Short Circuit

Git

Pro Git
by Scott Chacon & Ben Straub
Second Edition
Up until Section 7.1
Already did Section 7.3
Half-assed Section 4.1

Summary by Emiel Bos

1 Intro

For command line notation, we use `<>` for required arguments and `[]` for optional arguments. `|`-separators indicate mutually exclusive items, usually used to show the full option name and its shorthand.

2 Basics

Local Version Control Systems (LVCSs) are simply local databases that track version control of files (on the same system). *Centralized Version Control Systems* (CVCSs) like CVS, Subversion and Perforce, have a single server that contains the versioned files, and therefore allow clients collaborating on and checking out files from that central place, which only stores mere snapshots locally. Both constitute a single point of failure, and therefore *Distributed Version Control Systems* (DVCSs) like Git, Mercurial, Bazaar or Darcs, fully mirror the project and its full history. However, where other DVCSs treat a project as a set of files and store different versions as a list of changes to the original versions of these files on a per-file basis (this is called delta-based version control), Git instead treats a project as a miniature filesystem, and stores different versions as snapshots. Each snapshot stores the entire project at that moment, and only omits files that haven't changed and simply links to a previous identical file. This has benefits for branching.
A snapshot is created by committing[1], which creates a `commit` object containing author name and e-mail, commit message, pointers to the (parent) commit(s) that precede it – zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches – and a pointer to the root `tree` object that represent the actual project. This tree corresponds to the project's directory structure; each subdirectory is checksummed and stored recursively as another `tree` object, each of which stores a list of its containing subdirectories as `tree` object and files as `blob` objects, which store the file itself and it's checksum.

Because the entire history of a project is stored locally, most operations don't need to communicate to other computers.
Everything in Git is checksummed with SHA-1 before it is stored and is then referred to by that checksum. Therefore, commit IDs are hashes.
Git is designed such that almost all operations only add data, making it very hard to screw something up irreversibly.

2.1 File states

Files in an initialized repository can be either tracked or untracked, which basically means whether Git knows about them. A tracked file is either a newly tracked file or a file that was in the previous commit. A tracked file in Git always has one of three states:

- Modified, meaning the file has been changed since the last commit. These are the files you work on in your working

[1] The word "commit" is often used synonymously with "snapshot (pointed to by a commit)".

directory.

- Staged, meaning the (modified) file is added to the staging area (also in the `.git` directory) and marked in its current version to go into your next commit snapshot. The staging area is basically a waiting room for the next commit.
- Committed, or Unmodified, meaning a file is safely stored in your local database: the `.git` directory where Git stores the metadata and object database of the project.

2.2 `git config` – configuring

Gets and sets configuration variables. There are three types of configuration variables, set with the following options:

- `--system` variables; for all users and repositories on the system, stored in `<path>/etc/gitconfig`.
- `--global` variables; for individual users, stored in `~/.gitconfig` or `~/.config/git/config`.
- `--local` variables; for individual repositories, stored in a repo's `.git/config`. This is the default.

You can define variables by running `git config --<type> <variable> <value>`. Running only `git config <variable>` gives you that variable's value. More specifically defined variables override more general ones. You can view all settings with the `--list` option and where they are coming from (i.e. which configuration file has highest priority) with the `--show-origin` option.

Every Git commit uses your user name, stored in the `user.name` variable, and your e-mail, stored in the `user.email` variable.
If you want to use a different editor than the system's default, set the `core.editor` variable (e.g. with `emacs` as value).[2]
The merge conflict GUI to use when using git mergetool is stored in the `merge.tool` variable.
The default branch name is stored in the `init.defaultBranch` variable.
The default remote name is stored in the `clone.defaultRemoteName` variable.
`pull.rebase` is a boolean indicating whether to rebase when pulling.
Use `git config --global alias.<alias> 'command'` to create an alias for the supplied (sub)command, which can be any Git subcommand and any other command if prefixed with the `!` character. Then, if you type `git <alias> (other stuff...)`, Git will replace the alias with the subcommand. Some useful examples:

```
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
```

2.3 `.gitignore`

`.gitignore` text files in a Git repository tells Git which files to exclude. It is possible to have multiple `.gitignore` in subdirectories, but mostly one in the root directory is sufficient.

```
/file.txt # Ignore "file.txt" in the current directory
file.txt # Ignore any "file.txt"
/directory/ # Ignore directory "directory" in the root directory
directory/ # Ignore any "directory"
ancestor/**/descendant/ # Ignore any directory named "descendant" inside directory "ancestor" and
    its subdirectories
*~ # Ignore any files ending with a tilde (often used to mark temp files)
*.<oa> # Ignore any files ending in ".o" or ".a"
!lib.a # Track files named "lib.a" regardless (! is negation)
```

2.4 `git clone` – cloning existing repository

In order to "clone" (copy) an existing repository, use `git clone <url> [name]` followed by the repository's URL. This creates a new subdirectory in whatever directory it is run (named after the project by default or using the optional name argument), initializes a `.git` directory inside it, pulls down the data (a full copy by default; every version of every file) for that repository, checks out a working copy of the latest version, adds the source remote from which is cloned as `origin` by default, and sets the `master` branch as tracking branch. `origin` is just the default remote name; use the `-o|--origin <name>` option to name the repository something different.
In practice, you'll likely be using the `https://` protocol, but you can also use `git://` or `user@server: path/to/repo.git`, which uses SSH.

[2] On wIndows, you must specify the full path to the editor's executable.

All files will be in the Committed state.

2.5 `git init` – initializing regular directory

To turn a regular directory into a Git repository, `cd` into it and run `git init`. This creates a `.git` subdirectory. Nothing is tracked yet.

2.6 `git add` – tracking files

`git add` is a multipurpose command, meant to make files Staged for the next commit. Running it on untracked files will make them tracked and Staged. Running it on Modified files will make those files Staged exactly how they are at that point. This means those versions will go into the commit, and changing a Staged file before the next commit will lead to it being both Staged and Modified, with those latest modifications not going into the commit.

Run `git add <name>` with the filename you want to add, or with a directory name, in which case all files in it are added recursively. Similar to the `.gitignore`, it can take any Glob pattern (`git add .` adds everything recursively from the root directory).
This does not untrack deleted files; use`git add --update|-u` to remove deleted files in addition to staging modified files, or use `git add --all|-A` to do it all: adding and staging everything not in the `.gitignore` and remove deleted files.

2.7 `git rm` – untracking files

`git rm <name>` untracks a file, i.e. removes it from the staging area, or, stages the file's removal. It also removes the actual file from the hard drive; use the `--cached` to keep the file itself. The next `git commit` will actually remove the file. If you modified the file or already added it to the staging area, you must force the removal with the `--force|-f` option. (This is a safety feature to prevent accidental removal of data that is in no commit (yet) and therefore wouldn't be recoverable.

2.8 `git mv` – moving and renaming files

A convenience function that can move or rename files: `git mv <old> <new>` moves or renames a file from the first to the last argument. However, this is kind of superfluous, as Git is smart enough to detect any rename and/or move operations using other commands or using the folder explorer.

2.9 `git commit` – committing

Records a snapshot of the repository by committing the staging area.[3] All files transition from Staged to Committed.
Running plain `git commit` launches the editor as set in the `core.editor` variable (which usually is Vim er Emacs) for you to write your commit message (the commented lines are not included). You can also type your commit message inline with the `--message|-m` option, after which you should supply your message as a `"`-delimited string.
Because going through the staging area is sometimes a bit cumbersome and superfluous for a workflow, adding the `--all|-a` option makes Git automatically stage every file that is already tracked before doing the commit, avoiding a `git add --all|-A`. This does not track/stage new files, though, so if you want a commit that reflects your working directory, you should still run `git add --all|-A` beforehand.

The `--amend` option replaces the last commit with the current one, i.e. it basically redoes the last commit, handy if you have additional (staged) changes you forgot to include. Amending previously pushed commits may cause problems for your collaborators; use with cation.

2.10 `git status` – showing (file) state

Running `git status` tells you which branch you're on, whether that branch diverges from the one on the server, whether you have untracked files in the directory, whether you have files Staged to be committed, whether you have Modified files not staged for a commit, and whether you have unmerged paths (files) as a result of a merge conflict.
Running with the `--short|-s` option gives a much less verbose output, a list of files with their state in the staging area and their state in the working tree as the first and second column respectively. `?` indicates untracked next to them, `A` indicates added/Staged, `M` indicates modified.

[3] Often, and also in this summary, "commit" is often synonymous with "snapshot (pointed to by a commit)".

2.11 `git diff` – showing changes

Basically a much more sophisticated `git status`. Shows actual code changes.
Running `git diff` tells you what you've Modified but not yet Staged by comparing what is in your working directory with what is in your staging area. (If you've staged all of your changes, `git diff` will give you no output.)
Running `git diff --staged` tells you what is Staged and ready for commit by comparing your staged changes to your last commit. The `--cached` option is synonymous.

2.12 `git difftool` – showing changes in a GUI

For running `git diff` in a GUI. Use `git difftool --tool-help` to see which GUIs are available on your system.

2.13 `git log` – showing commit history

`git log` without arguments lists the current branch's commits in reverse chronological order (i.e. latest first). Each commit displays checksum, author's name and email, date, and message.
The `--patch|-p` option additionally displays the patch (i.e. the diff, or difference) introduced by each commit.
The `-<n>` option displays only the last n commits.
The `--since|--after` and `--until|--before` allow for specifying a timeframe in lots of formats, e.g. `--since="2 years 1 day 3 minutes ago"`.
The `--author` option allows for filtering by author. The `--grep` option allows for filtering commit messages by keywords. More than one instances of these options are allowed. The `--stat` option shows abbreviated stats under each commit: a list of modified files, how many files were changed, and how many lines in those files were added and removed. `--shortstat` display *only* the changed/insertions/deletions line.
The `--pretty` option is for pretty-printing: `--pretty=oneline` prints each commit on one line; `short`, `full`, and `fuller` dictates the verbosity/brevity of the output; and with `format` you can specify the output explicitly (handy for machine parsing), e.g.:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : Change version number
```

Useful specifiers are listed here.

The `format` and `oneline` options are especially useful in conjunction with the `--graph` option, which adds a nice little ASCII graph showing your branch and merge history next to the list of commits.
`--abbrev-commit` shows only the first few characters of the checksum.
`--oneline` is a shorthand for `--pretty=oneline --abbrev-commit`.
`--relative-date` displays the date in a relative format (e.g. "2 weeks ago").
The `--S` option shows only those commits that changed the number of occurrences of that string.
`git log <options> -- path/to/file_or_dir` shows only commits that introduced a change to those files.

`git log` only shows the current branch's (`HEAD` commit history. Use `git log <name>` to show the commit history of the specified branch. Use `--all` to show all the branches.
Use `--decorate` to see to which branch `HEAD` is pointing.

2.14 `git reset` – undoing

`git reset --hard` reverts the entire working directory to `HEAD`, which is the latest commit on the current branch, i.e. you're undoing all changes since the latest commit.
You can also specify a different commit than `HEAD`. `git reset HEAD <file>` unstages a staged file.

`git reset <commit>` without a `--hard` or `--soft` moves your `HEAD` to point to the specified commit, without changing any files. E.g. `git reset HEAD^` will get rid of the last commit but keeps its changes.

2.15 `git restore` – restoring files

`git restore --staged <file>` unstages a staged file. (Basically supercedes `git reset HEAD <file>`.)
Use `git restore <file>` to undo any local changes made in the file since – and revert back to the version at – the last commit. (Basically supercedes `git checkout -- <file>`.) Contrary to `reset`, `restore` does not update the current branch.

2.16 `git tag` – tagging

You can tag commits, usually used to mark release versions. There are two types:

1. a *lightweight tag* is just a pointer to a commit.
   Create a lightweight tag with `git tag <tag> [commit]` without any other options.
2. an *annotated tags* is a full object that additionally stores checksum; tagger name, e-mail, and data; message; and optional GNU Privacy Guard (GPG) signature.
   Create an annotated tag with the `--annotate|-a` option: `git tag -a <tag> [commit]` optionally with `-m "message"`.

In both cases, omitting the optional commit argument just tags the latest commit (`HEAD`).
`git tag` lists existing tags in alphabetical order (supplying `--list|-l` is optional).
If you wanna pattern search, use `--list|-l` with a `"`-delimited pattern string (in which `*` is a wildcard).
Use the `--delete|-d` option to delete a specified tag.
Adding and deleting tags on a remote is done using `git push`.

2.17 `git show` – showing tags

`git show <tag>` displays the specified tag's commit, and – if its annotated – also the tagger information, tag date, and message.

2.18 `git stash` – temporarily putting away work without committing

Useful if you want to work on something else without creating a commit. Use `git stash` (which is equivalent to `git stash push`) to save the dirty state of your working directory (i.e. your modified tracked files and staged changes) on a stack of such *stashes* that you can reapply at any time, even on a different branch. Use `--keep-index` to also leave all staged content in the index as well. Use `-include-untracked|-u` to also include untracked files in the stash. This will not include explicitly ignored files; to include those as well, use `--all|-a`. The `--patch` option will interactively prompt you which of the changes you would like to stash and which you would like to keep in your working directory.

Use `git stash list` to list all stashes on the stack in order of recency (e.g. `stash@{0}` is the latest stash), together with the branch in which it was stashed and the latest commit at that time (i.e. the commit onto which the stash was built).

Use `git stash apply [stash]` to replay the changes of the specified stash on the current branch. Of no stash is specified, the most recent is applied. May result in merge conflicts. Files that were previously staged won't be automatically restaged; use the `--index` option to restage those. `git stash apply` will not remove the stash; use `git stash drop` to remove it. Use `git stash pop` to both apply and then drop the stash.

`git stash branch <newbranch> [stash]` creates a new branch, checks out the commit on which the stash was created, applies the stash and drops it if it applies successfully.

2.19 `git clean` – cleaning working directory

Removes untracked files from your working directory. Use option `-d` to recurse into untracked directories as well; this requires `--force|-f` if `clean.requireForce` is set to `true`. Use `--dry-run|-n` to see what would be removed without actually doing it. Add option `-x` to also remove `.gitignore`d files. `-i` will run `git clean` interactively.

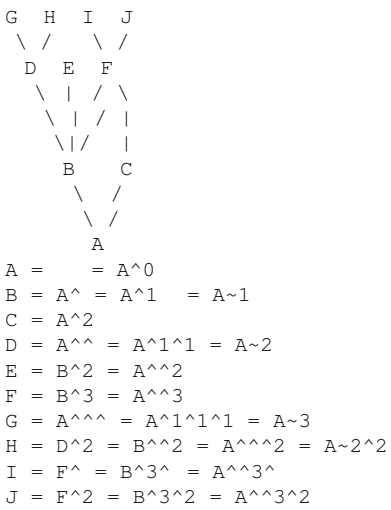
3 Branching

A branch in Git is simply a lightweight, movable (simple file that contains a 40 character SHA-1 checksum) pointer to a commit. The default branch name is `master` by default[4], but this can be changed in the `init.defaultBranch` config variable. Git maintains which local branch you're on in the `HEAD` pointer, which point to the latest commit (tip) of that branch.

`HEAD~[n]` is used to refer to `HEAD`'s `n`'th ancestor commit, i.e. `HEAD~` = `HEAD~1` refers to `HEAD`'s parent commit, `HEAD~2` = `HEAD~~` = `HEAD~1~1` to `HEAD`'s grandparent, etc. At a merge commit, which has more than one parent, the first/leftmost parent is traversed (the order of parents can be obtained by `git show`).
`HEAD^[n]` is used to refer to `HEAD`'s `n`'th parent, starting from the left, i.e. `HEAD^` = `HEAD^1` refers to `HEAD`'s first parent, `HEAD^2` = `HEAD^^` = `HEAD^1^1` to `HEAD`'s second parent, etc. Note that `HEAD~` equals `HEAD^`. In general, you'll mostly use `~`

[4] GitHub started using `main` in 2020 to avoid negative connotations.

to go back a number of generations, and you may need `^` at merge commits if you want to traverse a parent other than the first. You can also chain the two notations interchangeably, as demonstrated by this example:



It's also possible for `HEAD` to point to a specific commit rather than a branch, which is knows as a detached `HEAD` (and you can only get yourself in this state using `checkout`).

3.1 `git branch` – branch management

`git branch` without additional arguments lists all local branches, with `*` indicating the current `HEAD` branch. Supplying the `-v|--verbose` option additionally lists the latest commit per branch. `-vv` lists the remote-tracking branch a branch may track and the number of commits are ahead or behind that branch (note that this command itself doesn't communicate to the remote and only tells what it has cached since the last fetch). The `--merged` and `--no-merged` options filter the list to branches that you have or have not yet merged into the current `HEAD` branch. Using `--merged` or `--no-merged` with a specific branch name shows the list with respect to that branch. `git branch <name>` creates a branch pointing to the current commit (but doesn't switch to it).
Use `git branch (--delete|-d)<name>` to delete the specified branch. This will fail if the specified branch has unmerged work; use `-D` (shorthand for `--delete --force`) to delete it anyways.
To rename a branch locally (and keep its history), use `git branch --move <old_name> <new_name>`. Use `git push --set-upstream origin <new_name>` to push the renamed branch to the remote, and `git push origin -d <old_name>` to remove the old branch from the remote. Do not rename branches that are still in use by other collaborators.
To (retroactively) make a branch a tracking branch, you can use the `-u|--set-upstream-to` option with the specified remote-tracking branch.

3.2 `git switch` – switching branches

Use `git switch <name>` to switch to the specified, existing branch, i.e. move `HEAD` to point to the specified branch and change files in the working directory accordingly. Use option `--create|-c` to additionally create the branch, avoiding a `git branch` command.
Use `git switch -` to return to your previously checked out branch.

`git switch` was introduced in Git 2.23 to clear some of the confusion that comes from the many overloaded usages of checkout (switching branches, restoring files, detaching HEAD, etc.), i.e. it does exactly the same as a subset of `git switch`'s functionality, namely switching branches.

3.3 `git checkout` – switching and creating branches

Combines the functionalities of `switch` and `restore` into one swiss army knife command. (Actually, `checkout` is the original command, and `switch` and `restore` were introduced to split its functionality into its two use cases and "simplify" their commands.) Use `git checkout <name>` to switch to the specified, existing branch, i.e. move `HEAD` to point to the specified branch and change files in the working directory accordingly. You can only checkout when you have no uncommited changes, i.e. you have a clean working state.
Use option `-b` to additionally create the branch, avoiding a `git branch` command. If you also specify a remote branch

(i.e. `git checkout -b <branch> <remote>/<branch>`), you create a tracking branch that tracks the specified remote branch. This can be shortened with the `--track` option, in which case you don't need to specify the name of the tracking branch (it will get the same name as the remote branch to be tracked). This is so common that when you simply type `git checkout <branch>` and that branch doesn't exist locally but matches a branch on exactly one remote, it will create a tracking branch tracking that branch.
Use `git checkout <commit>` with to checkout to the commit of the supplied SHA-1 hash. This will get you into a detached `HEAD` state, which is a rather unintuitive phrase that means you are not currently on (the latest commit of) any branch. If you checkout or switch back to a branch again, all changes made in the detached `HEAD` state will be lost. If you want to keep the changes, you have to create a branch in the detached `HEAD` state and potentially merge that.

Use `git checkout -- <file>` to undo any local changes made in the file since – and revert back to the version at – the last commit.
Use `git checkout [tag]` to switch to the version of the project as per the tag's commit, in a detached `HEAD` state. Use `git checkout -` to have `HEAD` point to whatever it was pointing at previously (i.e. running `git checkout -` twice will have `HEAD` point to the same branch or commit).

3.4 `git merge` – merging branches

`git merge <name>` merges the specified branch into the current branch pointed to by `HEAD`.
If the current `HEAD` commit can be reached by following the about-to-be-merged commit's history, Git simply moves the `HEAD` pointer forward (because there is no divergent work to merge together) and indicates this with the `Fast-forward` phrase.
If this is not the case, and the respective histories have diverged, Git does a simple three-way merge between the two commits pointed to by the branch tips and their common ancestor, and creates a new special sort of commit: a *merge commit*, which has two parents.
If both divergent paths modified the same part of the same file, Git lists this as a conflict (also listed by `git status`) and pauses the process. A conflict in a file looks like this:

```
<<<<<<< HEAD:file.cpp
int number = 5;
=======
uint number;
>>>>>>> other_branch:file.cpp
```

This entire block should be replaced with manually resolved code. Use git add to stage the file after you've solved all conflicts in it, which will mark it as resolved. If everything has been resolved, use `git commit` to finalize the merge.
After a merge, you'll likely want to delete the merged branch with `git branch -d <name>`. (Use `git branch --merged` to list all such branches.)

3.5 `git mergetool` – merging branches in a GUI

Boots an appropriate visual merge tool to walk you through the conflicts. If the `merge.tool` config variable isn't configured, Git displays a list to choose from (and a default if you do not select any).

3.6 `git rebase` – merging branches

An alternative way – besides `git merge` – of integrating changes of one branch into another, that takes a sequence of commits that were committed on one branch and replays/copies them on some other (base) commit of another. There is no practical difference; the end product is the same, but with rebasing you get a linear history, which can be preferred.

There are two modes of rebasing:

- Standard. Use `git rebase <base_commit>` to rebase the current branch onto the specified commit (which can be any kind of commit reference, like an ID, branch or tag name, etc.), which takes the commits in your current working branch and applies them to the head of the specified commit. Technically, Git goes to the common ancestor of the two branches, gets the diff introduced by each commit of the current branch (and saves those diffs to temporary files), sets the current branch to the same commit as the branch you are rebasing onto (using `git reset --hard <branch>`), and applies each change in turn. This means that the rebased-onto branch is behind (only the just-rebased branch points to the latest commit; the rebased-onto branch pointer hasn't moved), so you can switch to that branch and do a fast-forward merge with the rebased branch to catch-up with the rebased diffs.
  You don't have to switch to the to-be-rebased branch if you specify it: `git rebase <onto_branch> <rebase_branch>`.

You still need to fast-forward.
Use option `--onto` to specify a different branch to rebase onto than the branch that the current branch branched off of. Basically, `git rebase --onto <onto_branch> <diverged_from_branch> <rebase_branch>` takes the patches of `<rebase_branch>` since it diverged from `<diverged_from_branch>` and applies these on the `<rebase_branch>` as if it was based directly off the `<onto_branch>` instead. Now you can fast-forward the `<onto_branch>` again.

- Interactive. With the `-i/--interactive` option you start an interactive rebase session, where you can edit each commit individually and rewrite history. It is very useful on a single branch as a sort-of `git commit --amend` on steroids. Because you specify the base on which to replay the commits, you specify the commit immediately before the earliest commit you want to modify, so the last commit you want to retain as-is. Your set text editor will open with a list of the commit, oldest first. All commits are picked by default. You can change the `pick` command to any of the command listed at the bottom. Closing the editor will start the process. Git will operate the listed command in the specified way in-order. Use `git rebase --continue` to move forward. Use `git rebase --abort` to abort and discard all changes.

Do **not** rebase commits that exist outside your repository and that people may have based work on, because you're essentially abandoning existing commits and creating new ones (with new SHA-1's) that are similar but different. Things will get messy.

4 Remotes

Remotes are like nicknames for the URLs of repositories; versions (generally one, may be more) of your project that are hosted elsewhere: on the internet, network, or even on the same system/host. If your repository is a fork, you could for example add the original repository as a remote, in order to occasionally update your fork with commits to the original.
Each remote has a name. `origin` is the default name Git gives to the remote you cloned from. You can interchangeably use a remote's name and URL.
Remote-tracking branches are local branches (though they can't be moved by the user) that take the form `<remote>/<branch>` and which reflect the corresponding branch in the remote; they're pointers to the commit at which the remote's branch was at the last time you connected to the remote.
Tracking branches are local branches that directly track a remote branch (aka an upstream branch) via a remote-tracking branch. In practice, nearly any branch you're working is likely a remote tracking branch. On such a branch you can use `git pull`/`git merge` to automatically pull/merge the remote branch into the tracking branch. `@{upstream}`/`@{u}` are shorthands for the upstream remote tracking branch.

4.1 `git fetch` – fetching from remotes

`git fetch <remote>` (only) fetches/downloads all data from the specified remote that isn't already locally available (but doesn't merge anything and doesn't change working directory) and sets/updates the corresponding remote-tracking branches. After this, you have references to all the branches from that remote which can be merged or inspected (locally). For example, this means that you may have a `master` branch and a `origin/master` branch that point to different commits. However, fetch new remote-tracking branches doesn't download actual working copies; you'd have to merge those remote-tracking branches first.

4.2 `git pull` – pulling (fetching + merging) from remotes

If your current branch is a tracking branch, `git pull` both fetch and then merge that remote branch into your current branch.
It will warn if the `pull.rebase` config variable is not set.

4.3 `git push` – pushing to remotes

`git push <remote> <branch>` pushes the specified branch upstream. Only works if you have write access to the remote and nobody has pushed in the meantime (so first pull). Use `git push <remote> <local_branch>:<remote_branch>` if you want to push your branch to a differently names remote branch.
Use the `--tags` option to push tags (that are not already there); it doesn't do this by default. Both types of tags will be pushed; replace `--tags` with `--follow-tags` to only push annotated tags.
To only push a specific tag, replace `--tags` with that tag. Use the `--delete|-d` option before the tag to delete the specified tag.
With the `--delete|-d` option you can delete remote branches from a remote: `git push <remote> --delete <branch>`.

4.4 `git remote` – remote management

It lists the names of each remote handle you've specified, which often is only `origin`.
Adding `--verbose|-v` additionally shows the URLs corresponding to the names.

Running `git remote add <name> <url>` explicitly adds another remote specified by the URL (with nothing fetched and no remote-tracking branches set).

`git remote rename <old> <new>` renames the specified remote and all associated remote-tracking branch names to the new name.

`git remote remove <name>` removes the specified remote and associated remote-tracking branches and configuration settings. (`git remote rm` is synonymous.)

`git remote show <remote>` gives more information about the specified remote: its URL; a list of remote branches and whether they are `tracked` (reference has been fetched), `new` (not yet fetched), or `stale` (removed from remote since fetch); what remote branch is pulled from to which local branch; and what remote branch is pushed to from which local branch.

5 Serverside

For collaboration, although you can technically push and pull to and from individuals' repositories, this is highly impractical, and you'd want to have a Git server with an intermediate remote repository. Such a remote repository is generally a *bare repository*: a Git repository that has no working directory checked out, i.e. there's only a `.git` folder on disk.

5.0.1 Protocols

Git can use any of the following five transport protocols:

- **Local**. The remote repository is in another directory on the same host, often used if collaborators have access to a shared filesystem such as an NFS mount (or if everyone uses the same PC). This is easy to setup and uses existing file permissions and network access, though can be difficult to reach from outside the network, and can be slower than using SSH on the same server.

  You reach the remote using e.g. `/srv/path/to/project.git`. You can also use `file:///srv/ path/to/project.git`, which is slower (because Git uses the process it normally uses to transfer data over a network instead of using hardlinks or directly copying the files) but which gives a clean copy of the repository with extraneous references or objects left out.

- **Dumb HTTP(S)**. Prior to Git 1.6.6, this was the only HTTP(S) protocol, but has been superseded by the smart HTTP protocol, only used for read-only repositories or as fallback when a server doesn't respond with a HTTP smart service.

  You reach the remote using e.g. `https://example.com/project.git`. It is very easy to set up; it only expects the bare Git repository to be served like normal files from the web server, and all you have to do is put a bare Git repository under your HTTP document root and set up a specific post-update hook.

- **Smart HTTP(S)**. Operates very similarly to the SSH or Git protocols, but runs over standard HTTP(S) ports and can use various HTTP authentication mechanisms. This means it's often easier than something like SSH. Only a single URL is needed for reading (cloning) and writing (pushing), both with (optional) authentication with user name and password rather than generating SSH keys, which makes it easy for the end user. Like the SSH protocol, it is fast and efficient. However, it can be a little more tricky to set up compared to SSH on some servers.

- **SSH**. Common when self-hosting because SSH access to servers is already set up in most places, and it's easy te setup otherwise.

  You reach the remote using e.g. `ssh://[user@]server/project.git`, or with the shorter scp-like syntax `[user@]server:project`
  In both cases above, if you don't specify the optional username, Git assumes the user you're currently logged in as. All data is encrypted and authenticated, and transfer is efficient. However, it doesn't support anonymous access to your Git repository. If you're using SSH, people must have SSH access to your machine, even in a read-only capacity, which doesn't make SSH conducive to open source projects for which people might simply want to clone your repository to examine it. If you're using it only within your corporate network, SSH may be the only protocol you need to deal with. If you want to allow anonymous read-only access to your projects and also want to use SSH, you'll have to set up SSH for you to push over but something else for others to fetch from.

- **Git**. A special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication. You must create a `git-daemon-export-ok` file.

  It is often the fastest network transfer protocol available. However, there is no authentication, so you'll likely pair it with SSH or HTTPS access for the few developers who have push (write) access and have everyone else use `git://` for read-only access. It's also probably the most difficult protocol to set up.