and Gliffy

Good graphics are critical

Did you know that good graphics could save lives? If you are not familiar with the Space Shuttle Challenger disaster that occurred on January 28, 1986, you can read about it here: https://en.wikipedia.org/wiki/Space_Shuttle_Challenger_disaster#Thiokol-NASA_conference_call. Suffice to say that if the contractor's charts were viewed, NASA would have realized that the failure rate of the O-rings increased exponentially in cold weather and they would have delayed that fatal launch which killed the crew.

General notes

Whether you are creating screenshots or graphics, let common sense be your guide. The goal is to display information graphically so that users can quickly understand what they are reading about. While you may choose to use http://resizemybrowser.com/ to resize your screen, it's not necessary if you use your laptop to take the screenshot. Currently, most of our docs are read on a laptop. If we begin to use our phones to view doc information, then we will have to revisit this.

Since Docker is a relatively young company, it makes sense to talk about *branding*. These guidelines will assist the doc team in creating screenshots and graphics that are similar in style and tone to the Docker web presence.

Introduce your screenshot or graphic before inserting it with a sentence or two. This gives readers a head's up of what they are looking at or what is important for them to focus on.

Setting up your tools

Whether you need to take screenshots or create diagrams, you'll first need to set up your tools.

Colors

To help you in choosing colors, here are the web safe ones to use (some of these are *not* not final):

- Black (standard) #333333 http://www.color-hex.com/color/333333
- **Screenshot borders** gray78 #c7c7c7
- blue #22b8eb This is the main color in case you were wondering and didn't use the free cool tool CSS Viewer.
- **Arrows**: Orange #FF66900
- **Callouts**: Orange #FF66900 or Tangerine FF9900 (Either color works depending on context
- **Text** in Callouts: Black (standard) #333333
- **Callouts with numbers**: Tangerine FF9900 (Currently I like the Snagit default of using circles since they can be placed anywhere and circles reinforce the wavy look.)

Screen Resolution (or getting the best screenshot possible)

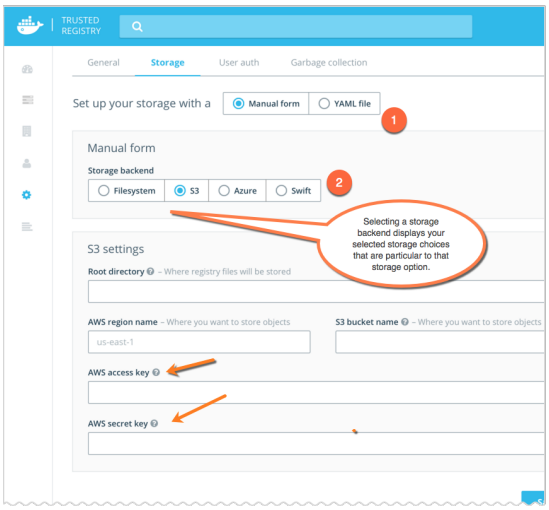
Assume that our audience is reading our documentation on a laptop. So, when taking a screenshot, it makes sense to create it from a laptop screen to ensure readers will not have to scroll to see it in its entirety.

Apple Retina display

Does Apple Retina display matter? Docker writers are using MacBooks with Retina display which Apple defines as screens that have a higher pixel density than their previous models. The goal of Retina displays is to make the display of text and images extremely crisp, so pixels are not visible to the naked eye. All 15-Inch Retina Display MacBook Pro models have a 15.4" color display with **2880x1800** native resolution at **220 ppi** and all 13-Inch Retina Display MacBook Pro models (A1425, A1502) have a 13.3" color display with **2560x1600** native resolution at **227 ppi**. Since Docker writers are using MacBooks, let's assume that all of us are using Retina display. This means that the resolution will be higher than non retina and might appear smaller if viewed on an older laptop that does not have this feature. The docs have been tested on a newer HP laptop and there was no discernible difference in viewing screenshots.

Screenshots

After gathering comments based on samples, this style is the winner:



Notice the thin gray border on three sides, and the wave on the bottom which mimics the wave at the end of our doc pages and reinforces that design element. People liked this look best as they preferred a minimal border with no distractions to the doc or screenshot. They understood that the edges of the screenshot were cut off and felt they didn't need an edge such as a tear to let them know this. Notice the various callouts: arrow, callouts, and numbers. They appear best with a slight shadow which adds weight and makes it appear that the elements are on *top* of the doc, and not a part of it. If you are going to use multiple elements, it also makes sense to create them using the same color.

Use Snagit to achieve this look.

- The Effects Border is gray78 #c7c7c7 and size (thickness) is 2pts.
- The Effects Edges is Wave at size 5 pts only on the bottom.
- You can save your settings as a style to quickly apply them.

Other considerations:

- First resize your screen so that while there is still white space, it is minimized without the screen looking weird.
- Leave enough space for the docker logo on the left.
- When editing your screenshot, first crop out non-essential information such as your bookmarks and URL.
- No need to take a large screen shot to illustrate your point. It's better to have additional screenshots or use the torn page tool.
- After applying the border effects, think of using other tools such as arrows or callouts with text to emphasize your point. To keep in the style of the screenshot, use arrows/callouts in a contrasting color such as red or orange and minimal borders as make sense to the screenshot. Shadows can be optional depending on what else is in the screenshot.
- If you are cropping a screenshot on all four sides, ensure that there is some context so users understand what you are trying to show.

Graphics or charts

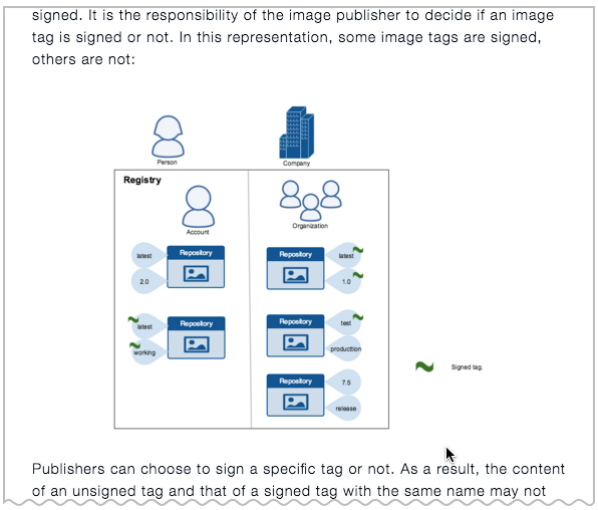
It doesn't matter which tool you use, it can be Snagit or Confluence's Gliffy. If you choose Gliffy, here's the documentation to get you started: https://www.gliffy.com/user-manual/?productId=1. You can also type in the bracket "[" while in edit mode to bring up a list.

When creating graphics, the current preference is to make your graphics:

- Borderless, as it's less distracting.

- In "soft" colors that compliment the UI when possible.
- With text, when it's used in a sans serif font for easy reading.
- Use stronger (bolder) colors to draw a readers eye to a specific area.
- Ensure to label your parts.
- Use a legend when there are many parts.

Here is an example:



I have included the text in this example to show you the actual size of the graphic.

Work to be done

- Address websafe colors - **IN PROGRESS**
- Include info for cutouts, callouts, and numbered bullets - **IN PROGRESS**
- guidelines for legends
- definitive font guidelines for text?
- match new doc UI for branding purposes

Other Books, Blogs, and Bibliostuff

It is really smart to have a subscription to Safari Books Online. Save a tree and invest in yourself at the same time.

Really good books to have (add your own)

Fast Reference	Information design	Best book on writing	
			

The Handbook of Technical Writing by Gerald J. Alred et al. Link: http://amzn.com/1457675528	How to Make Sense of Any Mess: Information Architecture for Everybody by Abby Covert Link: http://amzn.com/1500615994	The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century by Steven Pinker Link: http://amzn.com/0143127799	Minimalism Beyond the Nurnberg Funnel (Technical Communication, Multimedia, and Information Systems) by John M. Carroll Link: http://amzn.com/026203249X

Other resources blogs and such

- http://worrydream.com/
- http://alistapart.com/
- https://m.signalvnoise.com/
- http://99u.com/
- http://www.measuringu.com/

Tips, Tricks, and Tools

Convert Markdown to PDF

- Grip Tool

Grip Tool

One quick way to print Markdown page in a nice PDF is a tool called grip. It is super fast and captures images well.

Work with docs continuous integration (CI) tests

To retest a failed documentation build:

Comment on the PR, and say "**test this please**".
There is a Jenkins plugin that sees that comment and retests your build, and then updates the Jenkins build status. See:https://wiki.jenkins-ci.org/display/JENKINS/GitHub+pull+request+builder+plugin

Jenkins password for private repositories

Username	Password
`docs-team`	`cDaxUBLGuLm8hjAJ`

DOC Strategy Projects

- Improve documentation usability
- Revise documentation release process
- Validate documentation files on commit

Improve documentation usability

Our current documentation skin was derived from the www.docker.com site as part of a push for Dockercon. It didn't go through the design iteration we wanted it to. This project is designed to address that issue among others. In particular, the goals of this project are to:

- Have a full design iteration on the docs skin
- Support product-specific layouts
- Improve the fonts and general layout of content
- Ensure support multiple media formats (Computer, phone, tablet)
- Ensure the site is usability tested

Tasks addressing this situation

Key	Summary	T	Created	Updated	Due	Assignee	Reporter	P	Status	Resolution

No issues found

Revise documentation release process

Our current release process relies on releasing all the documentation at once to an S3 bucket. This is insufficient for the number of projects and the frequency of their release.

This project is being tracked in **DOCS-237** - Automated Documentation publishing TODO

The release Timeline is set by Core Release 1.10

docs.docker.com User Stories

One of strategic goal of the documentation team is to improve the documentation release process**.** We want to support release more frequently and at a more granular level. We have two initiatives in place already to support and prepare for this, the Validate documentation files on commit and Improve documentation usability. Given the number of projects and the variety of products, we want to make sure we have a good understanding of the requirements for publishing or releasing the documentation.

PURPOSE: Collect user stories related to constructing documentation content, building the content locally, and/or publishing documentation.

GOALS: The following

- Build a list of requirements for Docker's documentation release process.
- Add to or expand requirements of other strategic projects if necessary

A **user story** is a tool used in Agile software development to capture a description of a software feature from an end-**user** perspective. The **user story** describes the type of **user**, what they want and why. A **user story** helps to create a simplified description of a requirement.

What is user story? - Definition from WhatIs.com
searchsoftwarequality.techtarget.com/definition/**user-story**

Feedback

Open Source Documentation Contributor

Click for a user description...

An open source contributor, often a first time contributor. Likely to be unfamiliar with or new to Docker.

- [x] I want to see a change I made published to docs.docker.com within a day of when it is merged
- [x] I want to use an editor I am familiar with to write documentation.
- [x] I don't want to spend a lot of time learning a new tool so I can write documentation.
- [x] I don't want to create a new login to create documentation.
- [x] I want to write plain GitHub markdown so that I can use GiHub docs like a web
- [x] I want to get may changes into a Docker repository with a pull request
- [x] I want to build the documentation on my machine so I can test what I write.
- [x] I want to be able to do all the docs processes on my Windows machine.
- [x] I want to be able to do all the docs process on my Mac.

Docker Technical Writer

Click for a user description...

An employee or contractor to Docker. Experienced writing documentation for software companies. Used to an automated authoring environment — the equivalent of a developer's IDE.

- [x] I want to be sure my changes don't break the documentation build
- [x] I want a global variable that I can put in which stands for the current release version. For example, if I am writing a file in the compose project, I should be able to put in this variable. Then, at build time, the build system should automatically replace this variable for me to the latest version.
- [x] I should be able to define my own global variables within a project.
- [x] I should be able to define global variables in one project and use them in another.
- [x] I should be able to use a global variable any format that the authoring platform supports. For example, I might want to use a global variable in a code example.
- [x] I want to rename or move a file in a directory structure and redirect the old file redirect.
- [x] I want to be able to run a local copy of the docs server and see my changes reflected locally by refreshing the page (must work on OSX/Windows)
- [x] I want to be able to write some documentation source in one file and include it into another documentation source file (includes). I'd like to do this within my project or across projects.
- [x] I want to be able to be able to define a set of conditions that I can use in the source files (conditional text). I'll use these conditions to specify under what conditions to publish different parts of the source file.

Docker Engineer or Engineering Team Member

Click for a user description...

An employee or contractor to Docker. Doesn't have a lot of time to learn the ins and out of Markdown. Unfamiliar with Hugo static file generator. Inexperienced writing docs for enterprise software. Unfamiliar with an automated authoring environment.

- [x] I want to be able to publish our product documentation
- [x] I want to know that the latest documentation is correct - or at least know when code changes have affected examples.
- [x] I want to be able to include the output of a command (ex: `docker-compose --help`) in the docs in an automated fashion
- [x] I want to be able to generate API documentation for display on docs.docker.com
- [x] I want the man pages and command line documentation to be kept in sync, preferably by coming from the same source files
- [x] I want man pages to be an automatic part of all projects with a cmdline
- [x] I want to be able to review documentation changes the same way I review code (github workflow)
- [x] I want our docs tools to play well with existing doc standards (godoc, github) and to not push breaking requirements on code (golint)
- [x] I want to be able to ship docs for the correct version of my product with the product itself

docs.docker.com Reader

Click for a user description...

Anyone reading the documentation. This person is trying to answer a question or solve a problem. Quickly finding information is important. This person also wants metadata about the documentation to answer questions like "Am I reading the right thing".

- [x] I want to be able to search the documentation from a search bar in the documentation site
- [x] I should be able to tell what version of the product the documentation I am reading is written for.
- [x] I want to be able to tell the last time a page was edited
- [x] I want to be able to easily tell someone about, annotation or update existing pages, without needing to login or register for an account, learn git, or deal with the strange signing process.
- [x] I want to look at content from older releases of a product.
- [x] I want to see quickly what's new in the current release (or see in which release something was added)
- [x] I want to see documentation for just the product I'm using (e.g. only docker-compose, swarm, engine, docker-machine)
- [x] I want to be able to download an offline copy of the documentation (in epub, PDF format)
- [x] I want to be able to read the documentation on a website
- [x] I want code examples to be better readable, without scrolling
- [x] I want to be able to copy / paste examples

Documentation Release Engineer

Click for a user description...

This person is responsible for building the tools for publishing the documentation and also for publishing the documentation.

- [x] I want to publish a new version for just a single product
- [x] I want to publish an update to a single page in an existing version
- [x] I want to archive an existing version of the product
- [x] I want to generate PDF of a set of product documentation
- [x] I should be able to rollback to a particular version of the documentation for any product.
- [x] I should be able to rollback to a particular version of a page.
- [x] I want to be able to authorize users to publish documentation for a project or as a whole
- [x] I want to be able to lock users from publishing per project or as a whole

Technical Documentation Manager/Docker Manager

Click for a user description...

This person is responsible for smooth operation and delivery of the documentation set. This person wants data about the documentation including how it was delivered and how it is being used.

- [x] Keep an audit log of what was published, who published it, and when
- [x] Google Analytics should be imbedded in each document source page
- [x] I want to be able to provide writers and contributors with documentation on how to create source content
- [x] I want to be able to provide user with instructions on how to publish

Documentation Release Tooling Requirements

Target release	1.10
Epic	**DOCS-237** - Automated Documentation publishing TODO
Document status	**DRAFT**
Document owner	Mary Anthony
Designer	
Developers	Sven Dowideit
QA	

Goals

- Support multi-architecture documentation
- Support per-product release cycles

Background and strategic fit

We expect to provide ...tBD

Assumptions

Requirements

#	Title	User Story	Importance	Notes
1	GitHub Support	• I want to write plain GitHub markdown so that I can use GiHub docs like a web • I want our docs tools to play well with existing doc standards (godoc, github) and to not push breaking requirements on code (golint)		• Feature implementations should avoid breaking linking between pages in a GitHub project • Feature implementations should support display of images within a GitHub project • Feature implementations should not surface into the GitHub visual representation of a page
2	Changes reflected on site	• I want to see a change I made published to docs.docker.com withi n a day of when it is merged because I like to see the impact of my contribution. When I do, I feel like I made a difference and it inspires me to contribute more.		• This requires a better cherry pick feature; right now cherry-picks are manual • Since the cherry pick feature increases the risk of bad changes being pushed, DEPENDS ON a rollback function
3	Validation support	• I the documentation build to report errors present in my source that break the documentation build. For instance links that don't resolve, markdown that isn't formatted properly, images that are missing, and other things. An automated provides confirmation of a manual check and also allows me as a writer to work more efficiently.		• CI Validation should run for local builds pull request and fail on error • CI Validation should run for each pull request and fail on error
4	Global Variable	• I should be able to define my own global variables within a project and use them in place of key terms. Good candidates for global variables are product or feature names, operating system, product versions, and current year. Using Global variables allows me to quickly upgrade a document for a particular product version or operating system. For example, if I am writing a file in the compose project, I should be able to put in this variable. Then, at build time, the build system should automatically replace this variable for me to the latest version. • I should be able to define docker-wide variables and use them across projects. • I should be able to use a global variable any format that the authoring platform supports. For example, I might want to use a global variable in a code example.		• This would be a key-value pair • The variable should be usable in any kind of paragraph format (code, heading) or font format (bold, italics, preformatted) • We should support cross-project global variables; maybe by means of a build variables file
5	Authoring Environment	• I want to use an editor I am familiar with to write documentation. • I don't want to spend a lot of time learning a new tool so I can write documentation. • I don't want to create a new login to create documentation. • I want to get may changes into a Docker repository with a pull request • I want to be able to review documentation changes the same way I review code (github workflow)		• We shouldn't require a specialized authoring environment; currently users can contribute with a text editor and an Docker installation alone • We shouldn't require special pull requests for documentation contributors; we should leverage the current open source process for pull requests

6	Local Builds	• I want to build the documentation on my machine so I can test what I write and make sure it presents well.
		• I want to build the documentation on my machine so I can test what I write.
		• I want to be able to do all the docs processes on my Windows machine.
		• I want to be able to do all the docs process on my Mac.
				• Users may be writing on Mac OSX, Windows, and Linux
				• As we expand to multi-architecture there may be more platforms
				• Currently we have this situation:
				  • Local builds are working as unit tests. And running Hugo checks which is Sven's patch. he is going to update that to be off the latest release
				  ☐ Need a JIRA for this
				  • Integration tests are happening on the PR checkin to GitHub these checks include:
				  • Markdown linter is being run
				  • Linkchecker is not running; still in development; Sven runs it on the stage build running from checkins. Sven is working on this and will let us know. This is definitely finding this on an actual S3. When you do a Hugo server what you get is only marginally related to what you get if you do to Hugo server – Hugo serve is not showing what happens on an actually deployed server. (This is fixed by moving yo using nginx - as the local build webserver is identical to the stage&prod one)
				  ☐ Need a JIRA to track for the Linkcheck

7	Redirect support	• I want to rename or move a file in a directory structure and create a redirect from the old to the new file. This is good because it does not break Docker's old SEO.		• Within our own systems we should not use redirects. Instead, we should fix our links to point to the new file. • Links to a file that has been redirected should provide a warning in the build system. ☐ Check redirects from within Compose. / Maybe start release from AWS instance rather than a virtualbox machine

8	Live refresh	• I want to be able to run a local copy of the docs server and see my changes reflected locally by refreshing the page (must work on OSX/Windows)		• This is the Hugo watch function. Works in the Linux case fine. Does not work on Mac or LInux where the DOCKER_HOST is a vm • Maybe new Docker Volumes allows this to work
9	Reuse/Includes	• I want to write a common text and reuse or include it in multiple other documentation source files.		

10	Conditional Publishing	• I want to be able to write content that contains sections that are displayed or hidden depending on certain conditions. define a set of conditions that I can use in the source files (conditional text). I'll use these conditions to specify under what conditions to publish different parts of the source file.		• Examples in other products
				  • Drupal
				  • Framemaker
				  • Madcap Flare

11	Version	• I want to be able to ship docs for the correct version of my product with the product itself • I should be able to tell what version of the product the documentation I am reading is written for. • I want to be able to under the schema of Docker's doc URI		• The documentation for a particular product should display the version of the product the documentation is assoicated with on the page (either page layout or URL is fine) ☑ We need to take back up the URL Discussion for Documentation Site URI Layout ☐ **DOCS-251** - fetch_content.py needs to get product-version from product repo TODO - get the prod-version info from the prod repo and put into build_info.json
12	Metadata	• I want to be able to tell the last time a page was edited/changed		☐ Needs a UX ☐ **DOCS-252** - fetch&store each page's metadata TODO
13	Page Comments	• I want to be able to easily tell someone about, annotation or update existing pages, without needing to login or register for an account, learn git, or deal with the strange signing process.		
14	Archive support	• I want to look at content from older releases of a product.		• This one is directed at having a facility in the doc site itself for presenting archived versions • **DOCS-248** - Solution for archived docs TODO ☐ needs a UX
15	Change control	• I want to see quickly what's new in the current release (or see in which release something was added)		• New in the product • New in the documentation • Interim solution maybe to use the GiHub repo
16	Search	• I want to be able to search the documentation from a search bar in the documentation site		• Google search is in the new layout design ☐ Verify how it looks with multiple products /version
17	Code examples	• I want code examples to be better readable, without scrolling • I want to be able to copy / paste examples • I want to know that the latest documentation is correct - or at least know when code changes have affected examples. • I want to be able to include the output of a command (ex: `docker-compose --help`) in the docs in an automated fashion		
18	Generate command docs	• I want the man pages and command line documentation to be kept in sync, preferably by coming from the same source files		
19	Generate API docs	• I want to be able to generate API documentation for display on docs.docker.com		• DTR is already there • Docker Python API is already there
20	Man pages	• I want man pages to be an automatic part of all projects with a cmdline • I want the man pages and command line documentation to be kept in sync, preferably by coming from the same source files		
21	Publish	• I want to publish a new version for just a single product • I want to publish an update to a single page into the latest published version • I want to publish an update to one or more pages in an older, archived version of the documenation • I want to archive an existing version of the product • I should be able to rollback to a particular version of the documentation for any product. • I should be able to rollback to a particular version of a page. • Keep an audit log of what was published, who published it, and when		• For 1.10 release the intention that "I" is Mary or Sven • see Publishing workflow

22	Authorize publishers	• I want to be able to publish our product documentation • I want to be able to lock users from publishing per project or as a whole • I want to be able to lock users from publishing per project or as a whole		• Requires that we have 23 Recovery/Rollback in place
23	Recovery	• I should be able to rollback to a particular version of the documentation for any product. • I should be able to rollback to a particular version of a page.		• May or may not come out by 11 • see Publishing workflow (make a PR to the docker/docs-html repo, validate and merge to GO-LIVE)
24	Output/Display	• I want to be able to read the documentation on a website • I want to see documentation for just the product I'm using (e.g. only docker-compose, swarm, engine, docker-machine)		• The new layout of the documentation is product-specific
25	PDF support	• I want to be able to download an offline copy of the documentation (in epub, PDF format) • I want to generate PDF of a set of product documentation		
26	Audit trail	• Keep an audit log of what was published, who published it, and when		• We need that before we can do the publish • Use the commit history and Pull Request history of the docker/docs.docker.com, docker/docs-sources and docker/docs-html repositories for test/stage, and repository tagging for each live site
27	Analytics	• Google Analytics should be imbedded in each document source page		• Supported in current version of the product
28	Support	• I want to be able to provide writers and contributors with documentation on how to create source content • I want to be able to provide user with instructions on how to publish		

User interaction and design

Questions

Below is a list of questions to be addressed as a result of this requirements document:

Question	Outcome

Not Doing

- I want to use an editor I am familiar with to write documentation.
- I don't want to spend a lot of time learning a new tool so I can write documentation.
- I don't want to create a new login to create documentation.
- I want to get may changes into a Docker repository with a pull request

Validate documentation files on commit

Traditional authoring environments validate a file when the file is saved. Depending on the environment, the validation may be very strict (DITA) or basic (Framemaker, Drupal, Confluence). What they have in common is they have features that automatically validate the file content. The analogy is to a compilation with code. Validation automates many things for a writer.

- Checks the file contain valid markup
- Automatically numbers procedures and headings
- Ensures the cross-references (links between files a company's documentation) in the file resolve
- Ensures links to external resources resolve
- Checks the markup logical (H1 is only followed by H2)
- Includes exist and resolve in the documentation

These kinds of automated validation may be done while a user writers, on a documentation save, or when a documentation set is generated.

An editor + Markdown doesn't automate any documentation validation. Instead, it relies on a human being to manually and personally validate. As a result, the checks are highly error prone.

Tasks addressing this situation

Develop checks that ensure markdown files are validated when users check them into a repository.

Key	Summary	T	Created	Updated	Due	Assignee	Reporter	P	Status	Resolution
DOCS-228	add a junit.xml file output to markdownlint, linkchecker and hugo		Nov 23, 2015	Nov 24, 2015		Sven Dowideit	Sven Dowideit	?	IN PROGRESS	Unresolved
DOCS-186	defuse hard-coded links to docs.docker.com		Sep 23, 2015	Jan 20, 2016	Oct 23, 2015	Sven Dowideit	Mary Anthony	?	SELECTED FOR DEVELOPMENT	Unresolved

2 issues

Doc Build, Validation, and Release

Releasing Docs: Illustrated and explained

This page explains the components of and process for releasing documentation both for an official release and as ***updates*** to the existing docs.docker.com site.

Understand what is involved

The documentation source files are co-located with the documentation product's code source files. This has an advantage:

- When new features are added to a product the documentation is also added; if done in a single PR it can be reverted if necessary in a single PR
- Feature code and documentation are reviewed simultaneously in a single rather than duplicated effort
- Easier for code contributors to find the documentation related to a product they are working on

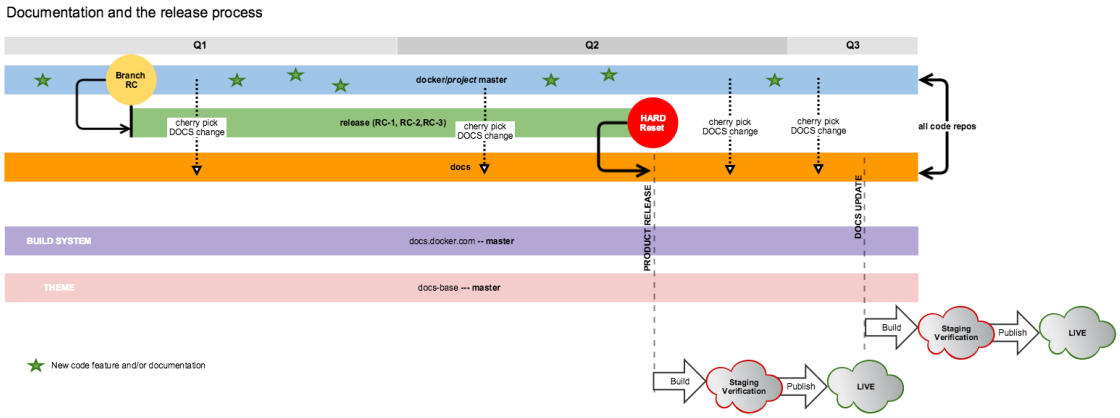
There is a tradeoff of course. Some disadvantages are:

- Documentation for new features can lag or require updates after a **release candidate(RC) branch** is cut.
- Small documentation changes require PRs that add to a product's PR review workload
- Documentation management must manage documentation across multiple project repositories

List of projects that feed docs.docker.com:

	Repository	Description
1	https://github.com/docker/docker	Docker Engine. Public. Docker CS. Commercial
2	https://github.com/docker/distribution	Docker Registry. Public.
3	https://github.com/docker/swarm	Docker Swarm. Public.
4	https://github.com/docker/machine	Docker Machine. Public.
5	https://github.com/docker/compose	Docker Compose. Public.
6	https://github.com/docker/notary	Notary. (TBD)
7	https://github.com/docker/dhe-deploy	Content only for DTR. Private.

8	https://github.com/docker/hub2-demo	Hub Private docs
9	https://github.com/kitematic/kitematic	Kitematic. Public
10	https://github.com/docker/opensource	Open source
11	https://github.com/docker/tutorials	Getting startet



New Product Release

Purpose: Publish new product documentation and updates to existing feature documentation.

1. As features are added, code and doc changes add into a product's **docker/*project*** **master** branch.
2. When the engineering team is ready, the cut an RC branch from **master** that includes code and documentation.
3. Engineer creates a **release** branch
4. Engineer updates the **docs** branch from **release**.
5. The documentation team publishes **staging.docs.docker.com.**
6. Release watchers verify the content on the staging site.
7. The documentation team publishes **docs.docker.com** site.

Between Release Documentation Updates

Purpose: Publish updates to existing feature documentation ***NOT*** *a ny* new feature documentation.

1. As new features are added, code and doc added into a product's **docker/*project*** **master** branch
2. As documentation is updated for existing features the change is added to a product's **docker/*project*** **master** branch.
3. At this point, **master** contains documentation for ***existing*** and ***new*** features co-mingled.
4. Writer copies *a documentation update for* ***existing*** *feature* from **master** into the **docs** branch with a cherry pick
5. Build and pushed to **stage.docs.docker.com**
6. Interested party verifies staging
7. The documentation team publishes **docs.docker.com** site.

Repository	Description
https://github.com/docker/docs-base	Contains • The documentation theme (look and feel) • The documentation splash page • The release-notes page
https://github.com/docker/docs.docker.com	Contains the build scripts for building and publishing documentation