

Version 1.4
05/12/23



Prerequisites

- Have a recent version of Git downloaded and running on your system
 - For Windows, suggest using Git Bash Shell interface
- If you don't already have one, sign up for free GitHub account at
<http://www.github.com>
- Workshop docs are in <https://github.com/skilldocs/git4>
- Labs doc for workshop

<https://github.com/skilldocs/git4/blob/main/git4-2-labs.pdf>

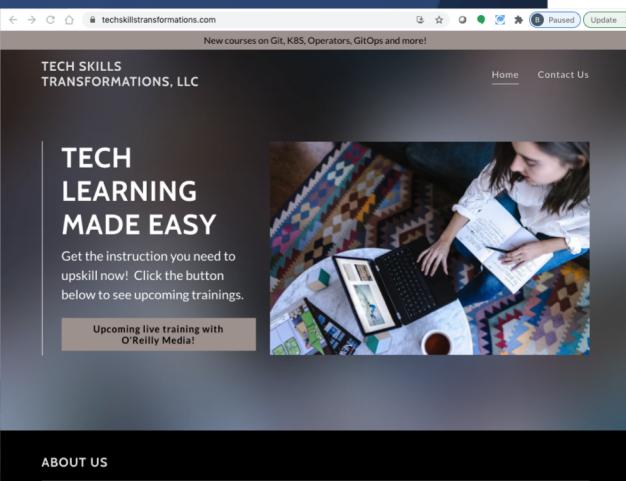


Git In Four Weeks

Week 2

Brent Laster
Tech Skills Transformations

About me



- Founder, Tech Skills Transformations LLC
- R&D DevOps Director
- Global trainer – training (Git, Jenkins, Gradle, CI/CD, pipelines, Kubernetes, Helm, ArgoCD, operators)
- Author -
 - OpenSource.com
 - Professional Git book
 - Jenkins 2 – Up and Running book
 - Continuous Integration vs. Continuous Delivery vs. Continuous Deployment mini-book on O'Reilly Learning
- <https://www.linkedin.com/in/brentlaster>
- @BrentCLaster
- GitHub: brentlaster



Professional Git Book

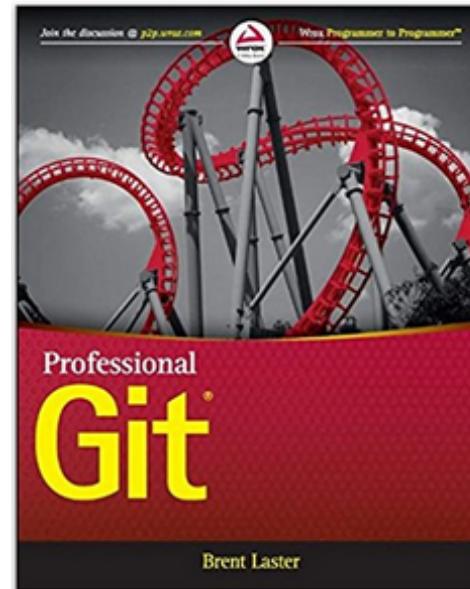
- Extensive Git reference, explanations,
- and examples
- First part for non-technical
- Beginner and advanced reference
- Hands-on labs

Professional Git 1st Edition

by [Brent Laster](#) (Author)

7 customer reviews

[Look inside](#)



© 2023 Brent C. Laster &

Tech Skills Transformations LLC



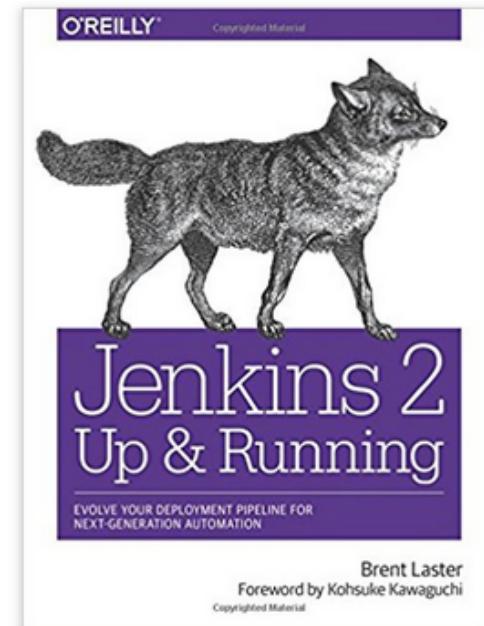
Jenkins 2 Book

- Jenkins 2 – Up and Running
- “It’s an ideal book for those who are new to CI/CD, as well as those who have been using Jenkins for many years. This book will help you discover and rediscover Jenkins.” *By Kohsuke Kawaguchi, Creator of Jenkins*

★★★★★ 5 customer reviews

#1 New Release in Java Programming

[Look inside](#) ↴

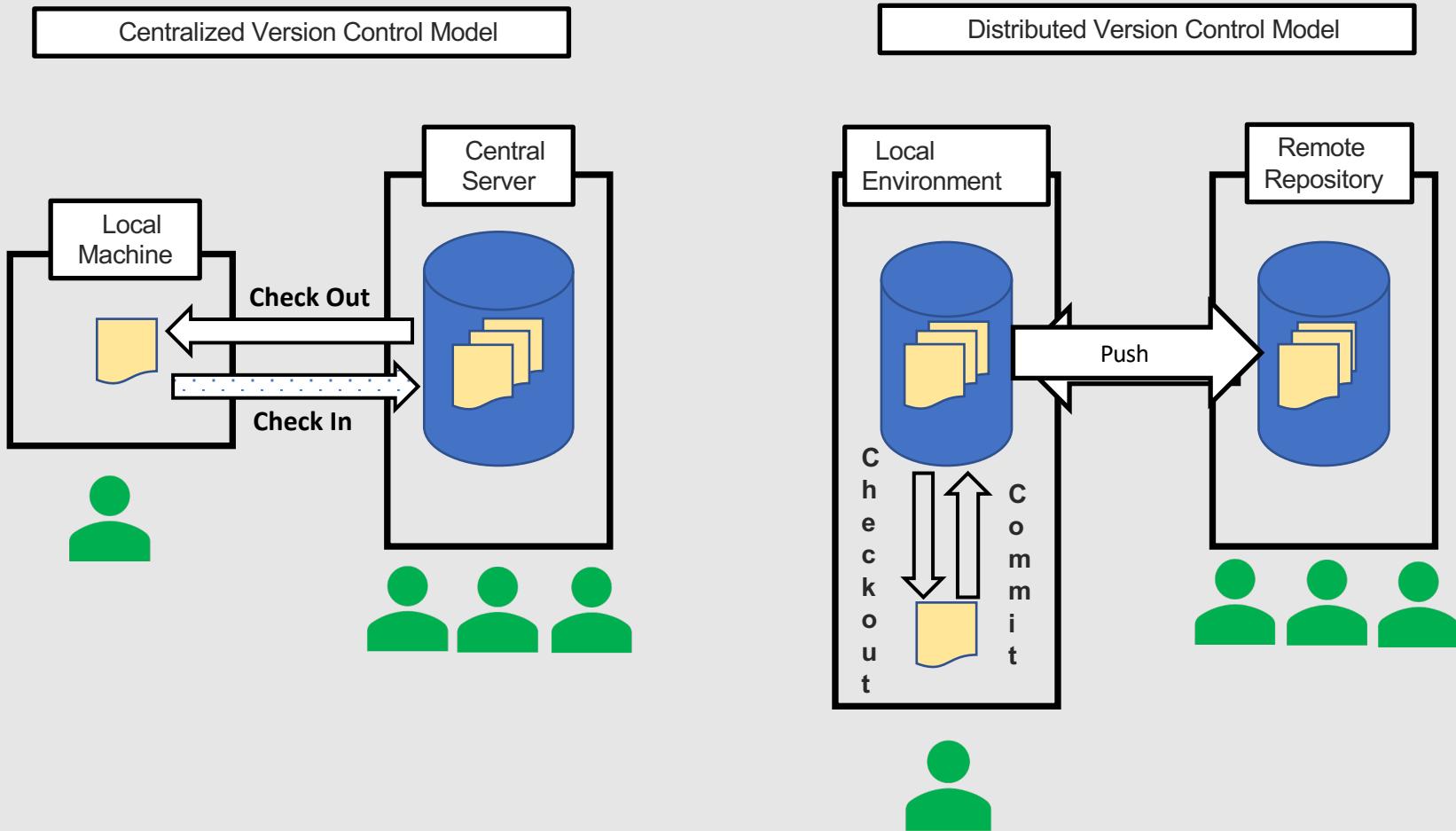


Review



Centralized vs. Distributed VCS

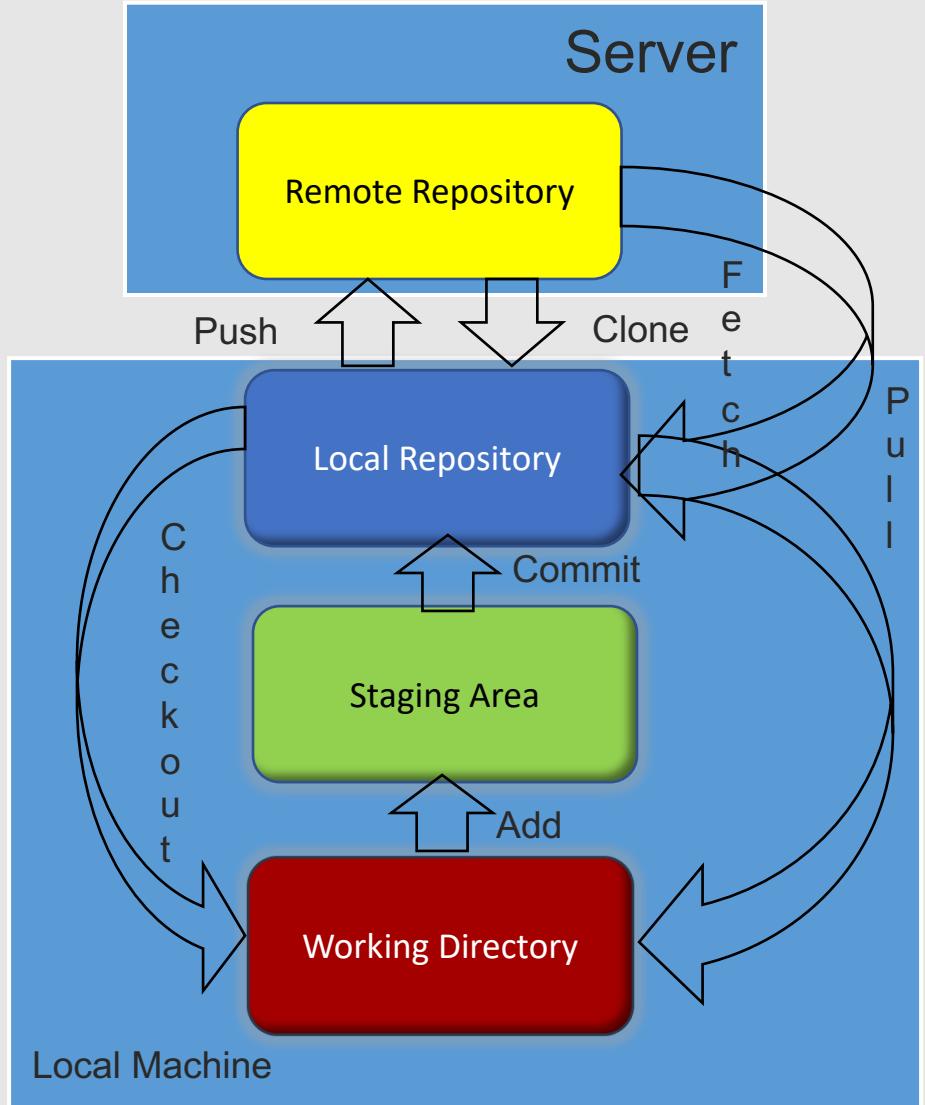
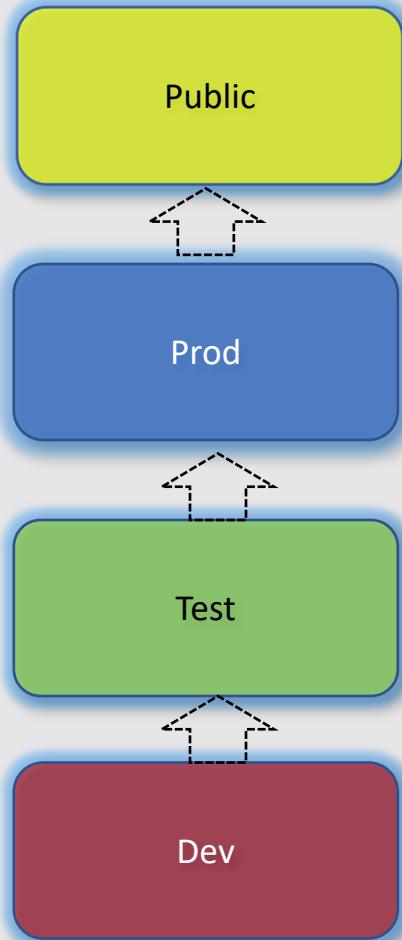
8





Git in One Picture

9

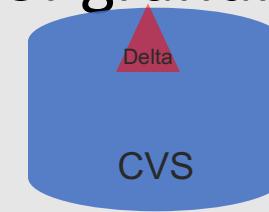




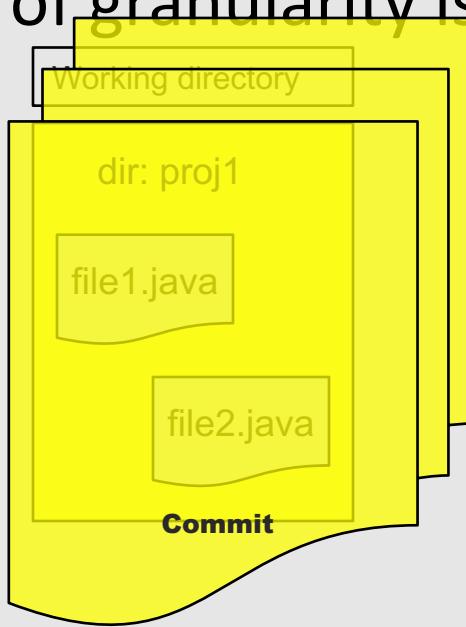
Git Granularity (What is a unit?)

10

- In traditional source control, the unit of granularity is usually a file



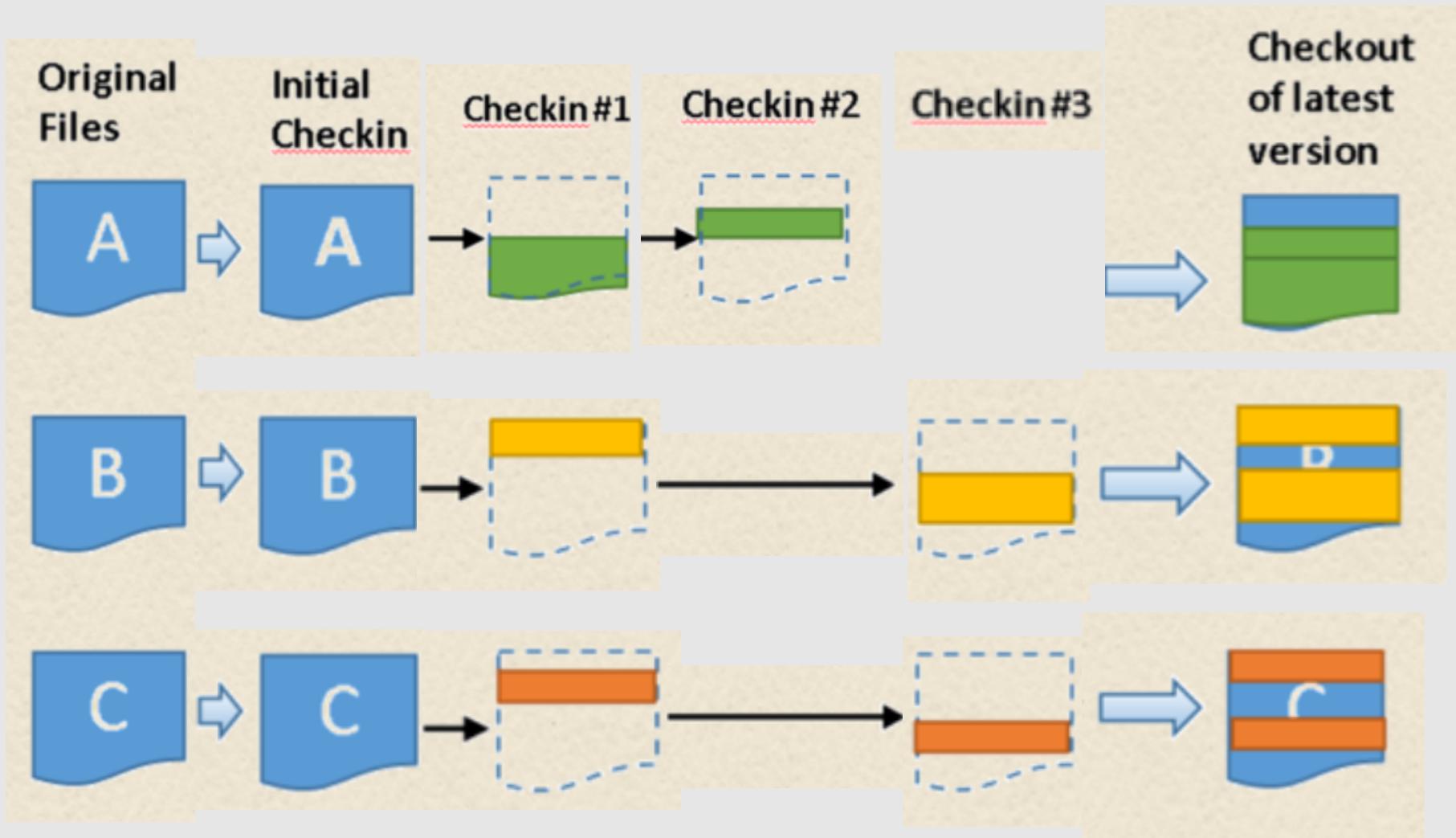
- In Git, the unit of granularity is usually a tree





Delta Storage Model

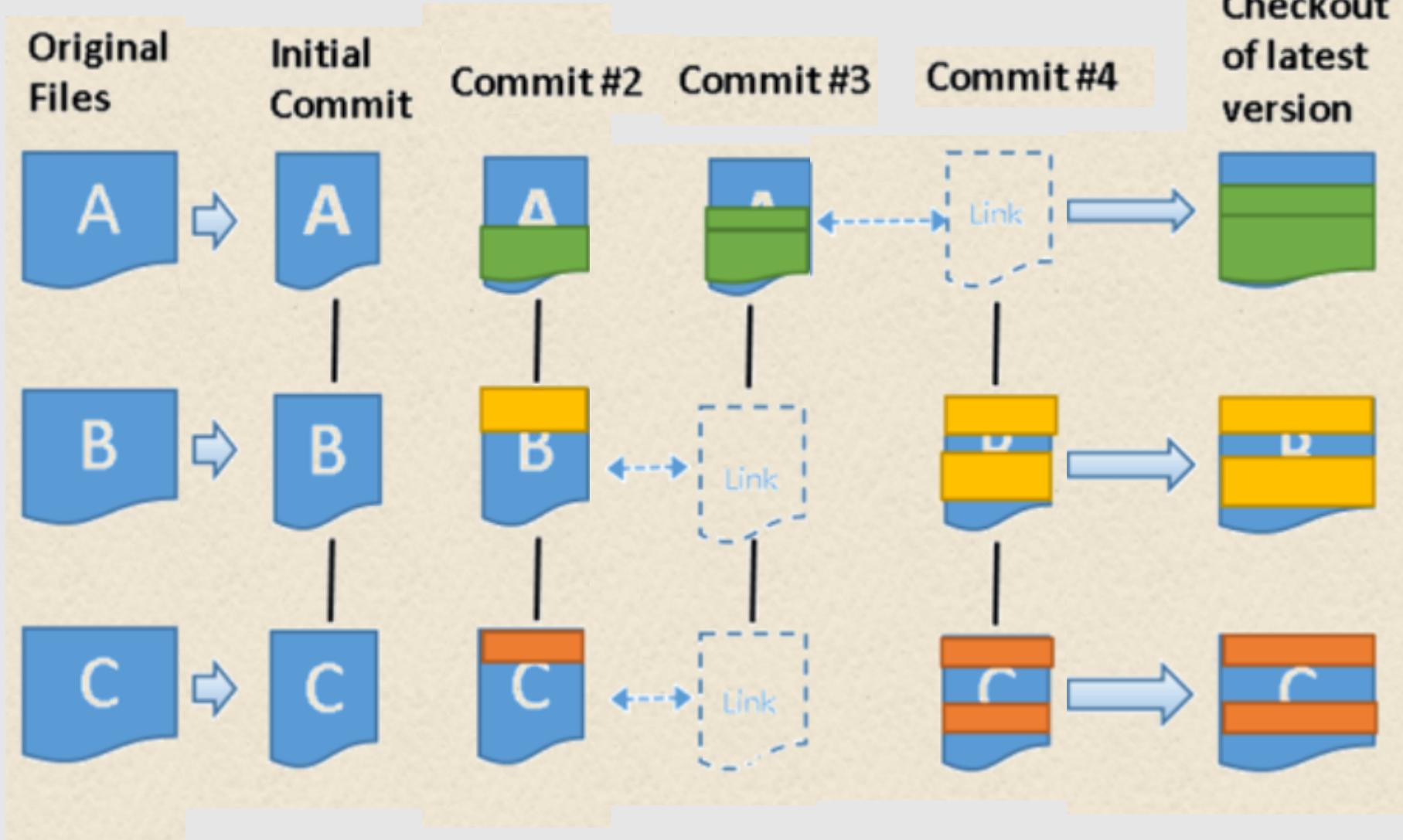
11





Snapshot Storage Model

12





Initializing a Repo and Adding Files (commands)

13

- **git init**
 - For creating a repository in a directory with existing files
 - Creates repository skeleton in .git directory
- **git add**
 - Tells git to start tracking files
 - Patterns: git add *.c or git add . (.= files and dirs recursively)
- **git commit**
 - Promotes files into the local repository
 - Uses –m to supply a comment
 - Commits everything unless told otherwise

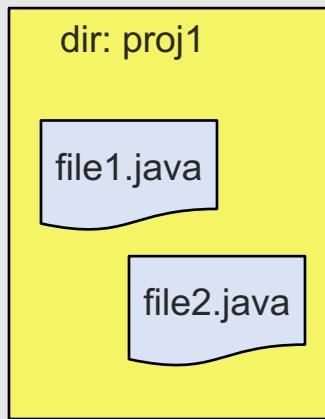


Git and File/Directory Layouts

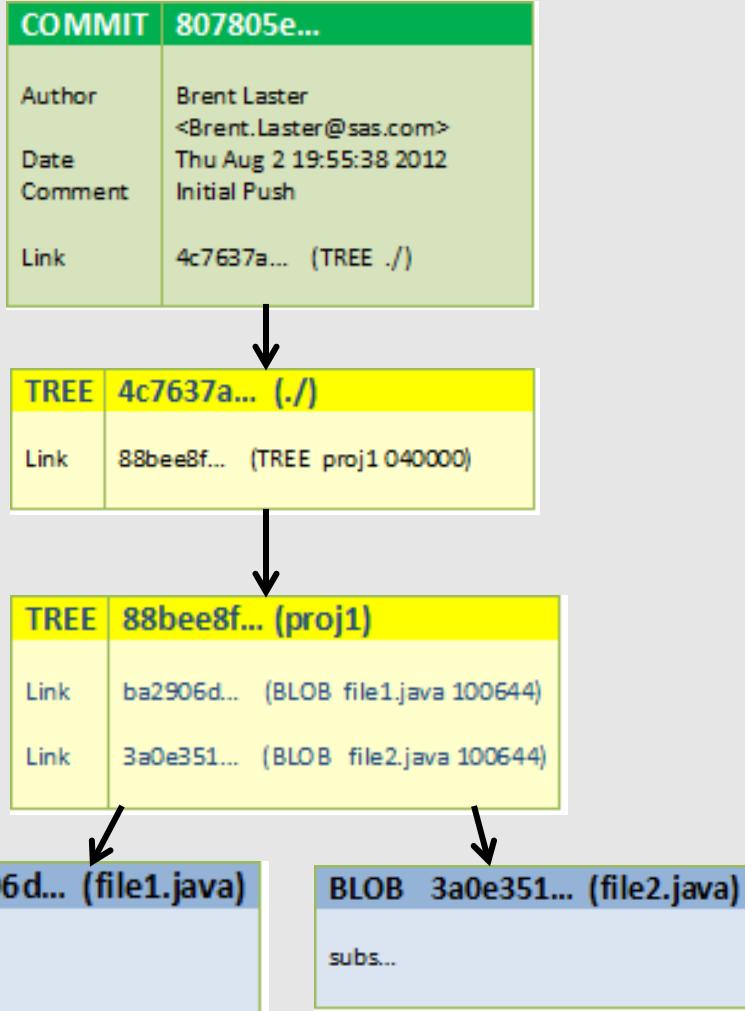
14

Local Repository File Layout

Working directory



Git snapshot



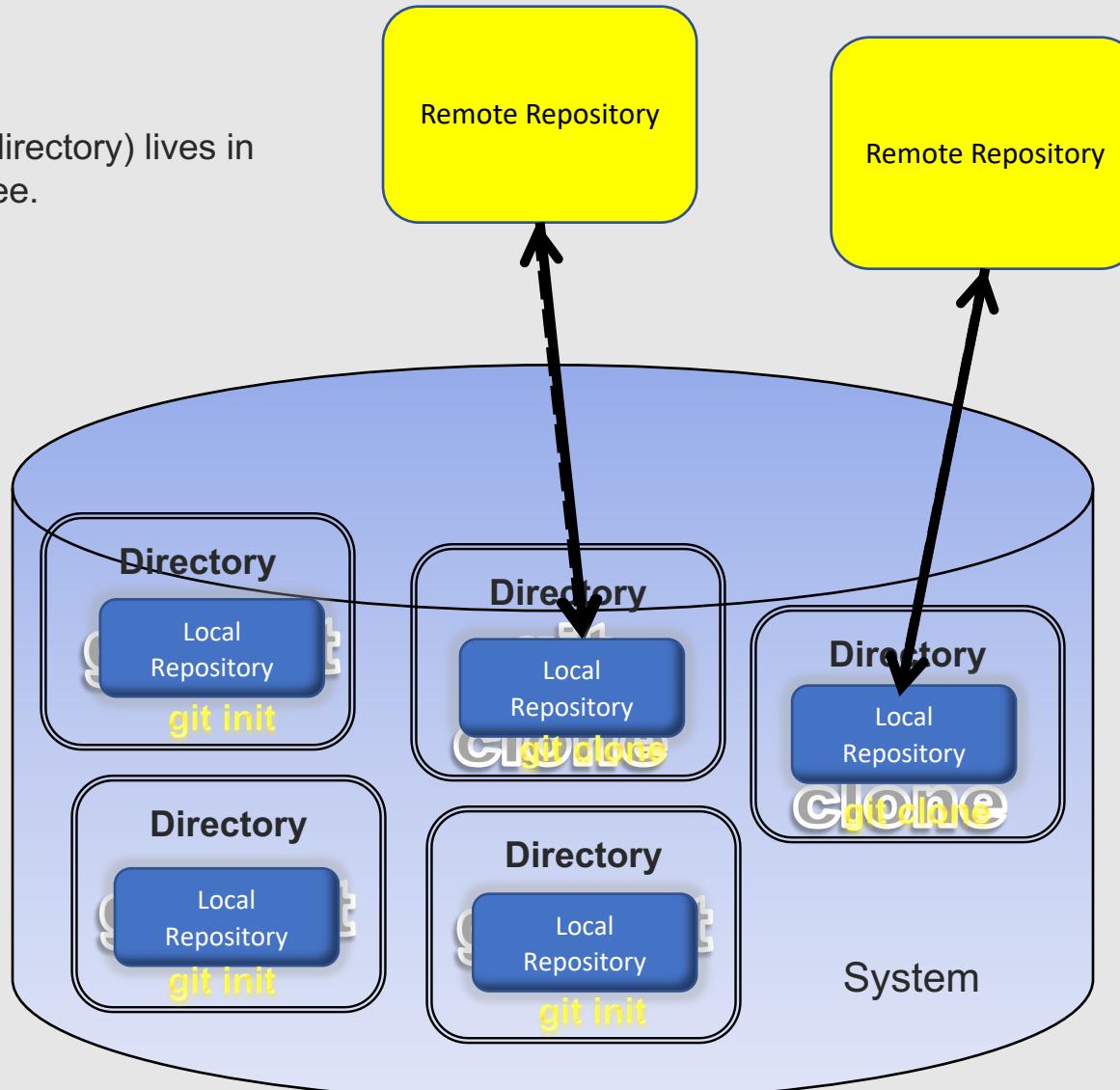
```
$ tree.sh - * .git
COMMIT_EDITMSG
HEAD
config
description
hooks
  applypatch-msg.sample
  commit-msg.sample
  post-commit.sample
  post-receive.sample
  post-update.sample
  pre-applypatch.sample
  pre-commit.sample
  pre-rebase.sample
  prepare-commit-msg.sample
  update.sample
index
info
exclude
logs
HEAD
refs
  heads
    main
objects
  3a  0e351dc84e32abec5d4dd223c5abdabd57b7f5
  4c  7637a4b66aefc1ee877ea1afc70610f0ee7cc
  80  7805ea7ae9fdf1b06e876af6e9a69a349b52a3
  88  bee8f0c181784d605eabe35fb04a5a443ae6b7
  ba  2906d0666cf726c7eaadd2cd3db615dedfdf3a
info *
pack *
refs
heads
  main
tags *
```



Starting Work with Git – Multiple Repositories

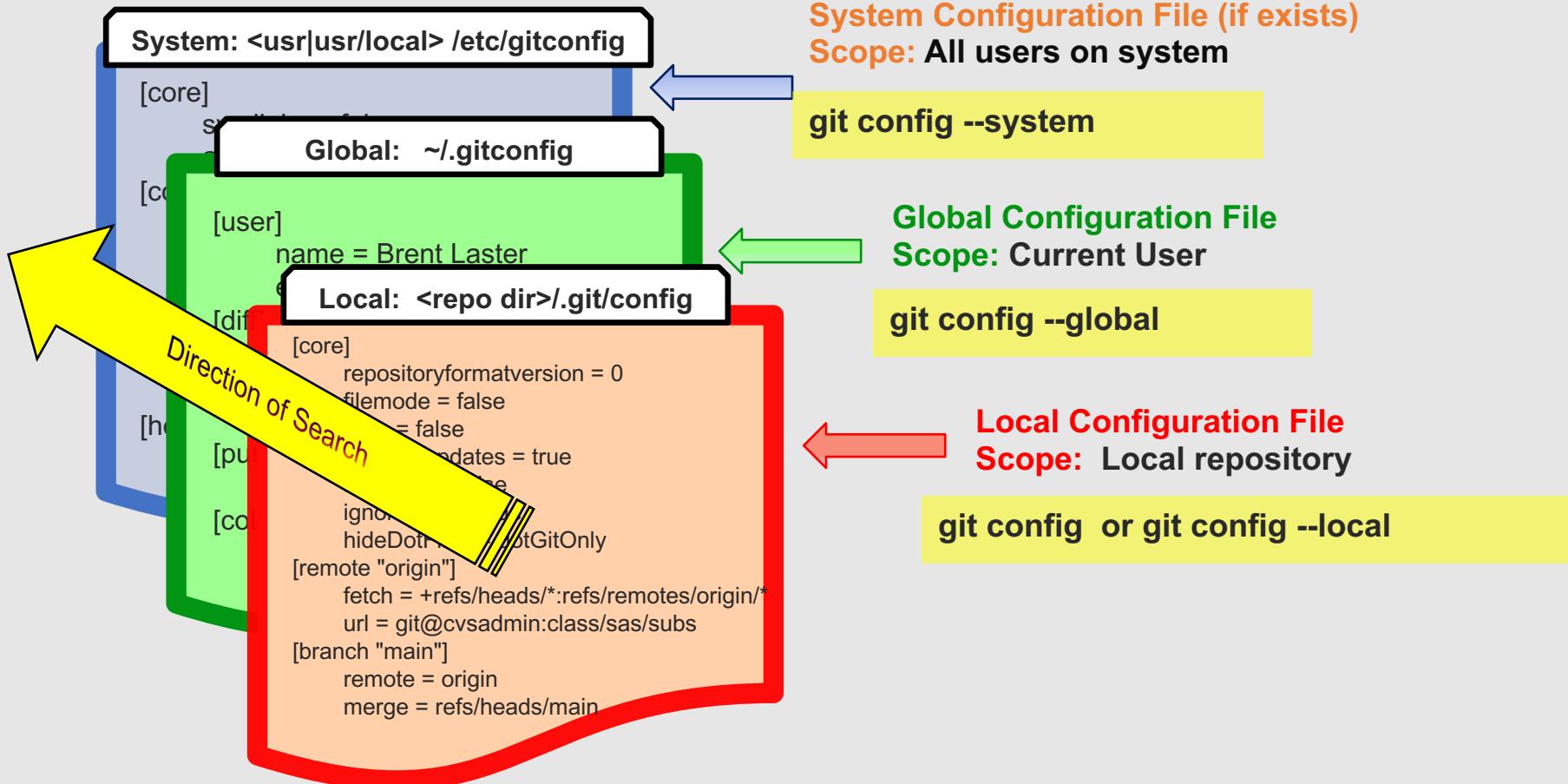
15

The repository (.git directory) lives in the local directory tree.

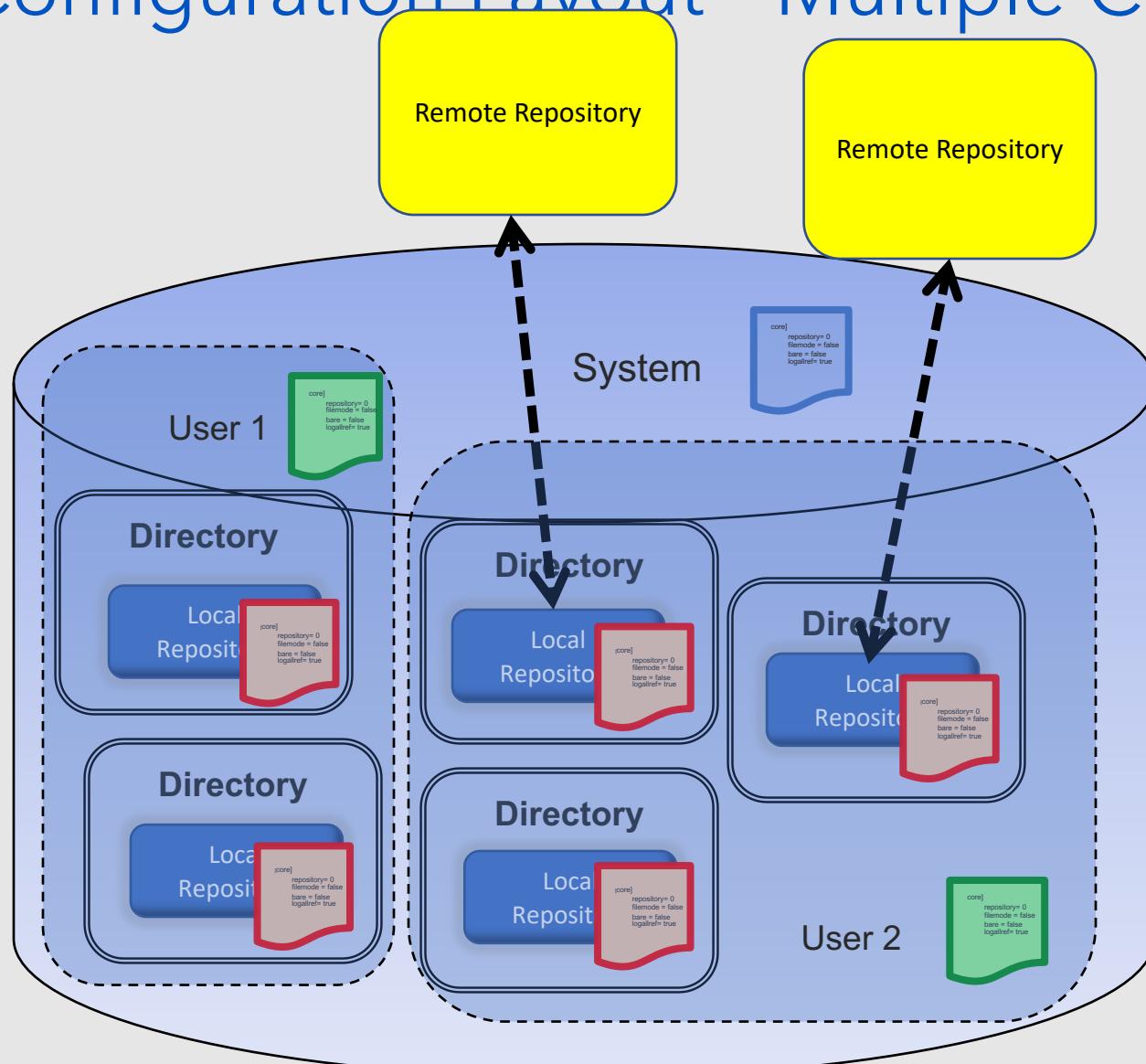




Git Configuration Files



Git Configuration Layout - Multiple Configs





How Git reports status on files

18

Files in your working directory can be in one of two main states:

- ***Tracked***

- Were in the last snapshot (last commit)
 - Can be
 - Unmodified - same as what's in Git
 - Modified - different from what's in Git
 - Staged
- OR -

- ***Untracked***

- Everything else
- Not in last snapshot
- Not in staging area

3 questions to think about for determining status

- Is Git aware of the file (is it in Git)? tracked or untracked
- What is in the staging area?
- What is the relationship of the latest version in Git to the version in the working directory?



Showing Differences

19

- Command : `git diff`
- Default is to show changes in the working directory that are not yet staged.
- If something is staged, shows diff between working directory and staging area.
- Option of `--cached` or `--staged` shows difference between staging area and last commit (HEAD)
- `git diff <reference>` shows differences between working directory and what `<reference>` points to – example “`git diff HEAD`”



Git log examples

20

- `git log --pretty=oneline --max-count=2`
- `git log --pretty=oneline --since='5 minutes ago'`
- `git log --pretty=oneline --until='5 minutes ago'`
- `git log --pretty=oneline --author=<your name>`
- `git log --pretty=oneline --all`
- `git log --oneline --decorate`



Git Aliases

21

- Allow you to assign alternatives for command and option strings
- Can set using
 - `git config alias.<name> <operation>`
- Example
 - `git config --global alias.co checkout`
 - After, “git co” = “git checkout”
- Example aliases (as they would appear inside config file)

[alias]

`co = checkout`

`ci = commit`

`st = status`

`br = branch`

`hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short`



Tags in Git

22

- Command: `git tag`
- Two types
 - Lightweight – like regular tag, pointer to specific commit
 - Annotated – stored as full object in the database
 - » Checksummed
 - » Contain full email and date, tagger name, and comment
 - » Created by `git tag -a <tag> -m <message>`
- Create a tag
 - `git tag <tagname> <hash>`
- Show tags
 - `git tag` (lists tags out)
 - `git show <tag>` - shows detail



New Content



Removing files

24

- Command: `git rm`
- What it does:
 - Removes it from your working directory (via `rm <file>`) so it doesn't show up in tracked files
 - Stages removal
- Then you do the commit to complete the operation
- If you have a staged version AND a different modified version, git will warn you
 - Use the `-f` option to force the removal
- Can remove just from the staging area using `--cached` option on `git rm`
- Can provide files, directories, or file globs to command

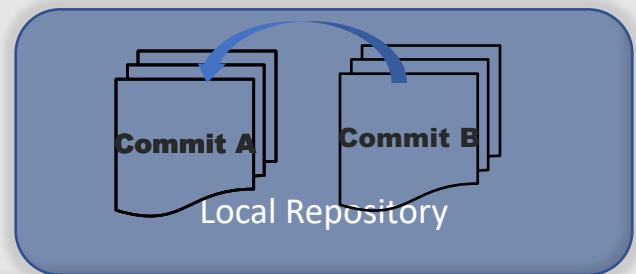
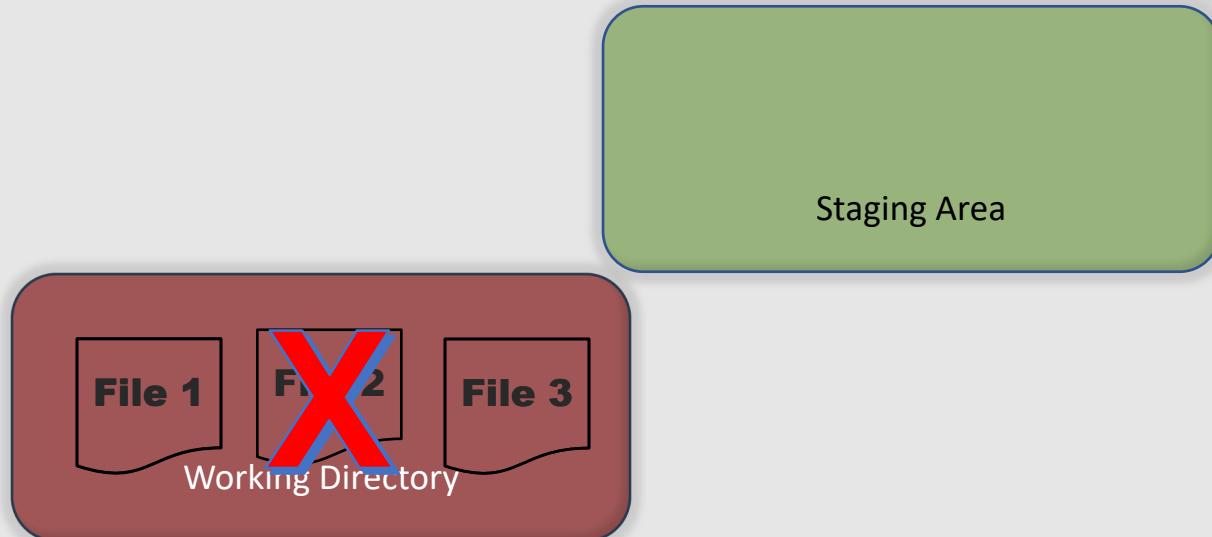


Git Workflow for operations like rm

25

1. Git performs the operation in the working directory
2. Git stages the change
3. User commits to finalize the operation in the repository.

git rm file2
git commit -m “finalize rm”





Rolling back/undoing - Reset and Revert

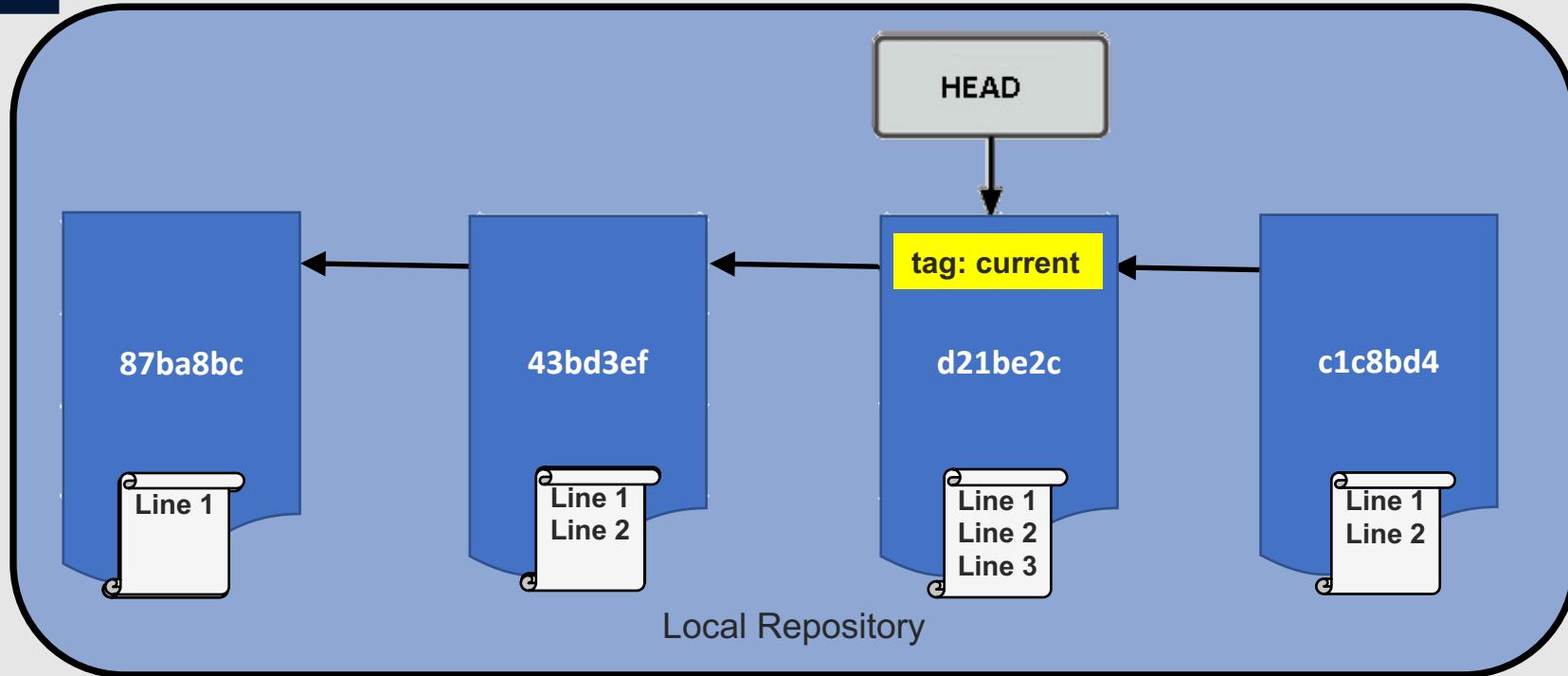
26

- Reset -- allows you to “roll back” so that your branch points at a previous commit ; optionally also update working directory to that commit
- Use case - you want to update your local environment back to a previous point in time; you want to overwrite or a local change you’ve made
- Warning: --hard overwrites everything
- Revert -- allow you to “undo” by adding a new change that cancels out effects of previous one
- Use case - you want to cancel out a previous change but not roll things back
- Note: The net result of using reset vs. revert can be the same. If so, and content that is being reset/revert has been pushed such that others may be consuming it, preference is for revert.



Reset and Revert

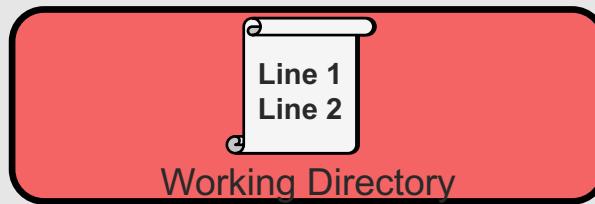
27



`git reset --hard 87ba8bc`

`git reset current~1 [--mixed]`

`git revert HEAD`

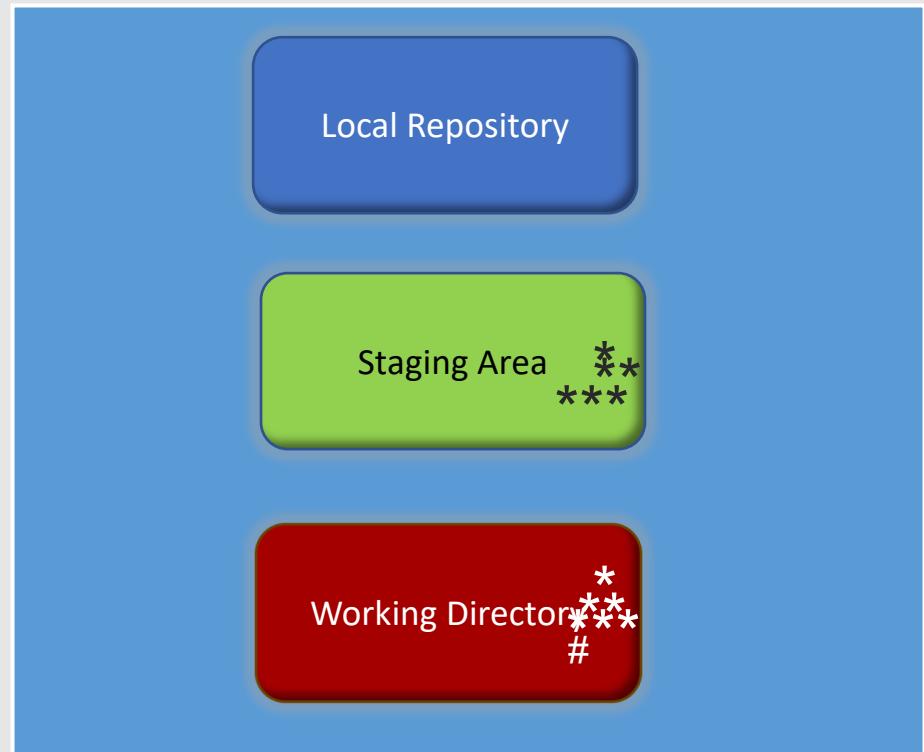
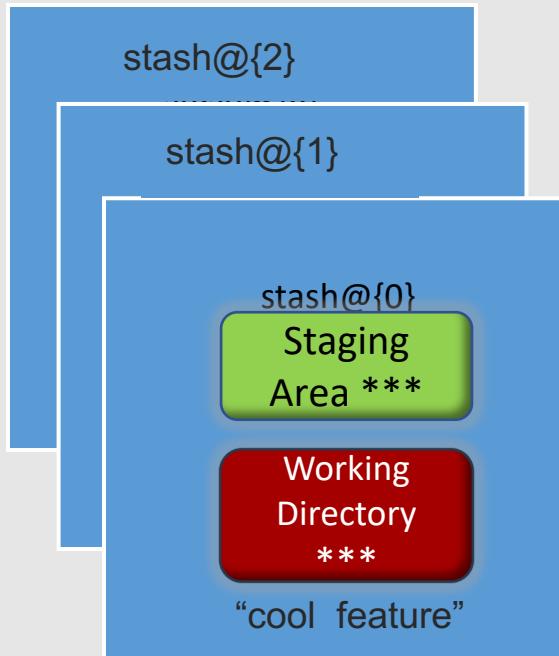




Git Stash

- Keep a backup queue of your work
 - command: **git stash [push]**
 - saves off state
 - use **git stash pop** or **git stash apply** to get old state back

\$ git stash push stash@{0} "cool feature"





Renaming Files

29

- Git doesn't track metadata about renames, but infers it
- Git mv command renames a file locally and stages the change
- Then you commit it
- Git will show “renamed” in status after a mv
- The mv command is equivalent to:
 - mv (old local file) (new local file)
 - git rm old file
 - git add new file



What is a branch in source control?

30

- Line of development
- Collection of specific versions of a group of files tagged in a common way
 - `cvs rtag -a -D <date/time> -r DERIVED_FROM -b NEW_BRANCH PATHS_TO_BRANCH`
- End result is I have an easy “handle” to get all of the files in the repository associated with that identifier – the branch name
 - `cvs co -r BRANCH_NAME PATHS`
- So – what you end up with in your working directory when you check out a branch is a set of specific versions of the files from the group, or a ...



Snapshot! What is a snapshot (in GIT)?³¹

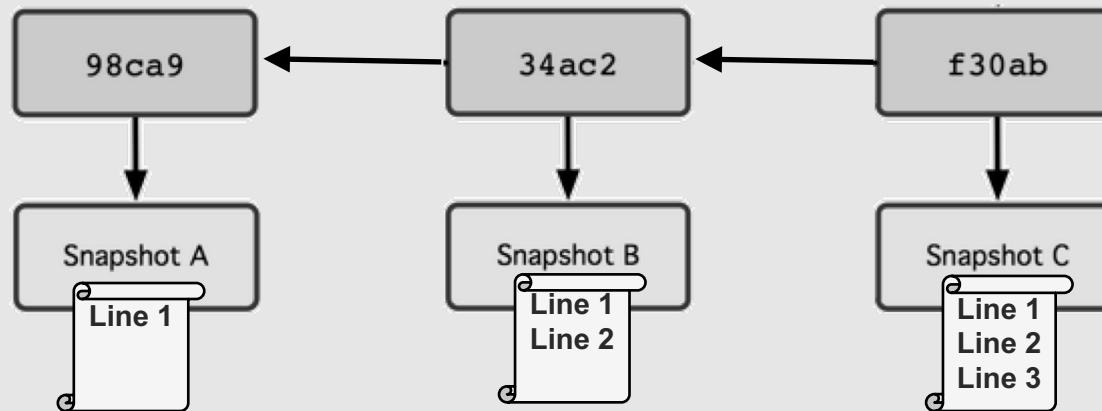
- Line of development associated with a specific change
- Collection of specific versions of a group of files associated with a specific commit
- End result is I have a “handle” to get all of the versions of files in the repository associated with that commit – handle = SHA1 for that commit
- What you end up with in your working directory when you check out a branch is a set of specific versions of the files from the group.



Branches

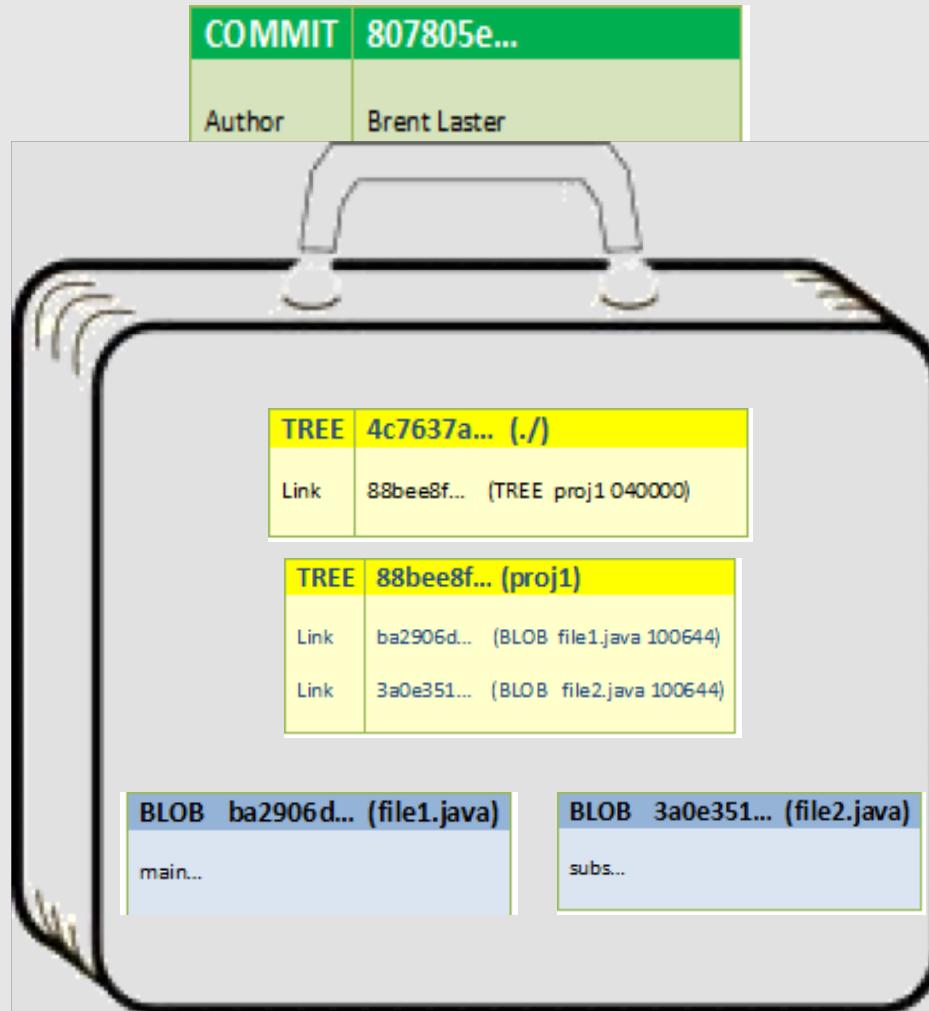
32

- Remember, git stores data as a series of snapshots, not deltas or changesets
- Commits point to snapshots – and the commit that came before them





Commit SHA1's are a handle to a snapshot³³

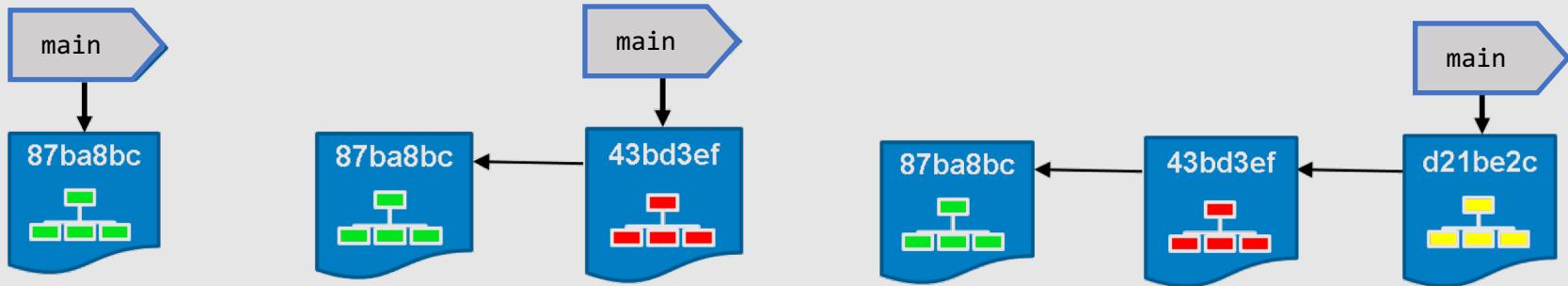




Lightweight Branching

34

- A branch in git is simply a lightweight, movable pointer to a commit
- Default branch is named main
- As you initially make commits, you're given a branch pointer that points to the last commit you made. Every time you commit, it moves forward automatically.



- A branch in Git is just a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and easy as writing 41 bytes to a file (40 characters and a newline).
- And, because we're recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do.



Creating and using a new branch

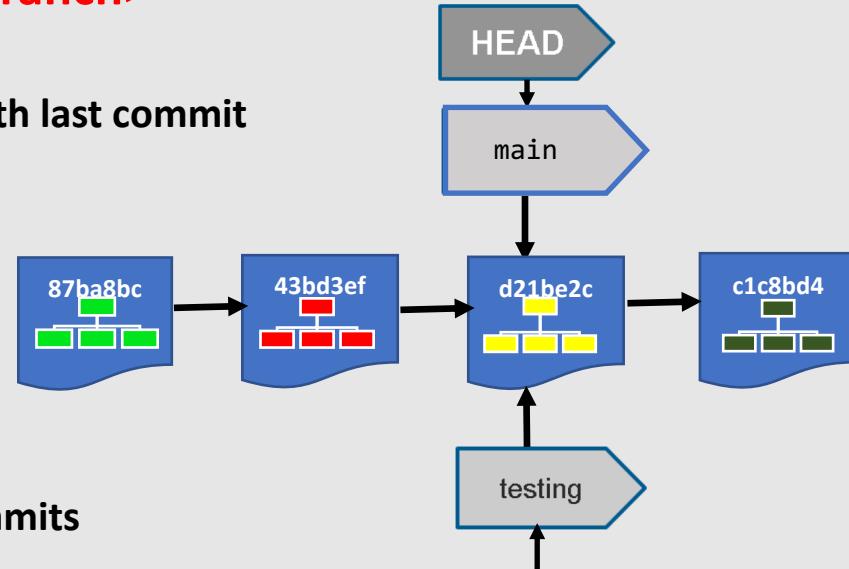
35

- Create a branch : **git branch <branch>**
 - Creating a new branch creates a new pointer

Git keeps a special pointer called HEAD that always points to the current branch.

git branch testing

- Change to a branch: **git checkout <branch>**
 - Moves HEAD to point to <branch>
 - Updates working directory contents with last commit from <branch> - if existing branch



git checkout testing

- Branch pointers advance with new commits



Switching between Branches

36

Command: git checkout <branch>

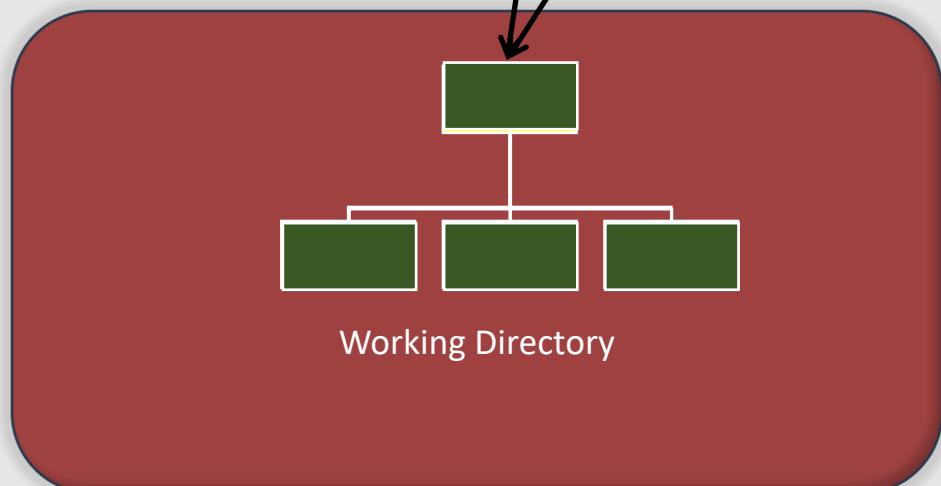
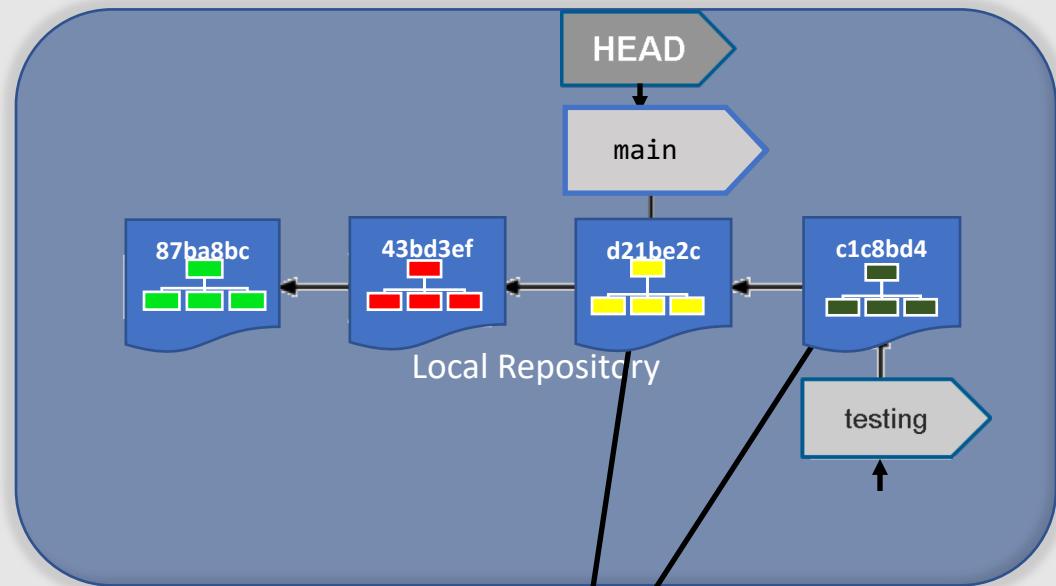
`git checkout main`

- Does three things
 - Moves HEAD pointer back to <branch>
 - Reverts files in working directory to snapshot pointed to by <branch>
 - Updates indicators

`git checkout testing`

`git checkout main`

`git checkout testing`





Which Branch am I on/in?

37

- Command: `git branch <no arguments>`
- “*” is indicator of which one you’re on
- Prompt will also change in some configurations
 - If it doesn’t, can be setup.



"Topic and Feature Branches"

38

- Topic Branches

- Term for a temporary branch to try something out
- Easy to try things in or come back to later
- Generally, create simple name based on type of work you're going to try – i.e. web_client
- Maintainer of a project may “namespace” these – as in adding initials on the front – i.e. abc/web_client
- Create just as any other branch

```
$ git branch abc/web_client
```

```
$ git checkout -b web_client
```

- Feature Branches

- Term for a branch to develop a feature
- Intended for limited lifetime
- Merge back into main line of development

Lab 4 - Working with Branches

Purpose: In this lab, we'll start working with branches by creating a new branch and making changes on it.



Merging Branches

- Command: `git merge <branch>`
- Relative to current branch
- Current branch is branch being merged into
- `<branch>` is branch being merged from
- Ensure everything is committed first



Merging: What is a Fast-forward?

41

- Assume you have two branches as below
- You want to merge hotfix into main (so main will have your hotfix for future development)

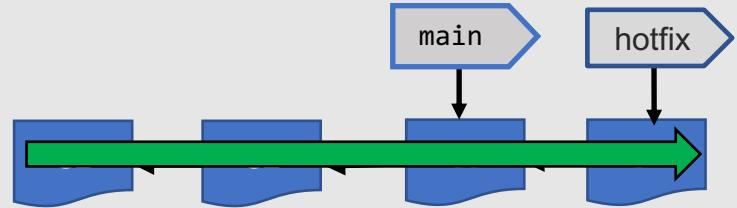
```
$ git checkout main  
$ git merge hotfix
```

Updating f42c576..3a0874c

Fast Forward

README | 1-

1 files changed, 0 insertions(+) 1 deletions (-)



About “Fast Forward” – because commit pointed to by branch merged was directly “upstream” of the current commit, Git moves the pointer forward

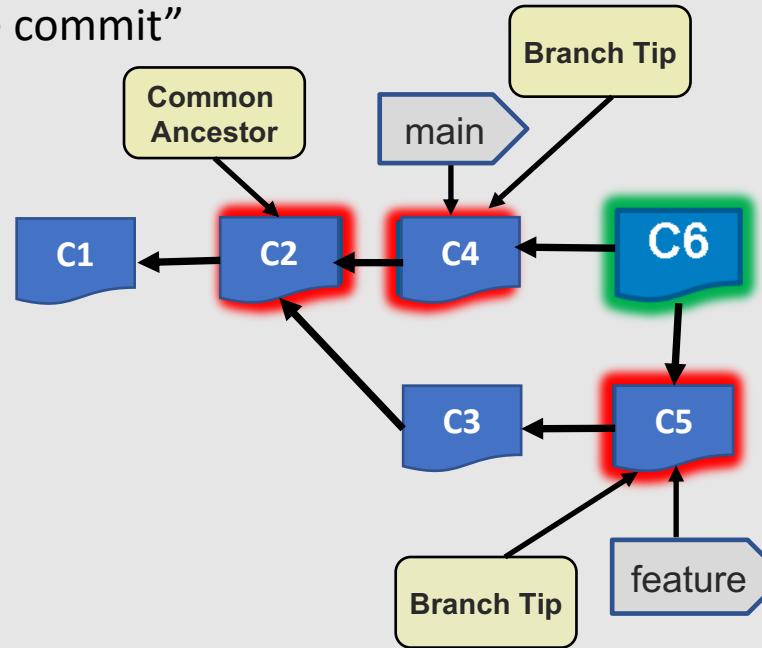
(Both branches were in the same line of development, so the net result is that main and hotfix point to the same commit)



Merging: What is a 3-way Merge?

42

- Assume branching scenario below
 - main and feature branches have both diverged (changed) since their last common ancestor (commit/snapshot)
- Intent is to change to main and merge in feature
- Current commit on target branch isn't a direct ancestor of current commit on branch you're merging in (i.e. C4 isn't on the same line of development as C5)
- Git does 3-way merge using common ancestor
- Instead of just moving branch pointer forward, Git creates a new snapshot and a new commit that points to it called a “merge commit”



```
$ git checkout main  
$ git merge feature
```



Merge Conflicts

43

- If you encounter a merge conflict during a merge operation
 - Will get CONFLICT message
 - Git pauses merge in place
 - To see which files are unmerged, use git status – will see “unmerged: or both modified: <filename>”
 - Adds <<<< and >>> markers in the file
- After resolution, run git add and then commit
- Can also run git mergetool to resolve graphically



Merge Scenario - Branches with Conflicts ⁴⁴ 44

```
$ git checkout main  
$ git merge feature  
$ git status
```

Changes to be committed:

modified: File 1

modified: File 3

Unmerged paths

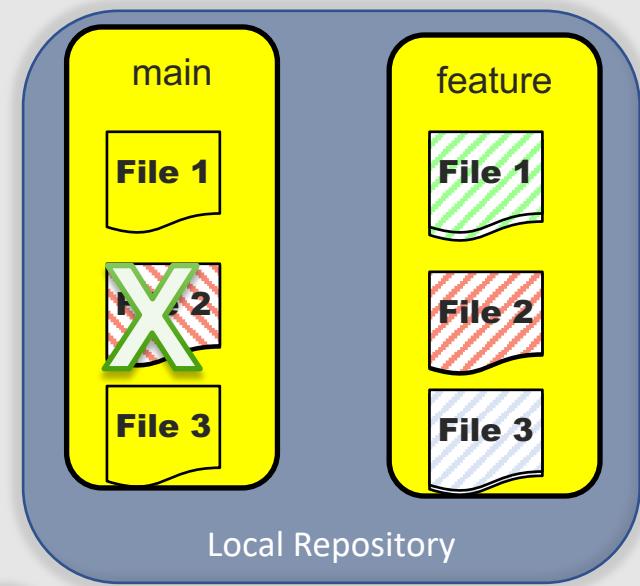
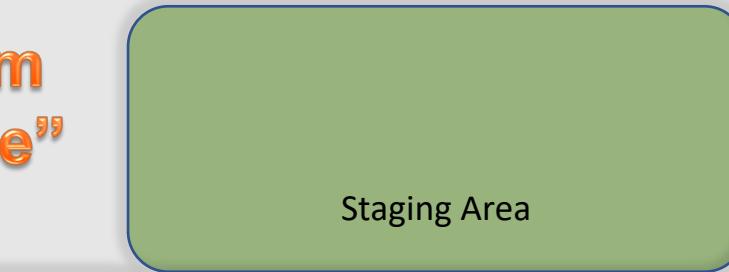
both modified: File 2

[fix conflicts]

```
$ git add .
```

```
$ git commit -m
```

“finalize merge”



Lab 5 - Practice with Merging

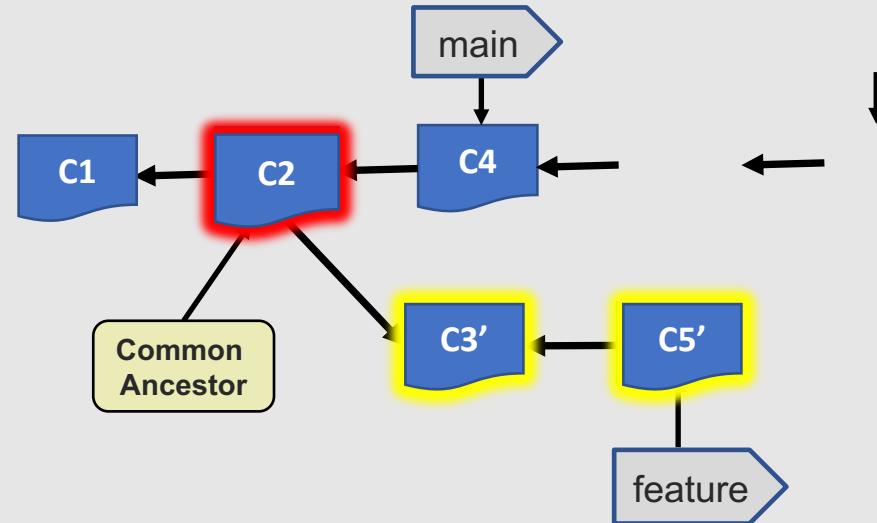
Purpose: In this lab, we'll work through some simple branch merging



Merging: What is a Rebase?

46

- Rebase – take all of the changes that were committed on one branch and replay (merge) them (one at a time in sequence) on another one.
- Process:
 - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
 - Gets the diff introduced by each commit of the branch you are on, saving them to temporary files
 - Applies each change in turn
 - Moves the branch to the new rebase point

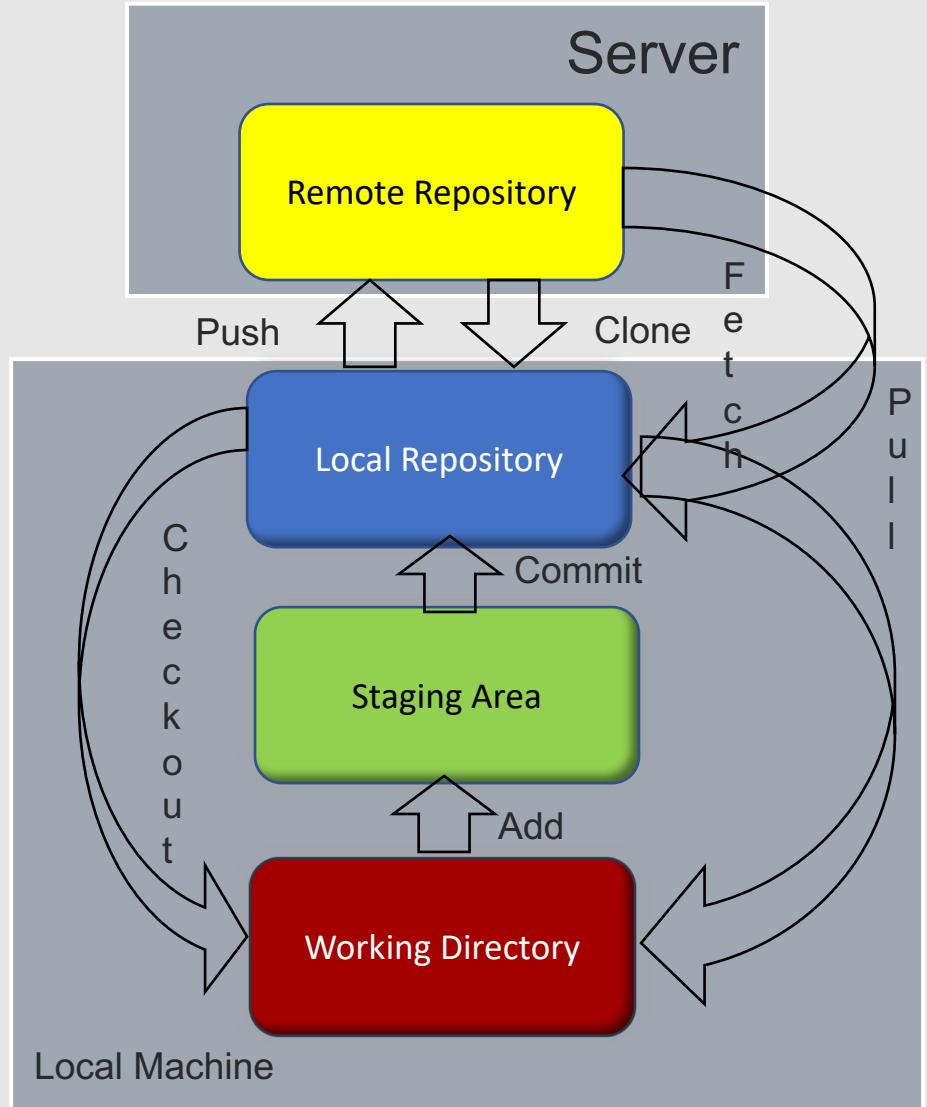
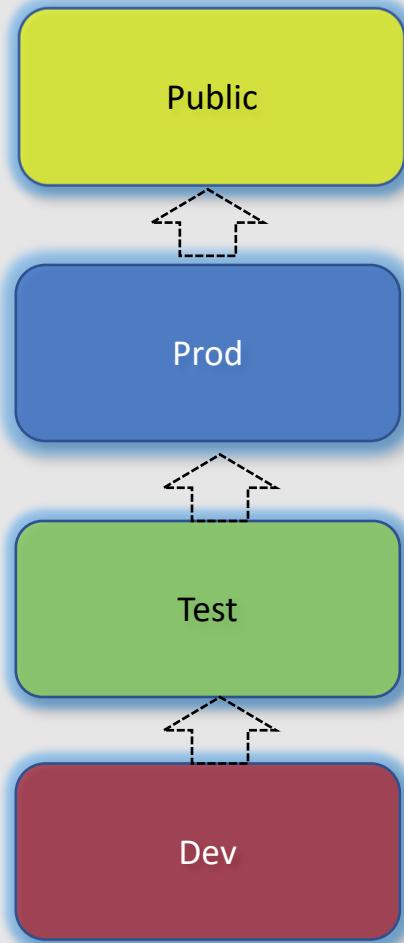


```
$ git checkout feature  
$ git rebase main
```



Git in One Picture

47





Cloning a Remote Repo to get a Local Repo

48

- Command: **git clone**
 - Use to get a copy of an existing repository from a system
 - Syntax: `git clone <url> <optional new name>`
 - Clone
 - » Git pulls nearly all data from cloned area – files, history, etc.
 - » Creates a directory with the project name
 - » Checks out working copy of latest version
 - » Does a bit more than fetch and pull – tracks local branch to remote branch
 - After cloning, have full access to files, histories, etc.



Git Remote References “remotes”⁴⁹

49

- The term “remote” can either refer to:
 - An actual remote repository
 - A handle to such a repository
- For local use, Git provides handles/aliases/nicknames that map to a remote repository location
- The default one of these is “origin”
- The reference “origin” is automatically setup (mapped) for you when you clone
- Example: origin = <https://github.com/<path>>
- This mapped name is what you use in local commands to tell git you want to push/pull/fetch/clone to/from the mapped remote repository
- Example: git pull origin <branch>



Git Remote Repositories

50

- Remote repositories
 - Think “server-side”
 - Versions of projects hosted on network or internet
 - Push or pull from (as opposed to checkout/add/commit)
 - Generally have read-only or read-write access
 - Handle + url
- Shared with multiple users as opposed to one user for local repository
- Multiple protocols for data transfer
 - Local – shared folder, easy for collaboration, slow, not for widespread use
 - SSH – standard with authentication, no anonymous access
 - Git – fastest, but no authentication
 - Http - easy, but inefficient
 - Https – easy and authenticated (temporarily)



Working with Handles to Remotes

51

- Seeing what you have access to
 - Git remote (show handle by default, -v shows url)
 - “origin” – default name for a handle that git gives to remote you cloned from
 - Can be ssh or https URL
 - » `git@github.com:techupskills/calc3.git`
 - » `https://github.com/techupskills/calc3.git`
 - Git remote show [handle] – shows extended info about remote handle
- Adding a remote
 - Think of adding a handle to an existing remote repository
 - Git remote add [handle] [url]
 - » Example : `git remote add remote2 https://github.com/userid/projectpath`
- Renaming remote handle
 - Git remote rename <old> <new>
 - Renames remote branches too
- Removing a handle
 - Git remote rm (reference)



Working with Remotes via Handles

52

- Retrieve latest from git remote (getting what you don't have)
 - Git fetch handle) as in git fetch origin2
 - Note that you are getting the full repository (history, branches, basically everything)
 - Fetch pulls the data to the local repository – it doesn't merge or disturb your local content
- Retrieve latest and merge
 - Git pull
 - By default, git clone automatically sets up to track changes on main branch
 - Pull does a “fetch and merge”
- Pushing back to the server (remote)
 - Git push (handle) (branch)
 - Only works if you have correct access and remote content hasn't been changed
 - If remote content has been changed, then would have to fetch latest and merge (like cvs up-to-date check)



Git Remotes Example

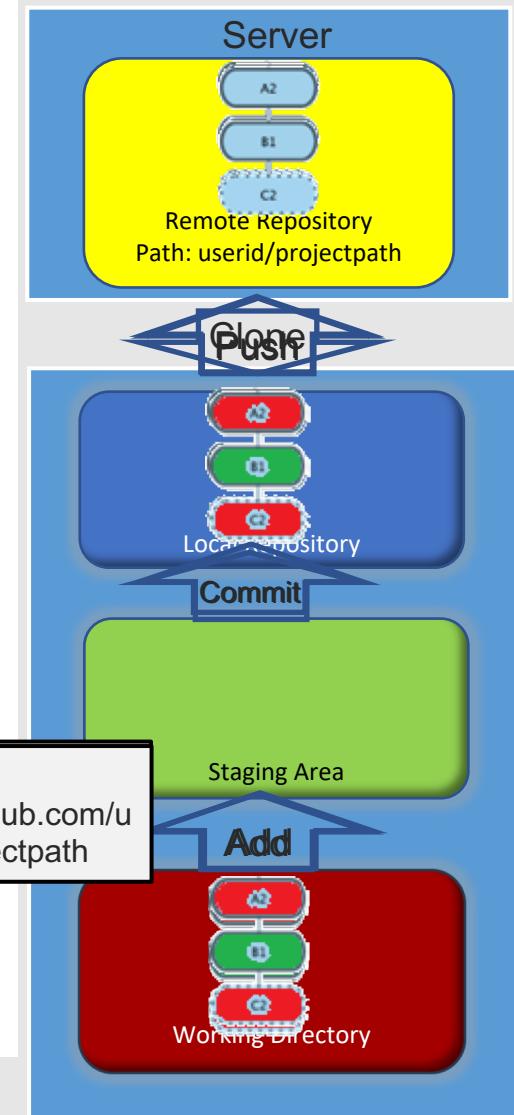
53

```
$ git clone https://github.com/userid/projectpath
$ git remote -v
origin https://github.com/userid/projectpath(fetch)
origin https://github.com/userid/projectpath (pull)

$ <edit File A and File C>
$ git commit –am “...”
$ git push origin main
(default is origin and main so could just “git push”)

$ git remote rm origin
$ git remote add project1 https://github.com/userid/projectpath
$ git remote -v
project1 https://github.com/userid/projectpath(fetch)
project1 https://github.com/userid/projectpath (pull)

$ <edit File B>
$ git commit –am “...”
$ git push project1
(or git push project1 main)
```





Working with a Remote and Multiple Users

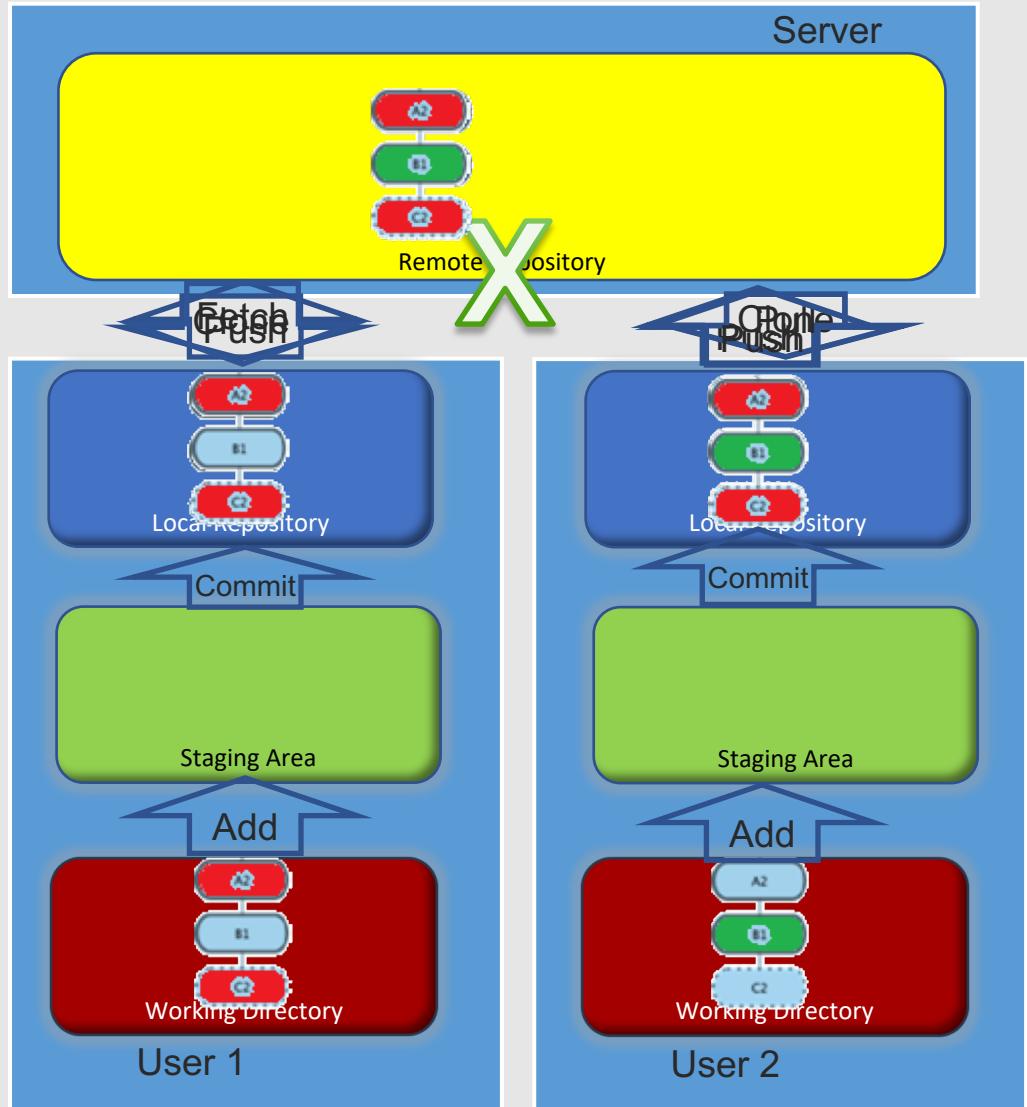
54

User 1

```
$ git clone ...
$ <edit File A and File C>
$ git commit –am “...”
$ git push
$ git fetch
```

User 2

```
$ git clone ...
$ <edit File B>
$ git commit –am “...”
$ git push
$ git pull
$ git push
```



Lab 6 - Using the Overall Workflow with a Remote Repository

Purpose: In this lab, you'll get some practice with remotes via a GitHub account. You'll fork a repository, clone it down to your system to work with, rebase changes, and deal with conflicts at push time.



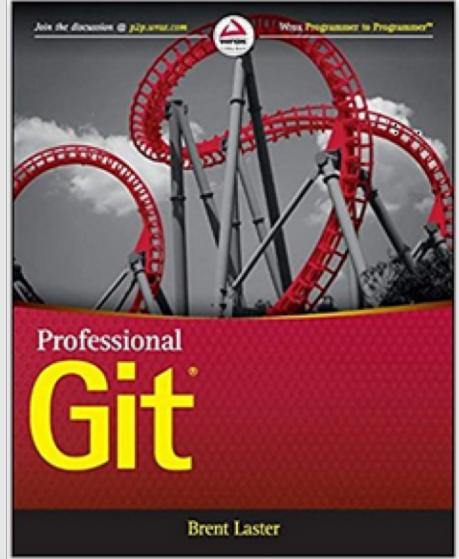
That's all - thanks!

Professional Git 1st Edition

by Brent Laster (Author)

★★★★★ 7 customer reviews

[Look inside](#)



New courses on Git, K8S, Operators, GitOps and more!

TECH SKILLS TRANSFORMATIONS, LLC

Home Contact Us

TECH LEARNING MADE EASY

Get the instruction you need to upskill now! Click the button below to see upcoming trainings.

Upcoming live training with O'Reilly Media!



ABOUT US

techskillstransformations.com
getskillsnow.com



Brent Laster