SLS Lecture 5a : Version Control and GIT: Part I"

UC-SLS: (re)Version Control Part I

Slides for UC:SLS Lecture 5

https://git-scm.com/

- "Pro Git book", written by Scott Chacon and Ben Straub
  - Much of the material and images for this lecture are based on or from this book
    - license: https://creativecommons.org/licenses/by-nc-sa/3.0/
- Reference Manual (man pages) https://git-scm.com/docs

Jonathan Appavoo

1

Programmers toolbox so far

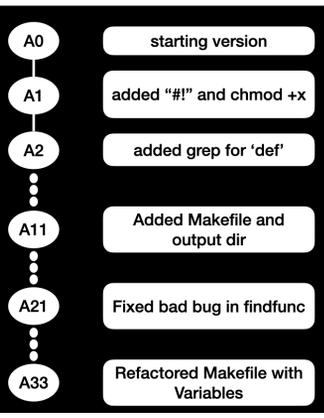
- Shell - Powerful programmable user interface
- Editor - Write and manipulate ASCII files
- Make - Automate the processing of files (transforming/building)
- Now:
  - VCS - capture and work with the "versions" of a collection of files

2

(re)Version Control Systems

- The directories and files of a project quickly take on a life of their own
  - You add new files and directories
  - You modify existing files
  - You move things around
  - You rename things

```
mkdir findfuncs
echo 'find . -name *.py' > findfuncs/findfuncs
```



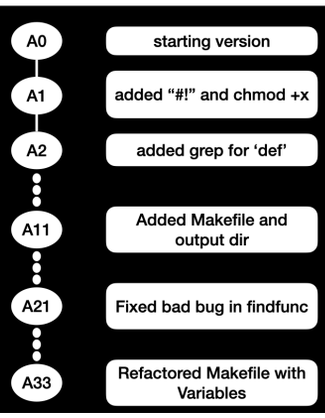
3

(re)Version Control Systems

```
mkdir findfuncs
echo 'find . -name *.py' > findfuncs/findfuncs
```

- The directories and files of a project quickly take on a life of their own
  - You add new files and directories
  - You modify existing files
  - You move things around
  - You rename things
- Tracking and working with this evolution/history/ lineage quick becomes a thing of its own
  - inspecting history, going back to prior versions, comparing versions, creating alternative versions, etc.

Not just source code any project can benefit from the use of a VCS



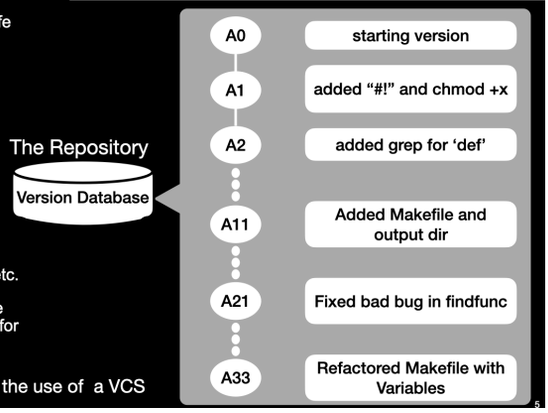
4

(re)Version Control Systems

```
mkdir findfuncs
echo 'find . -name *.py' > findfuncs/findfuncs
```

- The directories and files of a project quickly take on a life of their own
  - You add new files and directories
  - You modify existing files
  - You move things around
  - You rename things
- Tracking and working with this evolution/history/lineage quick becomes a thing of its own
  - inspecting history, going back to prior versions, comparing versions, creating alternative versions, etc.
  - To do this we create a Database that contains all the version information — We call this **The Repository** for a project

Not just for source code any project can benefit from the use of a VCS

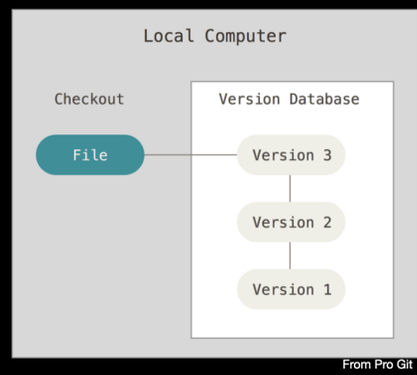


5

A new tool and skill for a developer

Mange your own daily work efficiently with a version control system

- added version points that keep track of what you are doing with comments that help you know what you have done and what you have to do next
- revert selected files back to a previous state
- revert the entire project back to a previous state
- compare changes over time
- create alternative branches to organize your explorations with out disturbing "stable/good" versions



From Pro Git

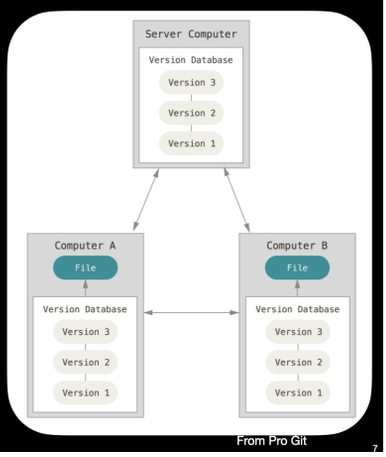
"Using a VCS also generally means that if you screw things up or lose files, you can easily recover."

6

VCS: The foundation for programming as a group endvour

Distributed Version Control

- Use version control databases as a way of publishing your projects
- Allow developers to
  - Clone copies so that they can contribute directly
  - Fork copies so that they can evolve or contribute
- An infrastructure for community based software development and sharing
  - Make it easy for community to review and accept contributions



From Pro Git

7

GIT Overview and Concepts

https://git-scm.com/

- "Pro Git book", written by Scott Chacon and Ben Straub
- Reference Manual (man pages) https://git-scm.com/docs

8

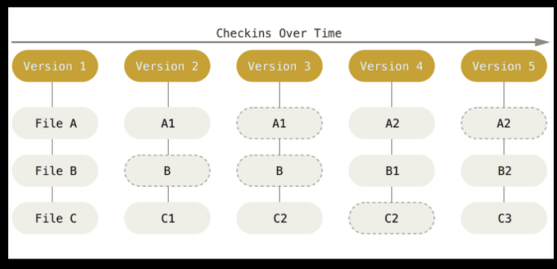
```
GIT(1)                                  Git Manual                                  GIT(1)

NAME
       git - the stupid content tracker

SYNOPSIS
       git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>]
           <command> [<args>]

DESCRIPTION
       Git is a fast, scalable, distributed revision control system with an unusually rich command set that
       provides both high-level operations and full access to internals.

       See gittutorial(7) to get started, then see giteveryday(7) for a useful minimum set of commands. The
       Git User's Manual[1] has a more in-depth introduction.

       After you mastered the basic concepts, you can come back to this page to learn what commands Git
       offers. You can learn more about individual Git commands with "git help command". gitcli(7) manual page
       gives you an overview of the command-line command syntax.

       A formatted and hyperlinked copy of the latest Git documentation can be viewed at
       https://git.github.io/htmldocs/git.html or https://git-scm.com/docs.
```

9

```
GIT(1)                                  Git Manual                                  GIT(1)

NAME
       git - the stupid content tracker

SYNOPSIS
       git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>]
           <command> [<args>]

DESCRIPTION
       Git is a fast, scalable, distributed revision control system with an unusually rich command set that
       provides both high-level operations and full access to internals.

       See gittutorial(7) to get started, then see giteveryday(7) for a useful minimum set of commands. The
       Git User's Manual[1] has a more in-depth introduction.

       After you mastered the basic concepts, you can come back to this page to learn what commands Git
       offers. You can learn more about individual Git commands with "git help command". gitcli(7) manual page
       gives you an overview of the command-line command syntax.

       A formatted and hyperlinked copy of the latest Git documentation can be viewed at
       https://git.github.io/htmldocs/git.html or https://git-scm.com/docs.
```

10

GIT and Snapshots

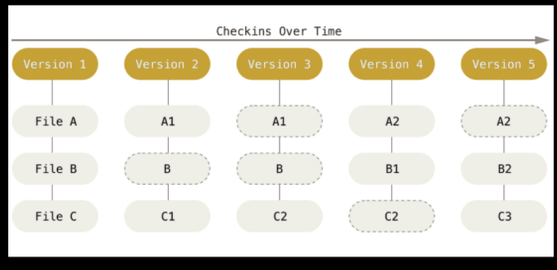


"With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot."

From Pro Git

11

GIT and Snapshots



"With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot."

- With Git you work locally without having to talk to another computer (no need for internet connection)
  - You have the whole data base of version on your computer (the version database)
- GIT uses checksums to track contents of files:

32 digit hash numbers

A SHA-1 hash looks something like this: `24b9da6552252987aa493b52f8696cd6d3b00373`

From Pro Git

12

Git states of a file

A File

hello.c

3 states in git ecosystem

Three states a file can be in

1. **Modified** → edit + save
2. **Staged** → marking it "a modified file"
3. **Committed** → up load

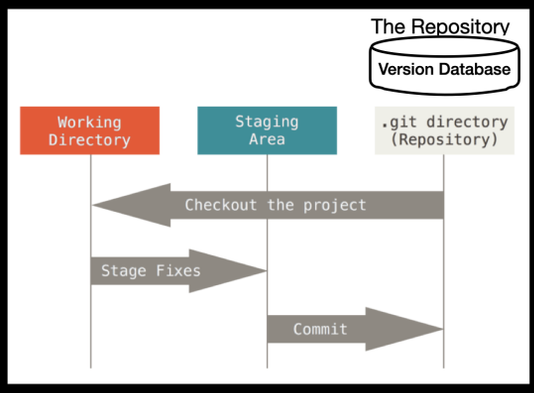
- **Modified** - means that you have changed the file but have not committed it to your database yet.
- **Staged** - means that you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed** - means that the data is safely stored in your local database.

From Pro Git

13

Two new "areas"

- Working Directory (tree): A directory full of our projects subdirectories and files
  - It is a single version that is "checked out" from the database
- Staging Area: A file that we mark what we want to go into our next commit
- .git directory : the database (all of the git stuff). This is what is copied when we "clone" a repository from another computer.



14

The basic Git workflow

1. Modify your projects files and directories
2. Selectively stage your changes you want to be part of your next commit (adding on those to the index)
2. Commit takes what you have staged and stores that as a new snapshot in your database

1. Edit/Modify
2. Stage
3. Commit

A version of a file in the database is "committed"

A modified file that was added to the index is "staged"

A file changed from the last checkout but not staged is "modified"

15

you don't need to commit Every single time

Remember

- Git is first and foremost a command line tool. It was designed to be used within a terminal from the shell.
  - An extra tool to make your life as a developer more productive and safer
    - Making the lifetime of your project's files and directories as versions that you can manage (and publish)
- So get comfortable knowing what to do with git as a command line too

16

Git Basics - The core stuff

- First time setup
- Manuals and help
- Getting a Git Repo
- Recording Changes
- Viewing history of commits
- Undoing things
- Working with Remotes

17

First Time setup (Pro Git 1.6)

- Git has some settings that you need to configure: `git config`
  - User specific values stored in ~/.gitconfig/*
    - `git config –global`
  - Repository specific values
    - `git config –local`
  - Show settings
    - `git config –list –show-origin`

18

First Time setup (Pro Git 1.6)

- Values we care about:
  - Identity:
    ```
    $ git config --global user.name "John Doe"
    $ git config --global user.email johndoe@example.com
    ```
    - So our commits and messages are properly identified
  - Editor:
    ```
    $ git config --global core.editor emacs
    ```
    - So that when git needs us to write a messages it will open the right editor

19

Git help and Manuals (Pro Git 1.7)

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

git status → status of git

20

Getting a Repo (Pro Git 2.1)

Two ways:

1. Create a new repo from a directory not under version control
2. Clone an exiting repository from somewhere else

21

Getting a Repo (Pro Git 2.1)

Create a new repo from a local directory that is not under VC

```
$ cd /home/user/my_project
$ ls -la
$ git init
$ ls -la  # notice something new?

$ git status
$ git add <files>
$ git status
$ git commit -m 'Initial project version'
$ git status
```

git log → commit 한걸 다볼수 있음

22

Getting a Repo (Pro Git 2.1)

Clone an exiting repository from somewhere else

Like a Git base site on the internet the "hosts" repositories Eg. Github or Gitlab (we will talk more about these later)
https://github.com/kornia/kornia

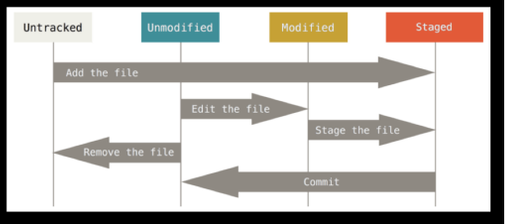
```
$ git clone <url>

# http (generally not encouraged on pubic services like github)
$ git clone https://github.com/kornia/kornia.git

# ssh (secure but requires you to generate and register a ssh key)
$ git clon git@github.com:kornia/kornia.git
```

23

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- Your friend:
  `$ git status`
- Lets follow the life cycle of a file
- Birth:
  ```
  $ echo 'My Project' > README
  $ git status
  ```
- Tracking (get putting in git's control)
  ```
  $ git add README
  $ git status
  ```

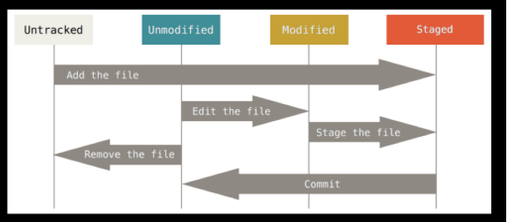


24

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- Modify an exiting tracked file
  - Lets add a new target to the Makefile
    `$ git status`
- Stage it too:
  `$ git add Makefile`
  `$ git status`
- Modify it again
  `$ git status`

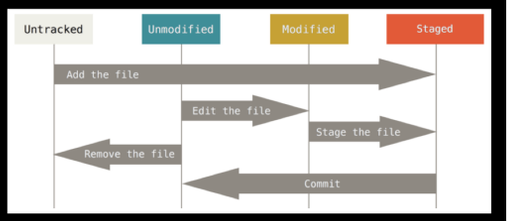


25

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- Modify an exiting tracked file
  - Lets add a new target to the Makefile
    `$ git status`
- Stage it too:
  `$ git add Makefile`
  `$ git status`
- Modify it again
  `$ git status`



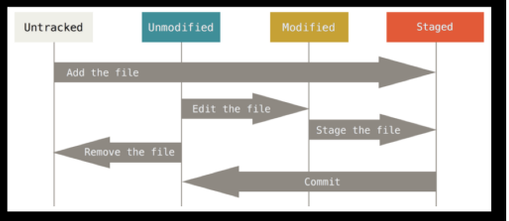
- ???
  - Adding a file is adding an exact version if you want to update the stage you have to add again

26

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- Git ignore
  - Very useful
    `$ git status`
    `$ echo "*~" > .gitignore`
    `$ git status`
  - Lets add the outputs so we don't accidentally put them in the repo

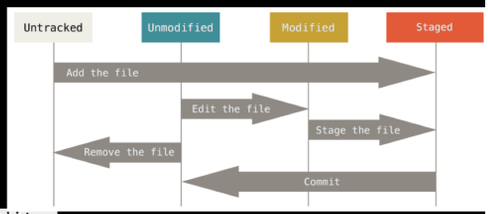


27

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- See what's changed
  ```
  $ git status
  $ git diff
  $ git diff –staged
  ```
- Let's get commit
  ```
  $ git status
  $ git commit # start editor
  $ git commit –m "message" # avoid editor
  ```
- Skip staging
  ```
  $ git status
  $ git commit –a –m "message" # stage and commit all changes
  ```



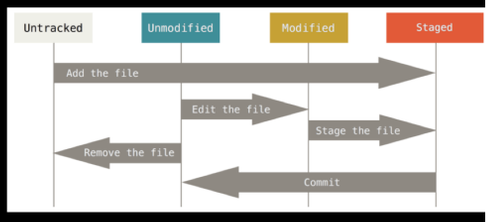
28

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- Removing files
  - Removing from git
    ```
    $ rm <file>
    $ git status
    $ git rm <file>
    $ git status
    $ git commit
    $ git status
    ```
  - Removing from git but not working directory (eg you forgot to add to .gitignore)
    ```
    $ git rm –cached <file>
    ```

patterns can be used to remove matching files

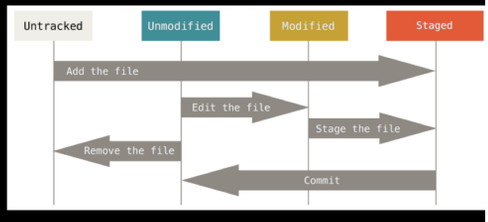


29

Git Recording Changes (Pro Git 2.2)

The lifecycle of the status of your files

- Renaming/Moving files
  - Removing from git
    ```
    $ git mv <file_from> <file_to>
    $ git status
    $ git commit
    ```
  - equivalent to:
    ```
    $ mv <file_from> <file_to>
    $ git rm <file_from>
    $ git add <file_to>
    ```



30

Git Viewing history of commits (Pro Git 2.3)

```
$ git log
```

- Git log is a very powerful command with many options here are some useful examples:

```
$ git log -<n>
$ git log -p
$ git log --oneliner
$ git log --stat
$ git log --relative-date
$ git log --oneliner --shortstat
$ git log --oneliner --graph
$ git log --since=2.weeks
$ git log --grep “Fix”  # regex matching commit message
$ git log --oneline -p -S "open" # -S string added or removed in code
```

Lots and lots of options… view by author, until, before, range, etc…

31

Git: Undoing Things (Pro Git 2.4)

Be careful some undoes are permanent

- Ammending: git commit —amend

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

- Unstaging:

```
$ git restore --staged <file>
```

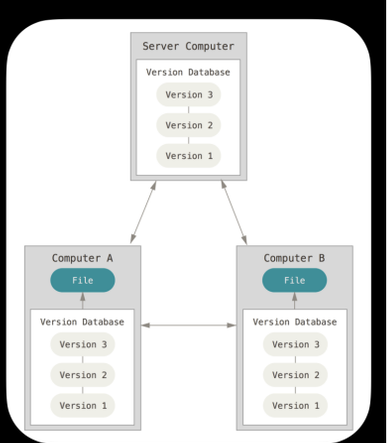
- Unmodifying — revert changes in working directory (CAREFULL!)

```
$ git restore <file>
```

32

Git Remotes (Pro Git 2.5)

- Remote repositories are version of your that are hosted on other computers on the internet or other networks
- There can be many copies
- Collaborating with others (and yourself) means managing these remote repositories
  - Push and Pull data to and from them to share work



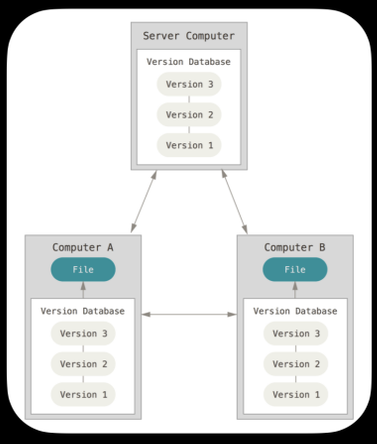
33

Git Remotes (Pro Git 2.5)

```
$ git remote
$ git remove -v
```

Git clone automatically adds the remote origin but we can explicitly add remotes as follows

```
$ git remote add cs400 https://cs400-gitlab.bu.edu/jappavoo/kornia.git
```



34

Git Remotes (Pro Git 2.5)

Fetching and Pulling

```
$ git fetch <remote>
```

Gets all the data from that remote you can now inspect it, merge it, modify it, etc.

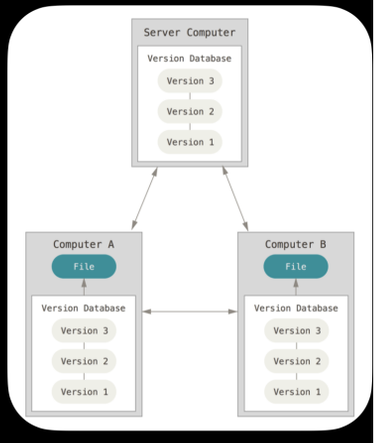
Default is to get data from origin if you cloned repo

Fetch only down loads it does not try and merge data into your local version

If your branch is tracking a remote branch then you can download and merge with one step. Note cloning repos sets this up automatically

```
$ git pull
```

This will make more sense when talk about branches



35

Git Remotes (Pro Git 2.5)

Pushing, inspecting, renaming, removing

To share your work with a remote you "push" your work to the remote

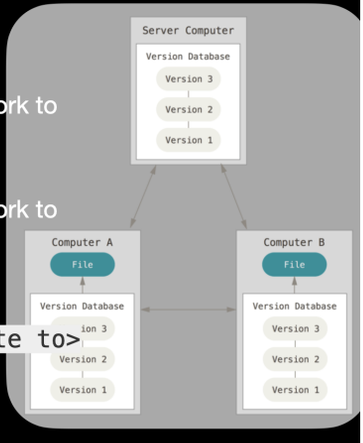
```
$ git push <remote> <branch>
```

To share your work with a remote you "push" your work to the remote

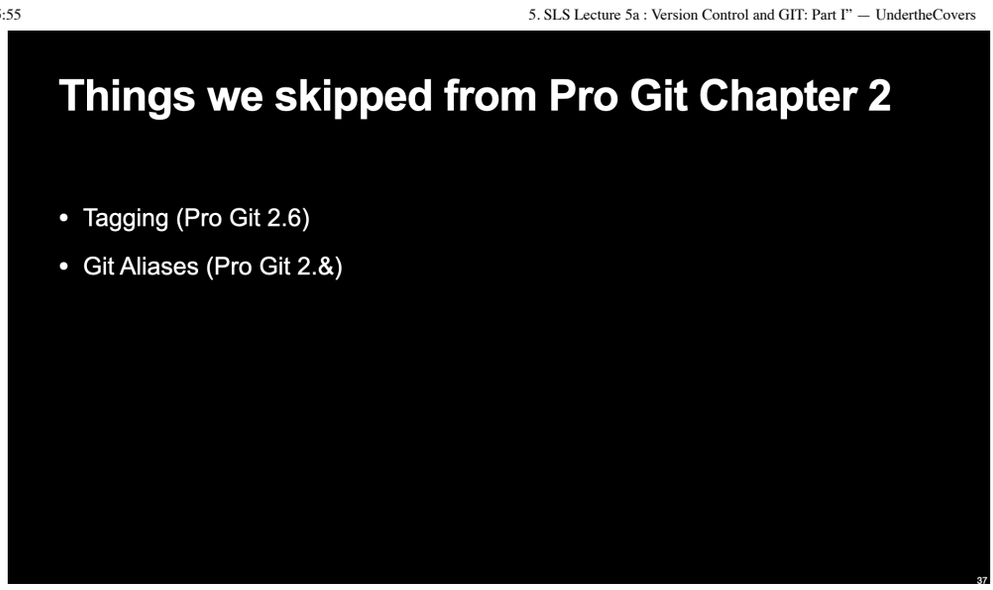
```
$ git remote show <remote>
```

renaming and removing

```
$ git remote rename <remote from> <remote to>
$ git remote remove <remote>
```



36



By Jonathan Appavoo