

Basics

First, check out [this brief explanation](#) on what Git is, why it was created, and general descriptions of features or ideas Git is built around. This will help a lot to understand the commands you are using, instead of just searching for a command that does something favorable. You will gain a lot of context by reading this page.

If you are new to git entirely, I'd recommend checking out [this interactive tutorial to learn git branching](#) and running through the examples they host there. This will get you working with Git quickly in various situations and difficulties. After this, you will have some experience with Git!

The [Git-scm Book](#) is a good read and also serves as a great online reference. Once you have an idea of what you are looking for and where you need further your understanding, this will be useful to you.

And when all else fails, [ohshitgit](#) outlines what to do in a few `oh shit, I fucked up` scenarios. [dangitgit](#) is the same reference, without the bad language and thus is more suitable to leave up on a work monitor or to share in a presentation.

Create a Repository

To create a repository, just create a directory or enter the root directory of the project you want to turn into a repository, and type `git init`. This initializes the directory as a *local* repository. To add the repository to GitHub and track it remotely, you'll need to login to GitHub and click 'New Repository', name the repository the same as your root folder and continue. GitHub will provide you with the rest of the instructions, but for completeness, tweak the lines below to push your local repository to your new remote on GitHub -

```
git remote add origin git@github.com:<username>/<reponame>.git
git push -u origin master
```

We just created the `origin` remote. This is the remote that is displayed and tracked on GitHub, when you clone your repository you are on a local remote, which means until you `git push <remote> <branch>` your changes will only be saved and tracked on your local machine.

Using a remote via SSH such as the above `git@github.com:<username>/<reponame>.git` requires you have configured an SSH key with your GitHub account that is associated with the machine you are pushing from. If you haven't already, check out [Creating SSH Login Keys](#) and simply `cat ~/.ssh/<USERKEY>.pub` the public key of your user and copy it over into your GitHub settings.

If you'd rather not mess with things like this, see how to create a [Person Access Token](#) below.

Ignoring Files

Within a Git repository, `.gitignore` files can be seen specifying a list of files or directories that Git should ignore when tracking changes. For example, this is useful when a project is expected to contain build files generated after being cloned. We would not want the user to then make a commit publishing the files they generated when building the project for their system, we would want to provide a clean slate for the next person that clones the project.

After creating a `.gitignore` file, the syntax below can be followed to specify files and directories to be ignored.

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

“ [Git-scm - Ignoring Files](#)

Check out [GitHub - gitignore repo](#) for some templates used in popular languages, like C, C++, or Python.

If you are ignoring a file that Git has already previously tracked, it may be necessary to remove the file (or directory) from Git's cache using the command below

```
# Remove cached file
git rm --cached path/to/file

# Remove cached directory
git rm -r --cached path/to/

# Remove all cached files
git rm -r --cached .
```

Commit Guidelines

Before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like `git merge` and `git revert`.

Use `git add --patch` to partially stage files (covered in detail in https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging#_interactive_staging)

The project snapshot at the tip of the branch is identical whether you do one commit or five, as long as all the changes are added at some point

As a general rule, your commit messages should start with a single line that's no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation

`git log --pretty=oneline` shows a terse history mapping containing the commit id and the summary

If the config option `merge.summary` is set, the summaries from all merged commits will make their way into the merge commit message

“ `git pull --rebase` What's happening here? Git will rewind (undo) all of your local commits, pull down the remote commits then replay your local commits on top of the newly pulled remote commits. If any conflicts arise that git can't handle you'll be given the opportunity to manually merge the commits then simply run `git rebase --continue` to carry on replaying your local commits.

[How to Avoid Merge Commits in Git - Kernowsoul](#)

`git shortlog` uses summary lines in the changelog-like output it produces -

`git format-patch`, `git send-email`, and related tools use it as the subject for emails.

Reflogs, a local history accessible with `git reflog`, is intended to help you recover from stupid mistakes by providing the hashes along with output similar to `git shortlog`.

Reversing Changes

Use `git reset <remote>` for local changes -

```
# one commit in the past
```

```
git reset HEAD^
```

```
# 2 commits in the past
```

```
git reset HEAD^^
```

```
# 2 commits in the past
```

```
git reset HEAD~2
```

```
# 3 commits in the past
```

```
git reset HEAD~3
```

Use `git revert <remote>` for changes that have already been pushed to a remote -

```
# 1 commit in the past
```

```
git revert HEAD
```

```
# 2 commit in the past
```

```
git revert HEAD^
```

```
# 3 commits in the past
```

```
git revert HEAD^^
```

```
# 3 commits in the past
```

```
git revert HEAD~2
```

```
# 4 commits in the past
```

```
git revert HEAD~3
```

```
# Note that HEAD could be replaced with v0.2, or any active branch that exists on the remote
```

Modifying Previous Commits

WIP

For now, [Here's a good tutorial](#)

:)

```
git rebase --interactive --autosquash --rebase-merges --root master
```

Revision #22

Created 19 July 2019 03:11:51 by Shaun Reed

Updated 18 December 2021 18:37:27 by Shaun Reed