# Version Control

## GIT

### 1 Pro Git, 2nd Edition

#### Introduction

* What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later
* DVCS systems deal pretty well with having **several remote repositories** they can work with, so you can collaborate with different groups of people in different **ways simultaneously within the same project**
* Conceptually, most other systems store information as a list of **file-based changes (deltas)**. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a **set of files** and the **changes made to each file over time**.
* Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically **takes a picture of what all your files look like at that moment** and **stores a reference to that snapshot**. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots
* Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means **it’s impossible to change the contents of any file or directory without Git knowing about it**. This functionality is built into Git at the lowest levels and is integral to its philosophy
* The mechanism that Git uses for this check summing is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git

#### Tagging & Branching

* A tag represents a version of a particular branch (e.g. master) at a moment in time.
* Usually you'll tag a particular version so that **you can recreate it**, e.g., this is the version we shipped to XYZ Corp.
* **Tags** are symbolic names for a **given revision**. They always point to the same object (usually: to the same revision); they do not change.
* A tag is supposed to represent an **immutable** content, used only to access it with the **guarantee to get the same content every time**.
* A branch represents **a separate thread of development** that may run concurrently with other development efforts on the same code base. Changes to a branch may eventually be merged back into another branch to unify them.
* A branch is more of a strategy to provide on-going updates on a particular version of the code while continuing to do development on it (on the main line). You'll make a branch of the delivered version, continue development on the main line, but make bug fixes to the branch that represents the delivered version.
* We often have to work on multiple things in parallel: feature X, bug fix #32, feature Y… This makes it all too easy to lose track of where each change belongs. Therefore, it’s essential to keep these contexts separate from each other. Grouping related changes in their own context has multiple benefits: your co-workers can better understand what happened because they only have to look at code that really concerns them. And you can stay relaxed, because when you mess up, you mess up only this context. Branches do just this: they provide a context that keeps your work and your changes separate from any other context.
* Eventually, you'll merge these bug fixes back into the main line. Often you'll use both branching and tagging together. You'll have various tags that may apply both to the main line and its branches marking particular versions (those delivered to customers, for instance) along each branch that you may want to recreate -- for delivery, bug diagnosis, etc.
* **Branches** are symbolic names for **line of development**. New commits are created on top of branch. The branch pointer naturally advances, pointing to newer and newer commits.
* If you checkout a tag, you will need to create a branch to start working from it.
* If you checkout a branch, you will directly see the latest commit in ('HEAD') of that branch.
* The main point is: a tag being a simple pointer to a commit, you will never be able to modify its content. You need a branch.
* HEAD is the symbolic reference to your current branch
* HEAD is the parent of your next commit
* HEAD is what is (or should be) last checked out in your working directory

#### Branching

In fact, this is so common that there’s even a shortcut for that shortcut. If the branch name you’re trying to checkout (a) doesn’t exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you: e.g.

$ git checkout serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

#### Useful commands:

* + git config --list 🡪 show all configuration key/value pairs
  + git config user.name 🡪 show value for specified key
  + git config --global core.editor notepad
  + git config --global http.proxy <http://10.9.64.9:8080>
  + git config --global commit.template ~/.gitmessage
  + git init 🡪 initialise repository
  + [alias] 🡪 .gitconfig
    - hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
  + [merge]
    - tool = p4merge
    - [mergetool "p4merge"]
    - path = C:\\Program Files\\Perforce\\p4merge.exe
  + [diff]
    - tool = p4merge
    - [difftool "p4merge"]
    - path = C:\\Program Files\\Perforce\\p4merge.exe
  + <http://stackoverflow.com/questions/20238883/git-diff-and-meld-on-windows>
  + git diff 🡪 changes not staged i.e. diff between working directory and the INDEX
  + git diff -- staged 🡪 staged but not committed i.e. diff between INDEX and HEAD
  + git diff -- cached 🡪 same as above
  + git diff -- HEAD 🡪 changes to tracked files against last revision (staged or not) i.e. diff between HEAD and working directory
  + git diff -- HEAD^ 🡪 same as above but compared to direct ancestor to HEAD
  + git log 🡪 view all commits made in the repo in reverse chronological order (latest first)
    - - p 🡪difference between each commit
    - -2 🡪 limits output to last two entries
    - -- stat 🡪 abbreviated status
    - -- pretty 🡪 changes log file output format (use with oneline, short, full, fuller)q
  + git reset HEAD <filename> 🡪unstage a file
  + git checkout -- <filename> 🡪 discard changes in working directory and revert it to last committed revision
    - *It’s important to understand that git checkout -- [file] is a dangerous command. Any changes you made to that file are gone – you just copied another file over it. Don’t ever use this command unless you absolutely know that you don’t want the file*
    - This command can also be used to retrieve a older version of the file i.e. git checkout <sha> <filename>
* If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with -Xignore-all-space or -Xignore-space-change. The first option ignores changes in any amount of existing whitespace, the second ignores all whitespace changes altogether.
  + **$** git merge -Xignore-all-space whitespace

#### Good Practices

* Setting up a .gitignore file before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository
* Pass the -v option to git commit. Doing so also puts the diff of your change in the editor so you can see exactly what changes you’re committing.)
* On long running branches, it is a **good idea to merge or rebase (?) in from master once every day** so that you don’t have conflicts while merging back
  + When you rebase your branch onto their branch, you tell Git to make it look as though you checked out their branch cleanly, and then did all your work starting from there. That makes a clean, conceptually simple package of changes that someone can review. You can repeat this process again when there are new changes on their branch, and you will always end up with a clean set of changes "on the tip" of their branch.
  + When you merge their branch into your branch, you tie the two branch histories together at this point. If you do this again later with more changes, you begin to create an interleaved thread of histories: some of their changes, some of my changes, some of their changes. Some people find this messy or undesirable.
* Use rebase for short-lived, local branches and ***merges*** for branches in the public repository.
* Write good commit messages, use a pre-defined template
  + <http://chris.beams.io/posts/git-commit/>
  + <http://karma-runner.github.io/1.0/dev/git-commit-msg.html>
* Use a commit message template



#### Bibliography

* <https://www.youtube.com/watch?feature=player_detailpage&v=ZDR433b0HJY#t=2791s>
* https://www.youtube.com/watch?v=4XpnKHJAok8
* <http://gitimmersion.com/>
  + <https://git-scm.com/book/en/v2/Git-Branching-Rebasing#The-Basic-Rebase>
  + <http://stackoverflow.com/questions/16666089/whats-the-difference-between-git-merge-and-git-rebase>
  + <https://blog.sourcetreeapp.com/2012/08/21/merge-or-rebase/>
* <https://nathanj.github.io/gitguide/tour.html> 🡨 UI GUIDE WITH EXAMPLES
* <https://www.atlassian.com/git/tutorials/refs-and-the-reflog>
* <https://www.atlassian.com/git/tutorials/undoing-changes>
* <http://zsoltfabok.com/blog/2012/02/git-blame-line-history/> 🡨 BLAME
* <https://wiki.opennms.org/wiki/Git_Refactoring_Commits> 🡨 REFACTORING COMMITS USING REBASE
* <http://nvie.com/posts/a-successful-git-branching-model/>
* <https://guides.github.com/features/wikis/>
* <https://coderwall.com/p/5cv5lg/generate-your-changelogs-with-git-log>
* <https://jerel.co/blog/2011/07/generating-a-project-changelog-using-git-log>
* <https://gist.github.com/PurpleBooth/109311bb0361f32d87a2#file-readme-template-md>
* http://www.drdobbs.com/architecture-and-design/driving-continuous-integration-from-git/240161383
* https://svenpet.com/2013/03/11/git-continuous-integration-feature-branches/
* https://www.atlassian.com/continuous-delivery/git-hooks-continuous-integration
* https://www.atlassian.com/continuous-delivery/continuous-delivery-workflows-with-feature-branching-and-gitflow
* https://www.atlassian.com/continuous-delivery/ci-friendly-git-repos
* https://devops.com/continuous-integration-doesnt-work/
* <http://stackoverflow.com/questions/37122520/how-to-use-git-on-a-continuous-integration-build-server>
* https://www.letscodejavascript.com/v3/transcripts/lessons\_learned/1
* https://www.git-tower.com/blog/become-a-better-programmer-5-essentials/
* <http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf> 🡪 saved to disk

Upwork 🡪 freelancing

* <http://www.hongkiat.com/blog/automated-php-test/>
* <https://www.smashingmagazine.com/2009/01/50-extremely-useful-php-tools/>
* <http://stackoverflow.com/questions/20755145/deploying-php-site-with-maven-it-cant-be-this-hard>
* Cloud development using GIT
* GIT deployment scripts / automated git push deployments

### Setting P4Merge as the difftool and mergetool

Open ~/.gitconfig (git config --global --edit) and add or change:

[merge]

tool = p4merge

[mergetool "p4merge"]

path = C:\\Program Files\\Perforce\\p4merge.exe

[diff]

tool = p4merge

[difftool "p4merge"]

path = C:\\Program Files\\Perforce\\p4merge.exe