Version 1.7
12/30/23

Git in 4 Weeks

(Part 2)

Presented by Brent Laster &

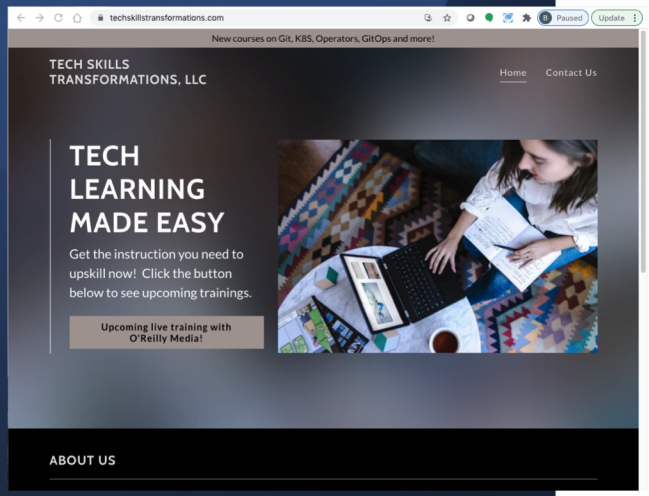
Tech Skills Transformations LLC

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

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 -
  - Professional Git
  - Jenkins 2 – Up and Running book
  - Learning GitHub Actions
  - Various reports on O'Reilly Learning
- https://www.linkedin.com/in/brentlaster
- @BrentCLaster
- GitHub: brentlaster

Professional Git Book

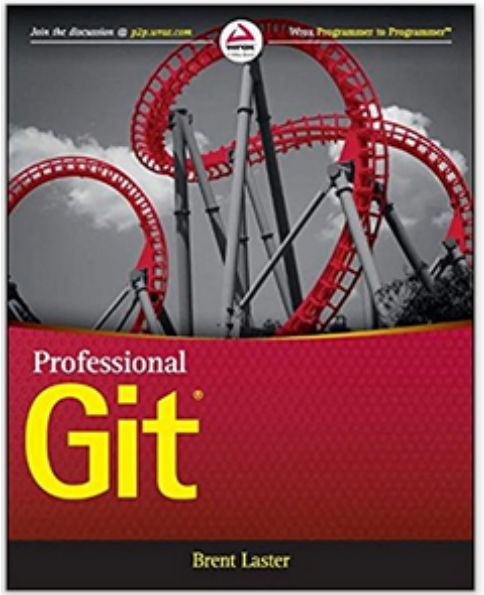
- Extensive Git reference, explanations,
- and examples
- First part for non-technical
- Beginner and advanced reference
- Hands-on labs

Professional Git 1st Edition

by Brent Laster (Author)

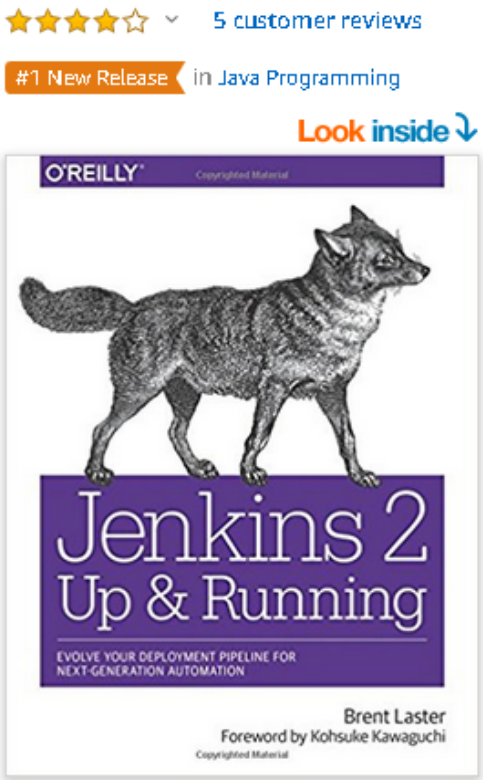
7 customer reviews

Look inside



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***



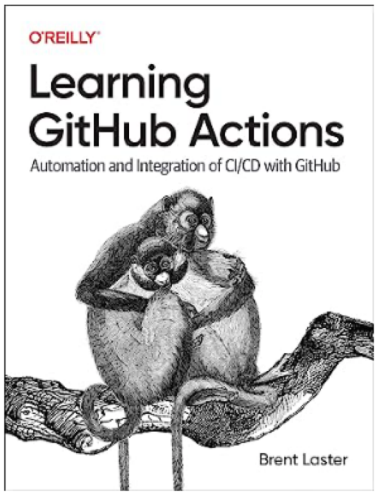
GitHub Actions book

All | Get the app | Prime Day | Back to School | Add People | Buy Again | Gift Cards | Recommendations | IT Supplies | Business Savings | Amazon Basics | EN | Hello, Accou

Guide buyers in your org | Group: Tech Skills Transf

Books | Advanced Search | New Releases | Best Sellers & More | Children's Books | Textbooks | Textbook Rentals | Best Books of the Month | Your Company Bookshelf

Books › Computers & Technology › Programming



Roll over image to zoom in

Follow the Author

Brent Laster | Follow

Learning GitHub Actions: Automation and Integration of CI/CD with GitHub 1st Edition

by Brent Laster (Author)

See all formats and editions

Paperback
$65.99 prime

1 New from $65.99

Pre-order Price Guarantee. Terms

Automate your software development processes with GitHub Actions, the continuous integration and continuous delivery platform that integrates seamlessly with GitHub. With this practical book, open source author, trainer, and DevOps director Brent Laster explains everything you need to know about using and getting value from GitHub Actions. You'll learn what actions and workflows are and how they can be used, created, and incorporated into your processes to simplify, standardize, and automate your work in GitHub.

This book explains the platform, components, use cases, implementation, and integration points of actions, so you can leverage them to provide the functionality and features needed in today's complex pipelines and software development processes. You'll learn how to design and implement automated workflows that respond to common events like pushes, pull requests, and review updates. You'll understand how to use the components of the GitHub Actions platform to gain maximum automation and benefit.

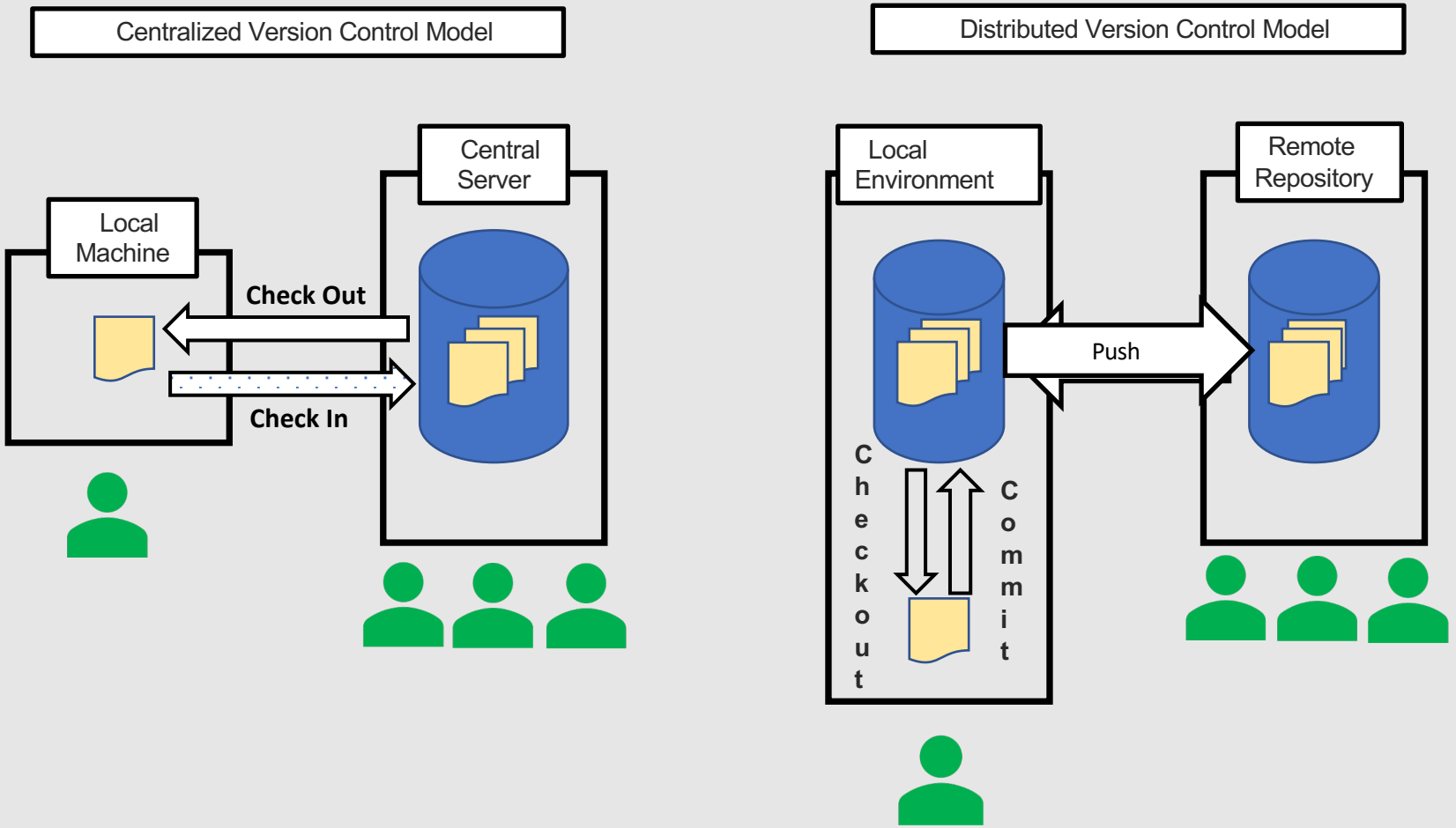
With this book, you will:

- Learn what GitHub Actions are, the various use cases for them, and how to incorporate them into your processes
- Understand GitHub Actions' structure, syntax, and semantics
- Automate processes and implement functionality
- Create your own custom actions with Docker, JavaScript, or shell approaches
- Troubleshoot and debug workflows that use actions
- Combine actions with GitHub APIs and other integration options
- Identify ways to securely implement workflows with GitHub Actions
- Understand how GitHub Actions compares to other options

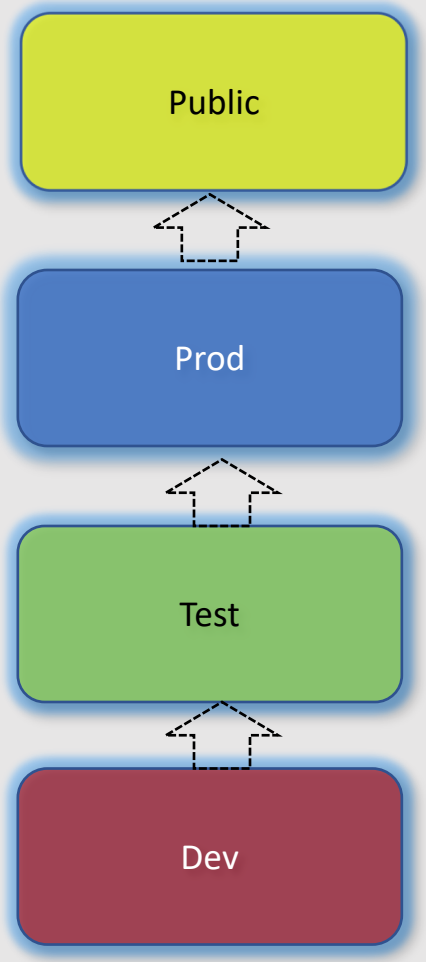
Read less

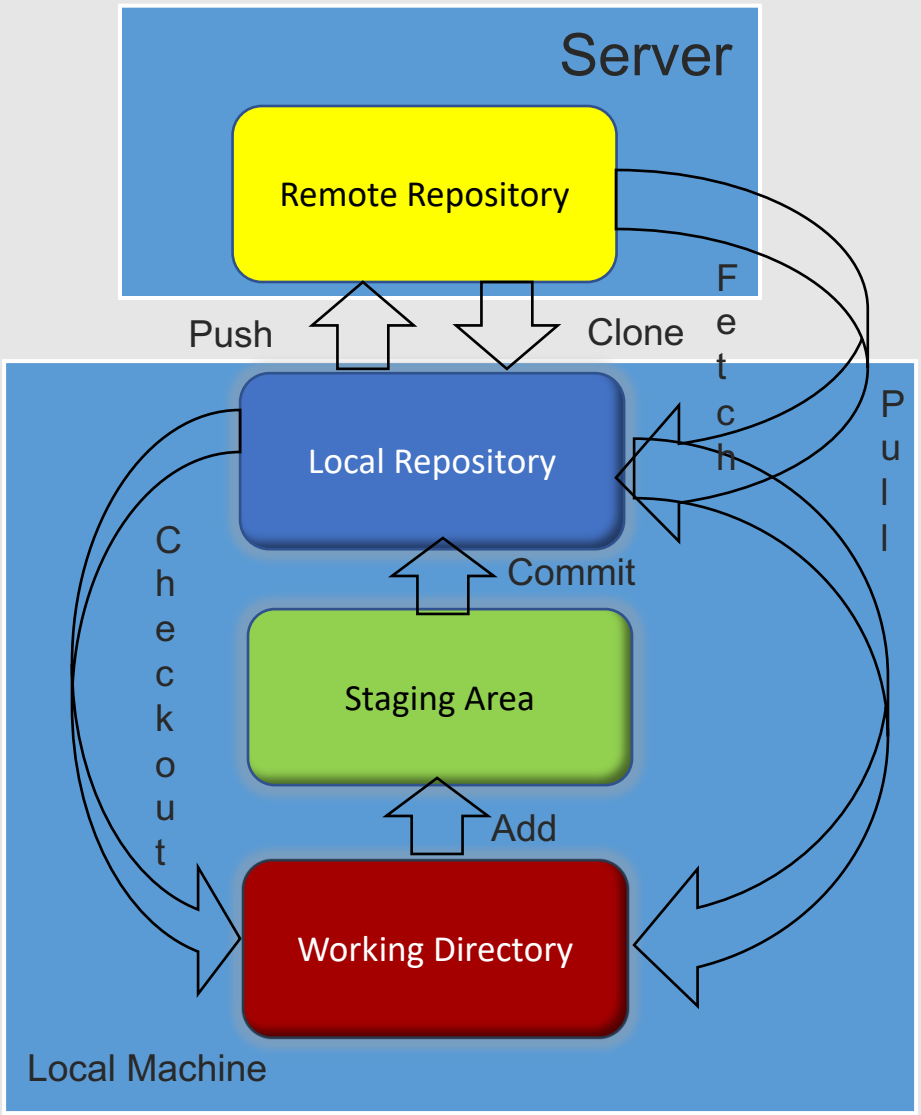
Review

Centralized vs. Distributed VCS



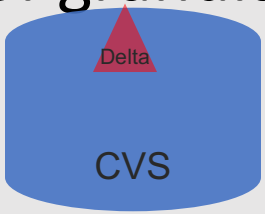
Git in One Picture



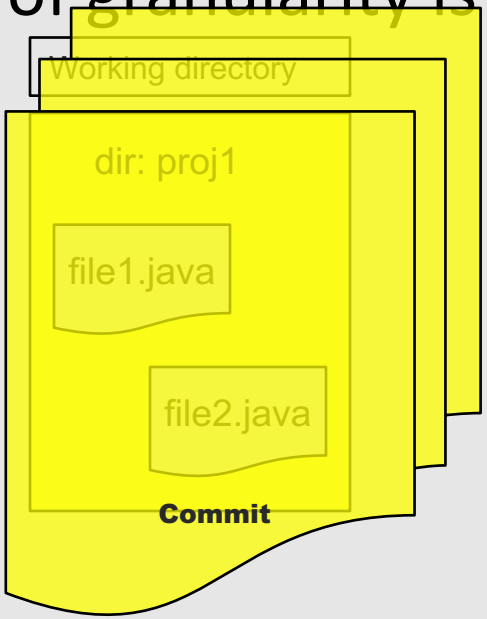


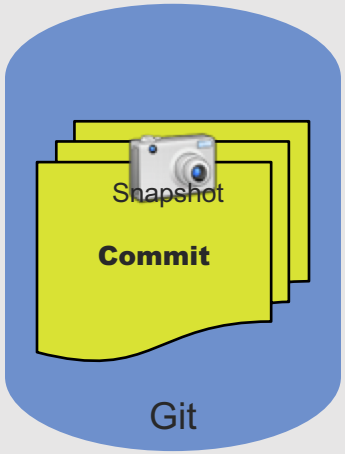
Git Granularity (What is a unit?)

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

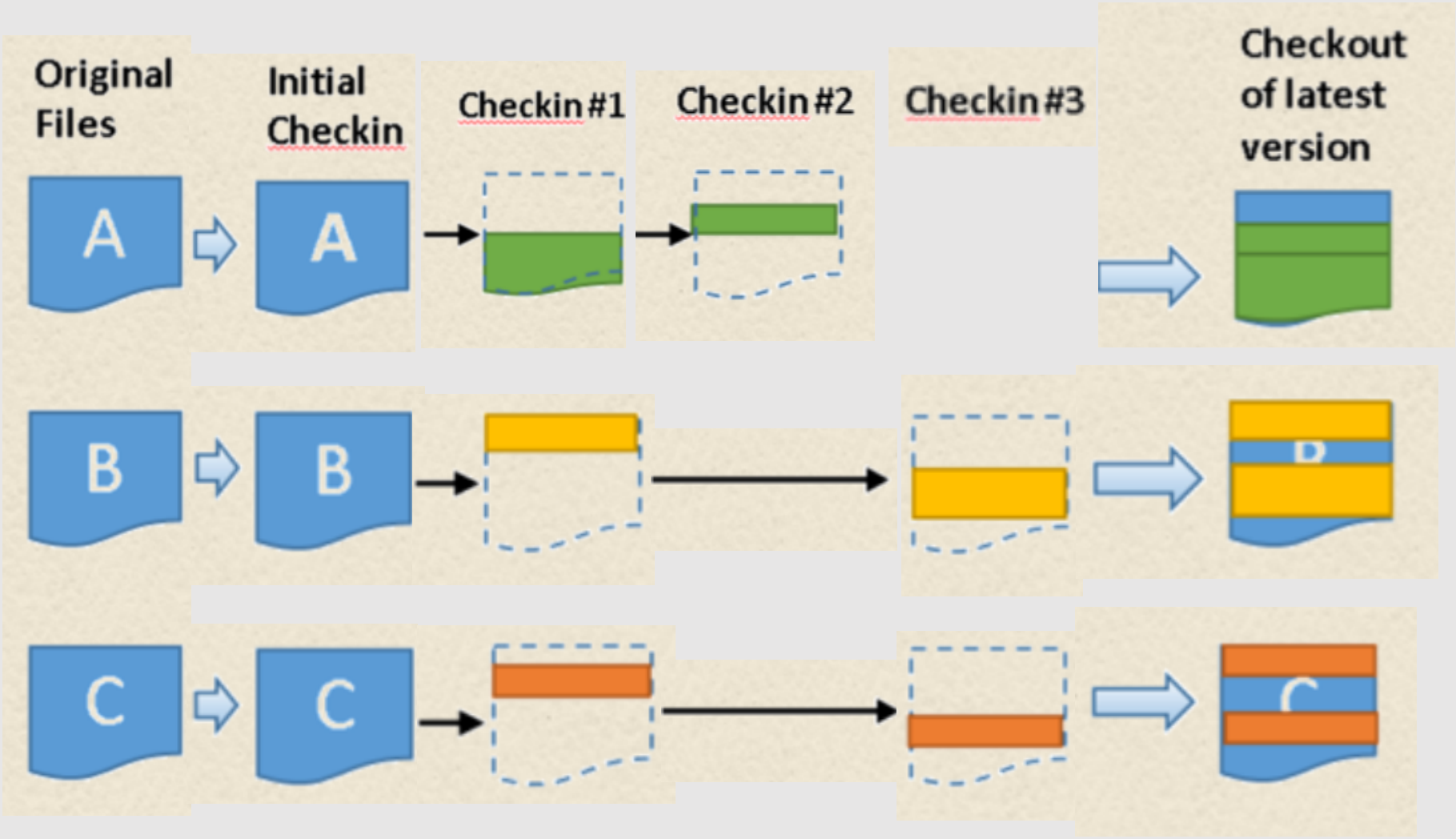


- In Git, the unit of granularity is usually a tree

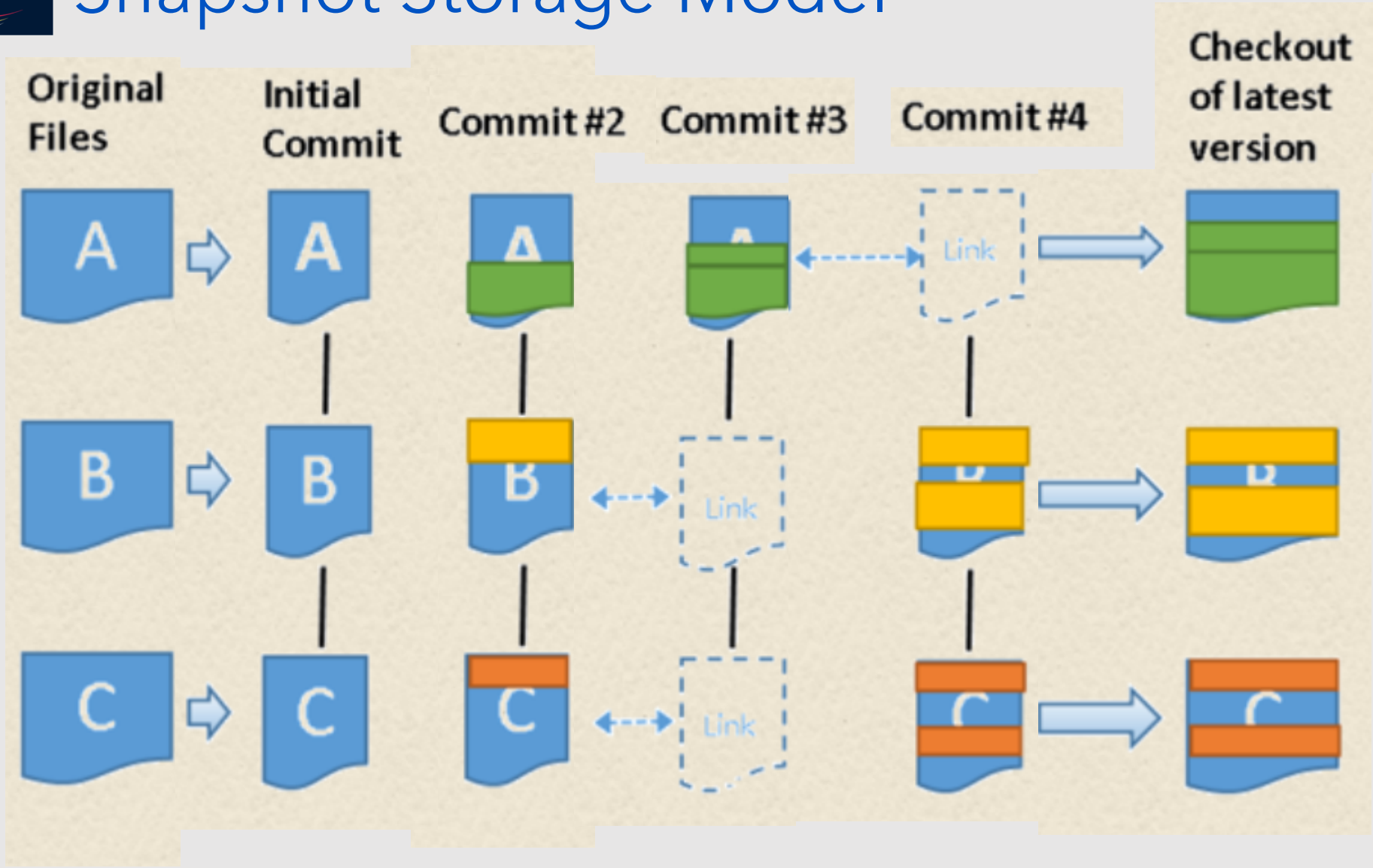




Delta Storage Model



Snapshot Storage Model



Initializing a Repo and Adding Files (commands)

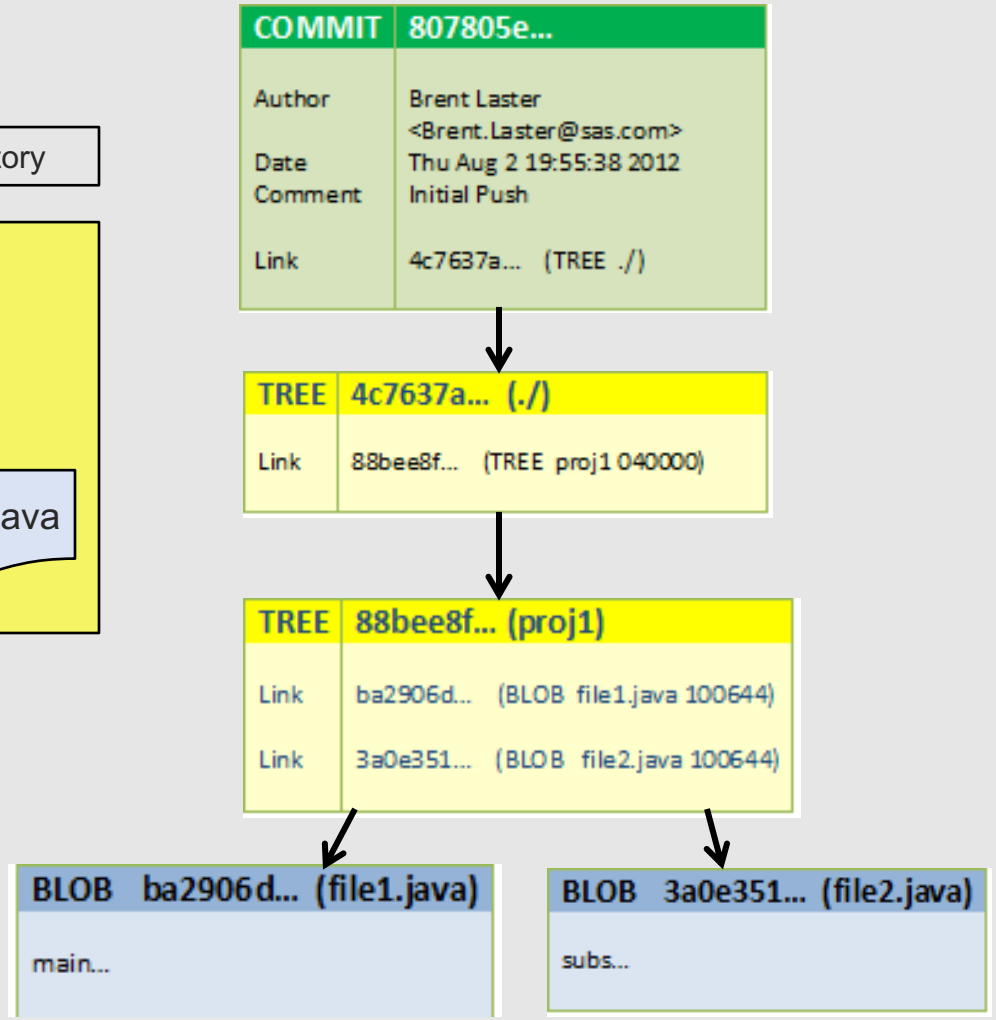
- git init
  - For creating a repository in a directory with existing files
  - Creates repository skeleton in .git directory
- git add
  - Tells git to start tracking files
  - Patterns: git add *.c or git add . (. = files and dirs recursively)
- git commit
  - Promotes files into the local repository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

Git and File/Directory Layouts

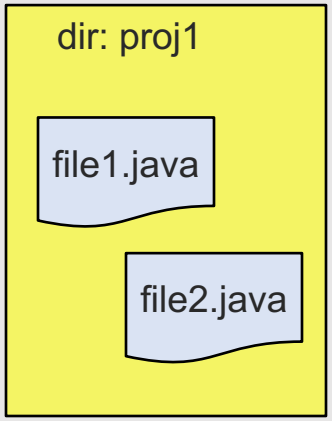
Local Repository File Layout

```
$ 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
                7637a4b66aefc1ee877eaa1afc70610f0ee7cc
        80
                7805ea7ae9fdf1b06e876af6e9a69a349b52a3
        88
                bee8f0c181784d605eabe35fb04a5a443ae6b7
        ba
                2906d0666cf726c7eaadd2cd3db615dedfdf3a
        info
                *
        pack
                *
refs
        heads
                main
        tags
                *
```

Git snapshot

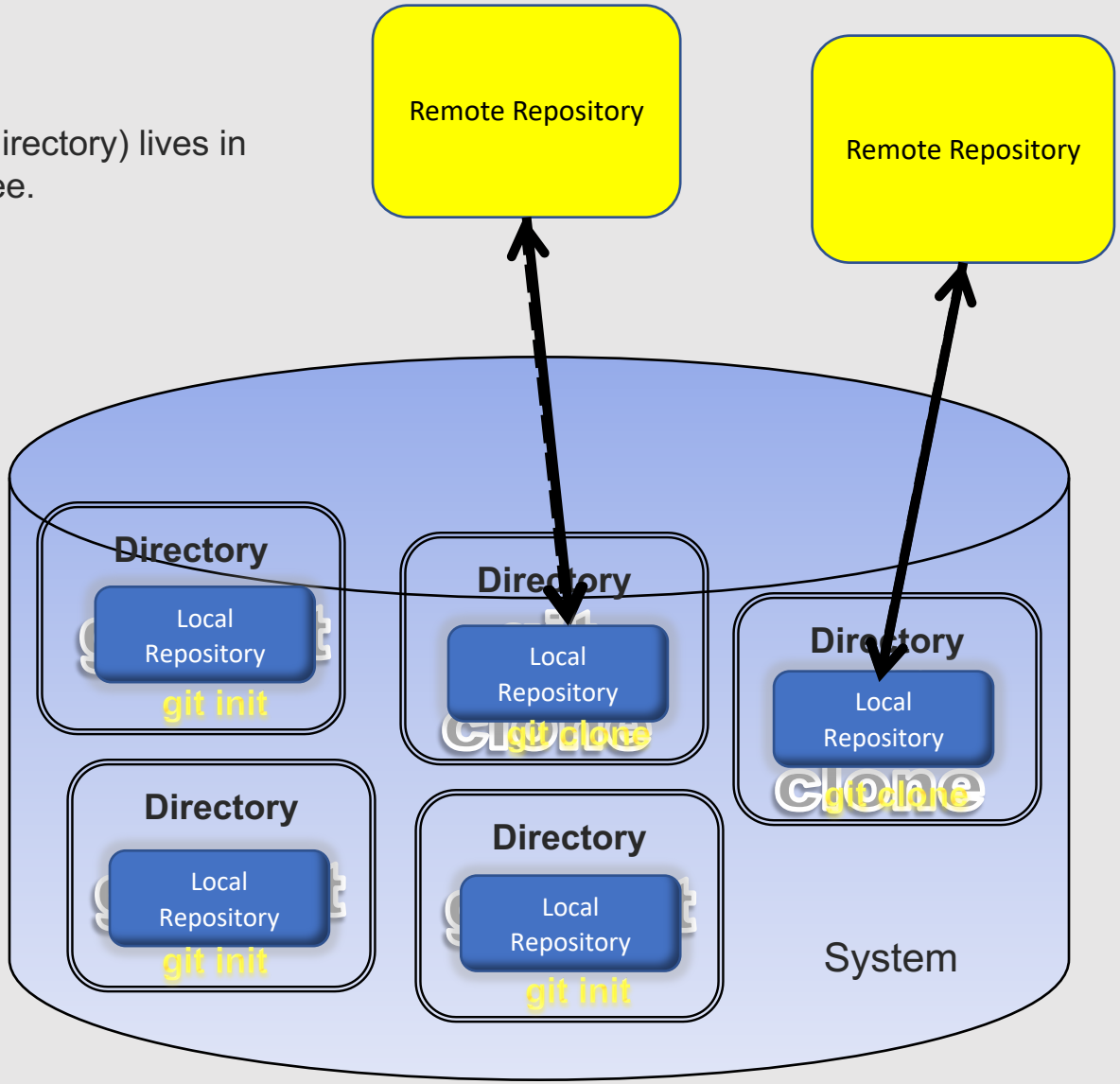


Working directory

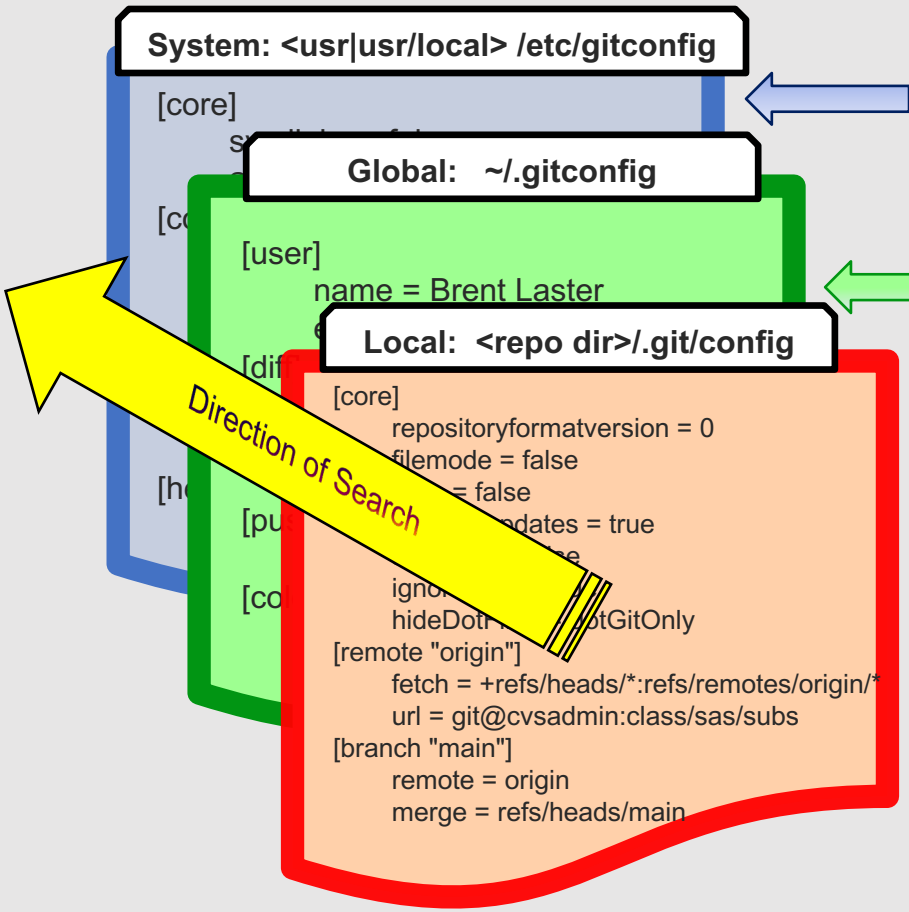


Starting Work with Git – Multiple Repositories

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



Git Configuration Files



System Configuration File (if exists)
Scope: All users on system

`git config --system`

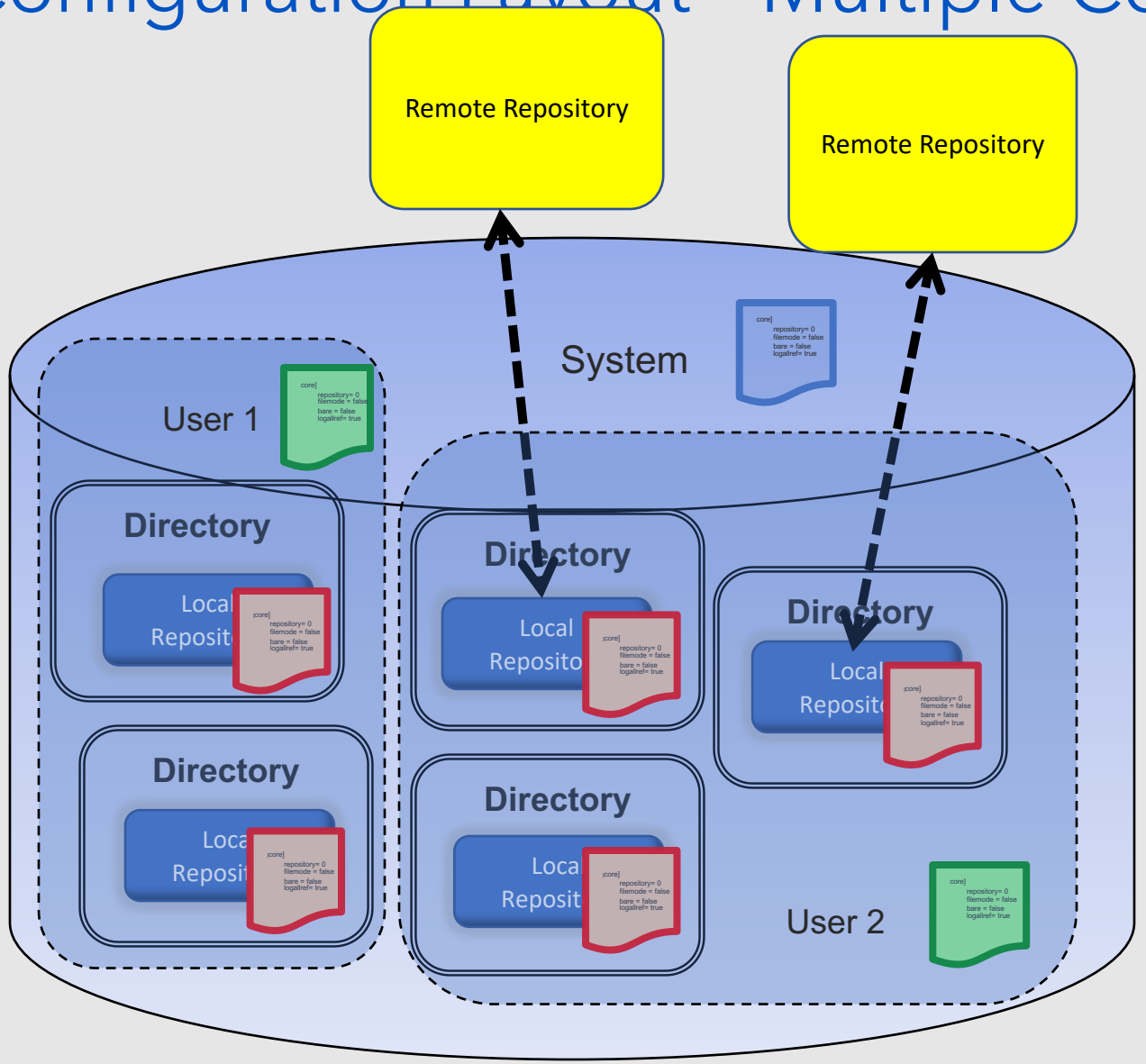
Global Configuration File
Scope: Current User

`git config --global`

Local Configuration File
Scope: Local repository

`git config or git config --local`

Git Configuration Layout – Multiple Configs



File Status Lifecycle

1. Files start out in wd (rev a)
Git aware? No – **untracked** files
Anything staged? No = Changes **not staged** for commit
Git compared to WD: N/A

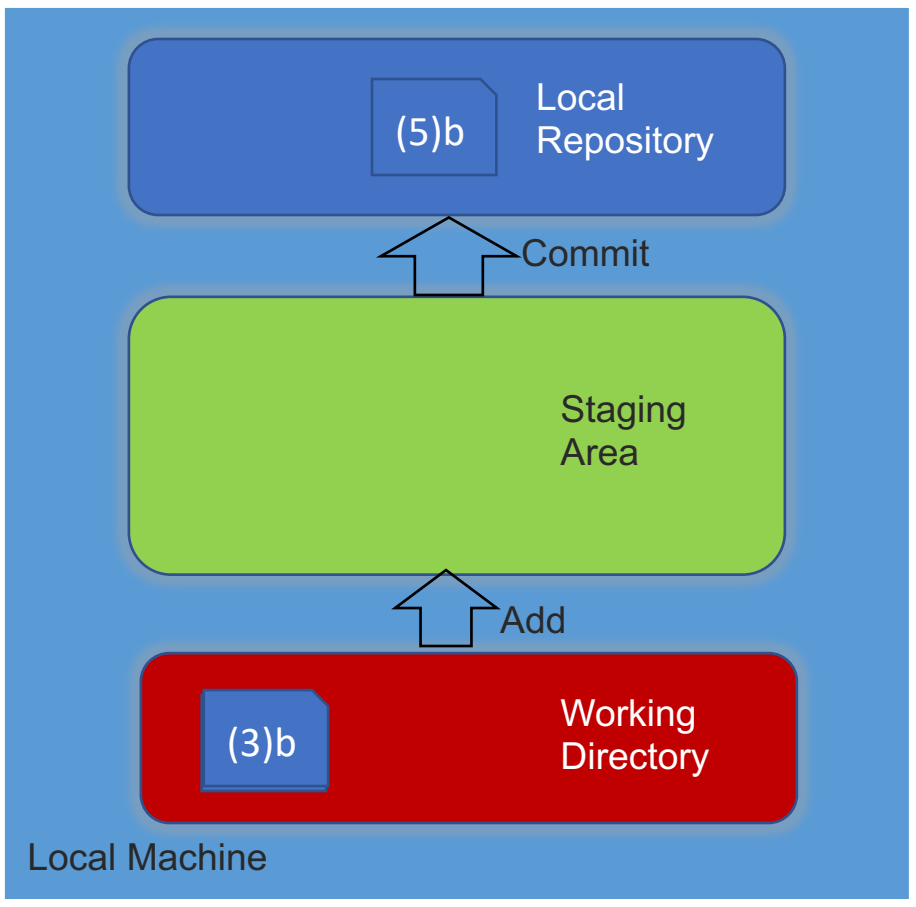
2. (Git) Add file.
Git aware: Yes – **tracked**
Anything staged? Rev a staged = Changes **to be committed**
Git compared to WD: Same - **unmodified**

3. Edit in working directory (rev b)
Git aware: Yes – **tracked**
Anything staged? Rev a **staged (to be committed)**
Git compared to WD : Different – **modified**

4. (Git) Add file. **Tracked**, rev b **staged** (rev a overwritten), **unmodified**

5. (Git) Commit. **Tracked**, Staged: Nothing, **unmodified**

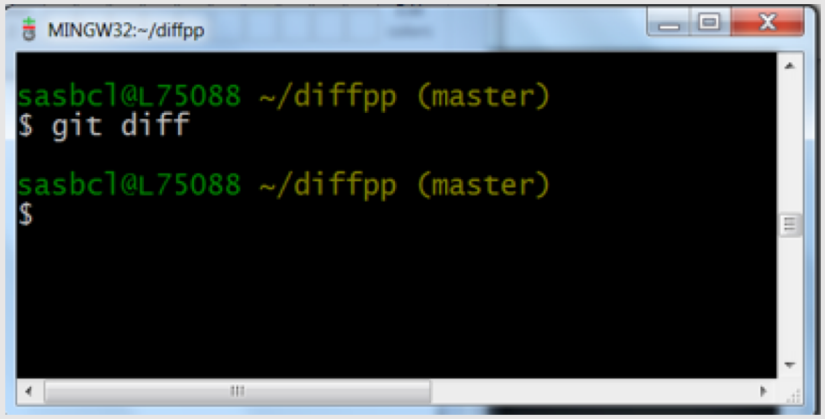
"nothing to commit (working directory clean)"

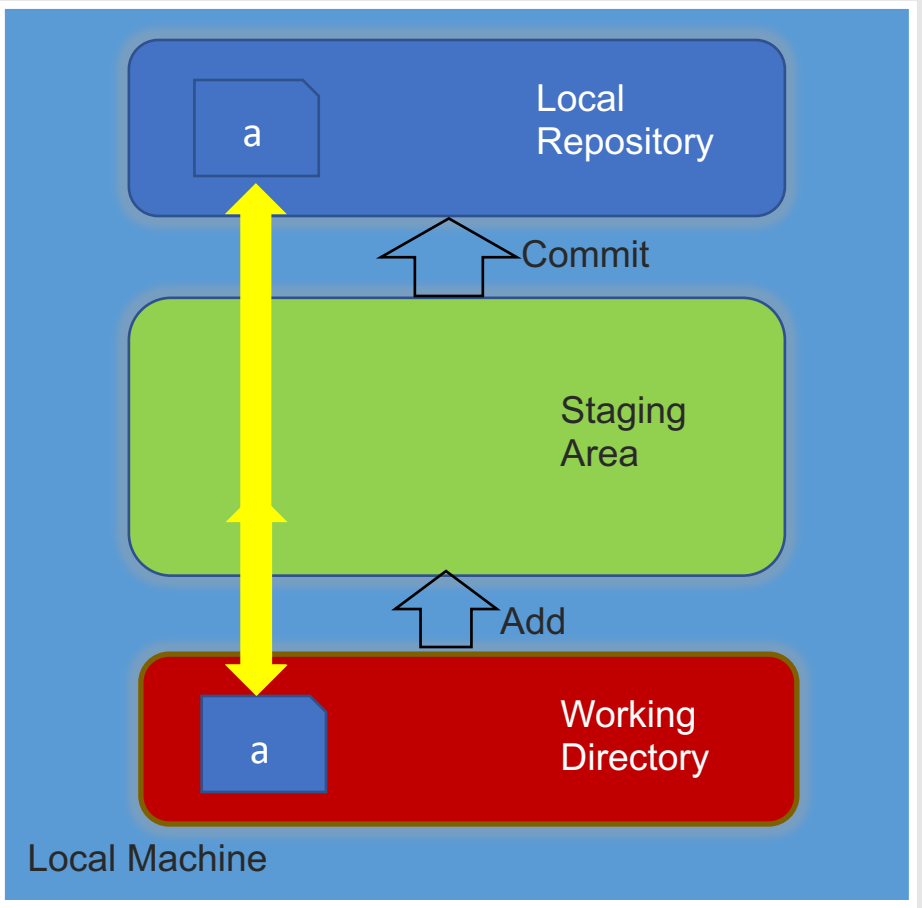


* "Changes to be committed" = staged
* "Changes not staged for commit" = wd

Understanding File Diffs 1

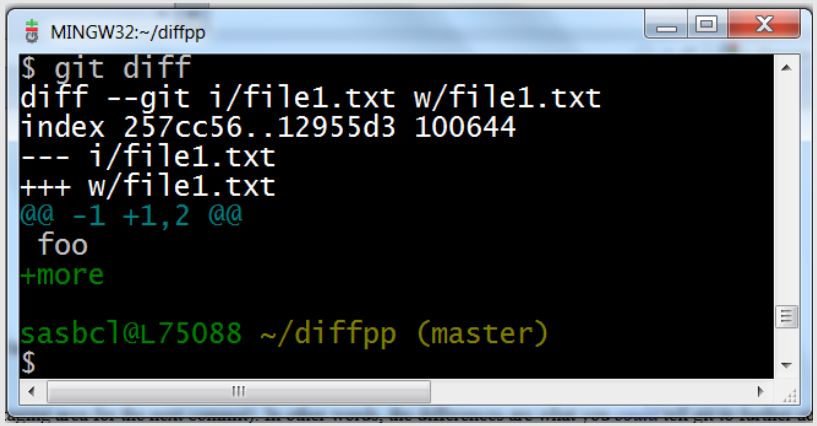
1.Assume file (rev a) created, added,and commited in local repo. WD version unchanged.
Git diff compares:
WD to SA (nothing there),
So compares to Local Repo
Results: Same
Output: Nothing

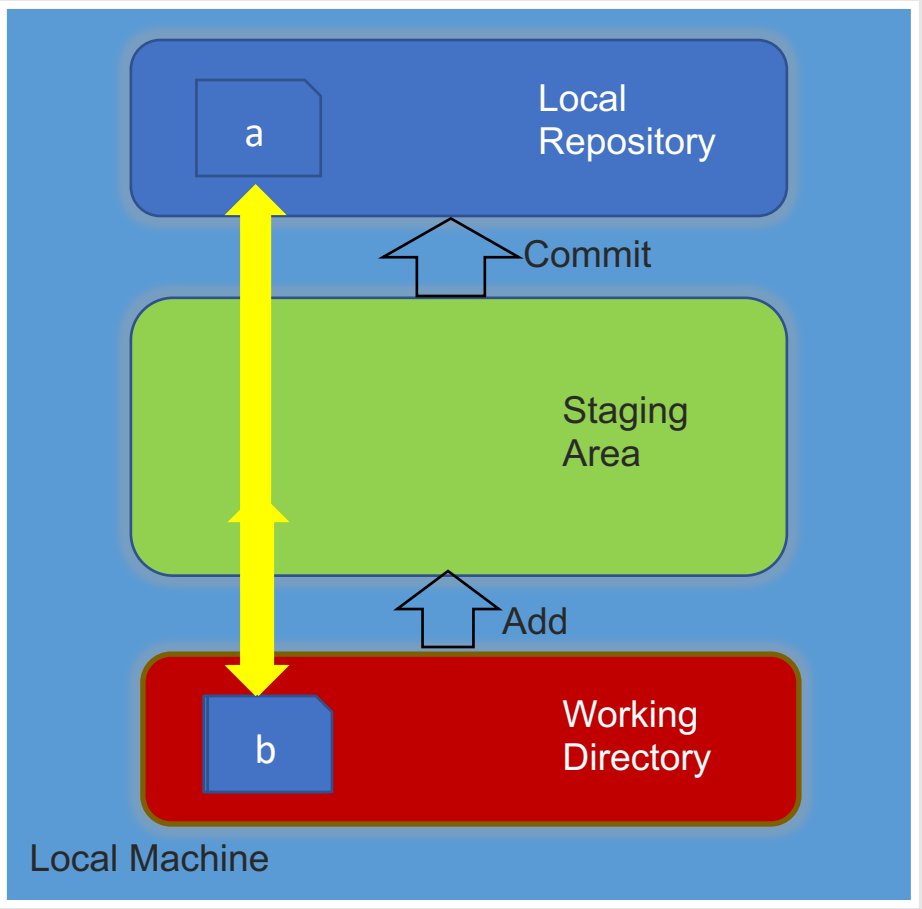




Understanding File Diffs 2

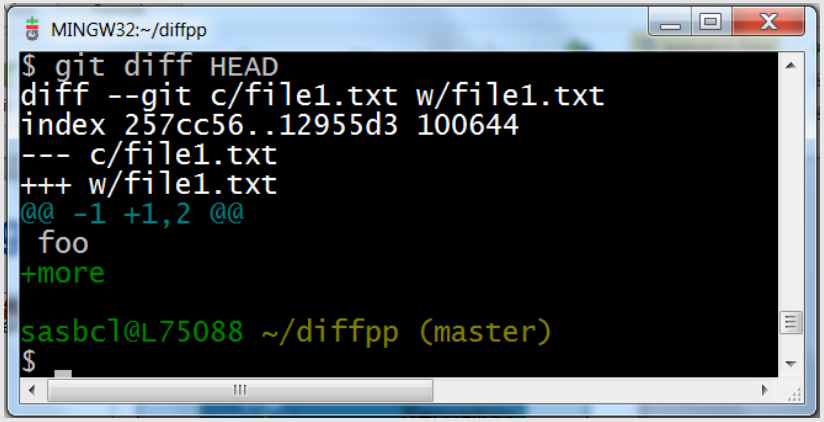
2. Now, we modify the file in the WD – rev b.
Git diff compares:
WD to SA (nothing there),
So compares to Local Repo
Results: Different
Output: Differences between WD and Local Repo

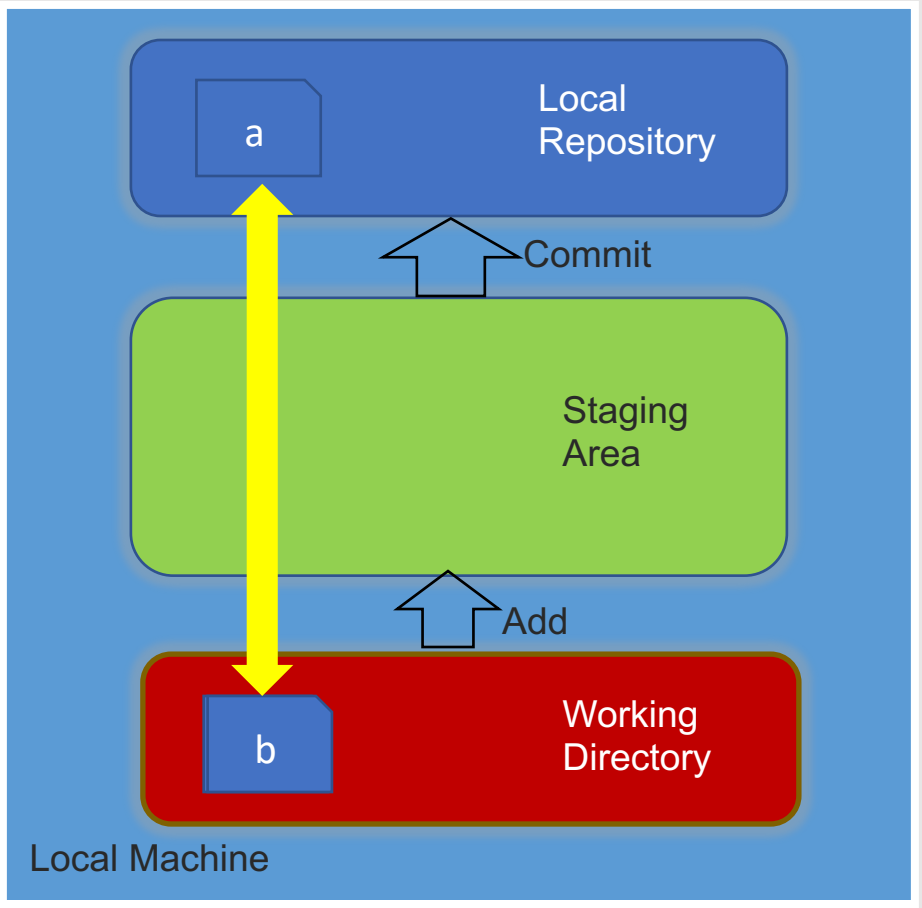




Understanding File Diffs 3

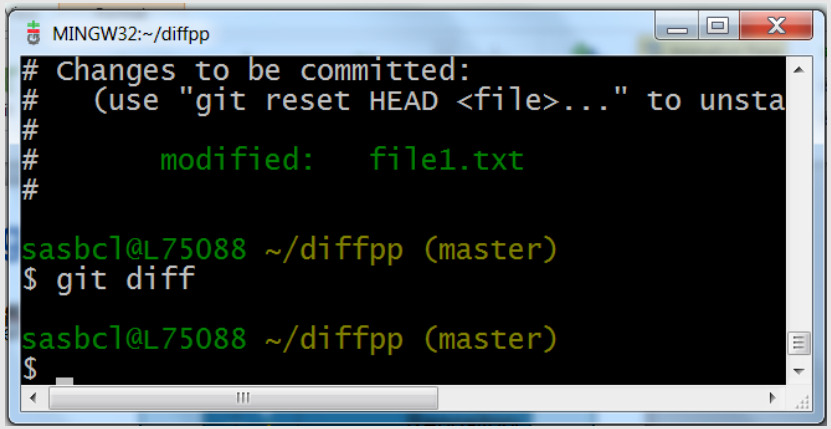
3. We can also compare explicitly against the Local Repo.
Git diff HEAD compares:
WD to Local Repo
Results: Different
Output: Differences between WD and Local Repo

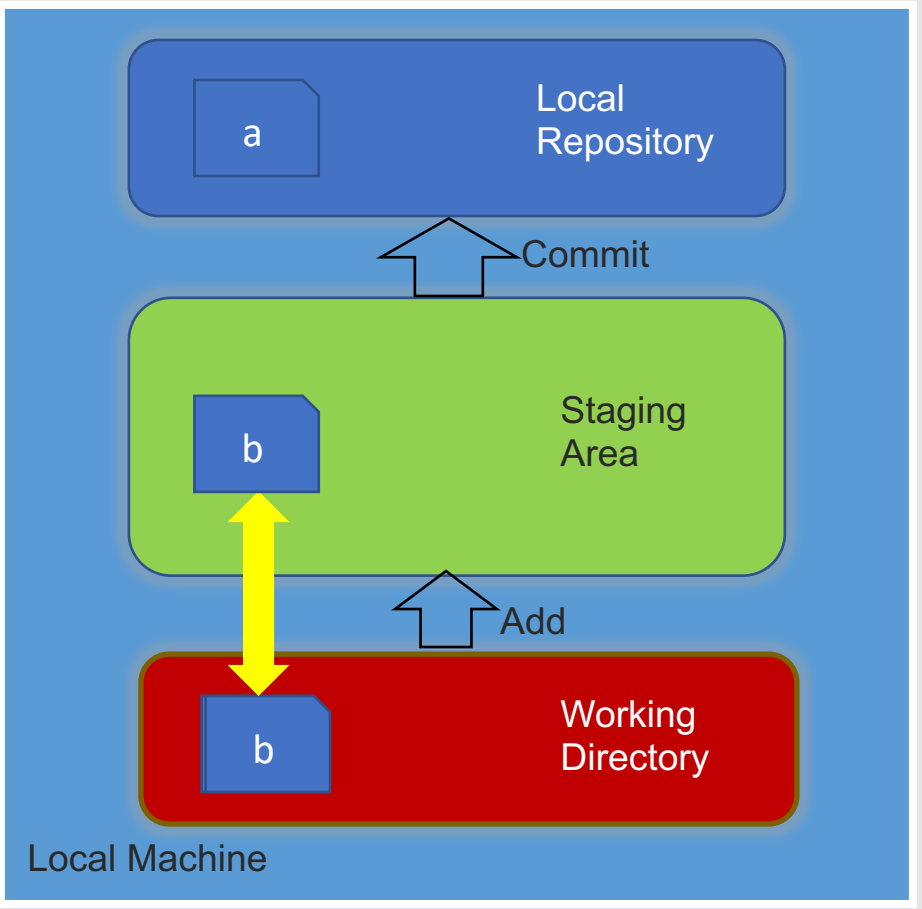




Understanding File Diffs 4

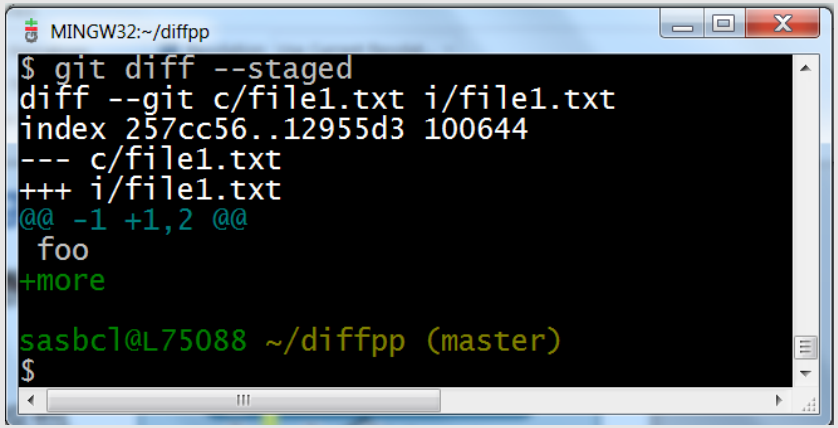
4. Let's add our change to the Staging Area (index).
Git diff compares:
WD to Staging Area
Results: Same
Output: Nothing

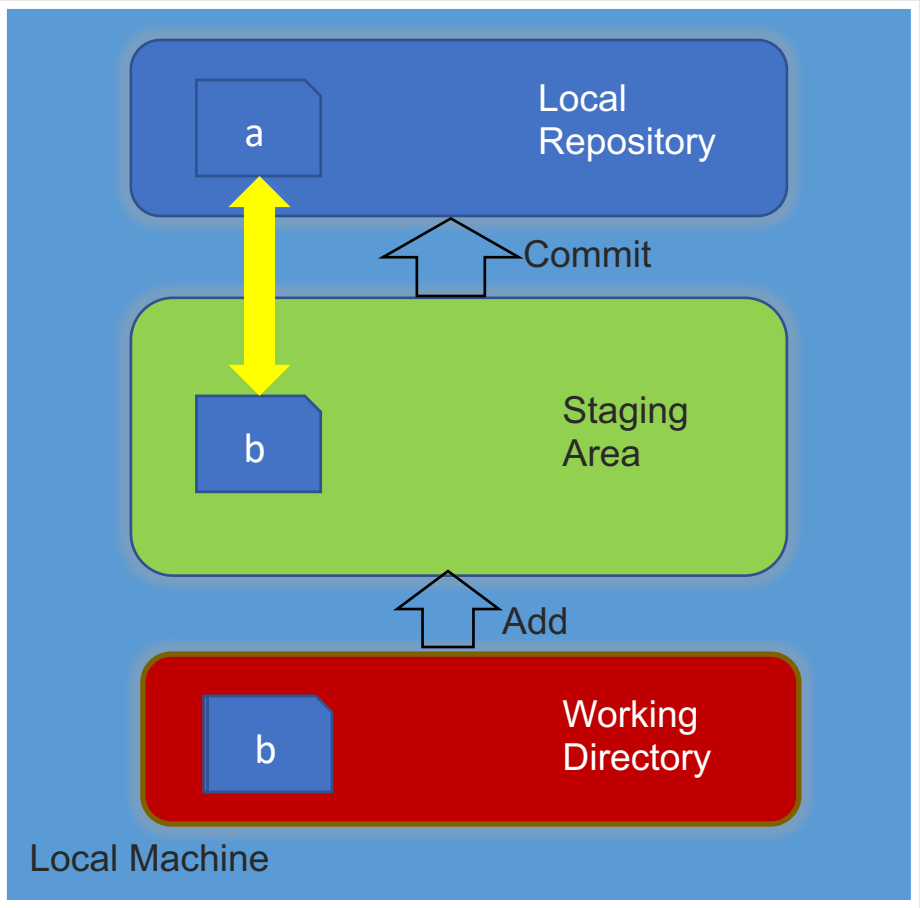




Understanding File Diffs 5

5. If we use the –staged or –cached option:
Git diff --staged compares:
Staging Area to Local Repo
Results: Different
Output: Differences between Staging Area and Local Repo





Git log examples

- 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

- 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

- 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

Supporting Files - .gitignore

- Tells git to not track files or directories (normally listed in a file named .gitignore)
- Operations such as git add . will skip files listed in .gitignore
- Rules
  - Blank lines or lines starting with # are ignored.
  - Standard glob patterns work.
  - You can end patterns with a forward slash (/) to specify a directory.
  - You can negate a pattern by starting it with an exclamation point (!).
- What types of things might we want git to ignore?

Supporting Files - .gitattributes

- A git attributes file is a simple text file that gives attributes to pathname – meaning git applies some special setting to a path or file type.
- Attributes are set/stored either in a .gitattributes file in one of your directories (normally the root of your project) or in the .git/info/attributes file if you don't want the attributes file committed with your project.
- Example use: dealing with binary files
- To tell git to treat all obj files as binary data, add the following line to your .gitattributes file:
- *.obj binary
  - With this setup, git won't try to convert or fix eol issues. It also won't try to compute or print a diff for changes in this file when you run git show or git diff on your project.

Git Attributes File – Example

- Basic example

```
# Set default behaviour, in case users don't have core.autocrlf set.
* text=auto

# Explicitly declare text files we want to always be normalized and converted
# to native line endings on checkout.
*.c text
*.h text

# Declare files that will always have CRLF line endings on checkout.
*.sln text eol=crlf

# Denote all files that are truly binary and should not be modified.
*.png binary
*.jpg binary
```

- Advantage is that this can be put with your project in git. Then, the end of line configuration now travels with your repository. You don't need to worry about whether or not all users have the proper line ending configuration.
- Some sample gitattributes files in GitHub for certain set of languages.

Branching

What is a branch in source control?

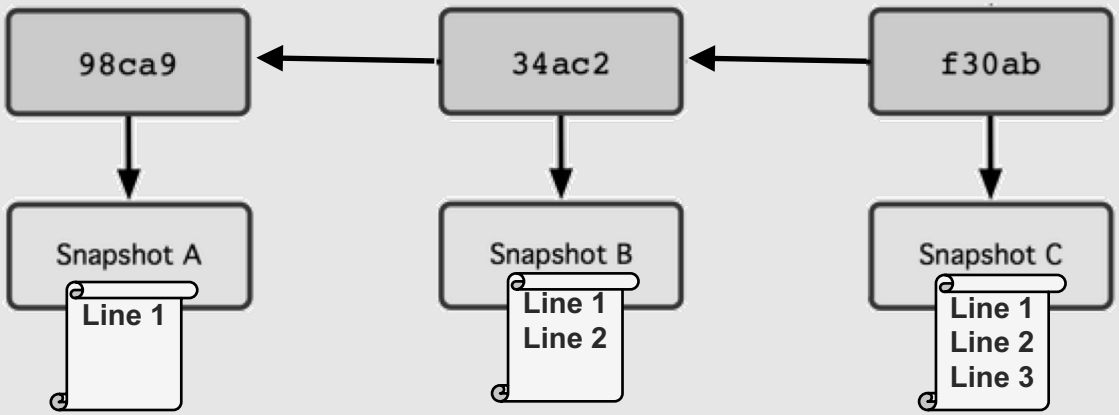
- Line of development
- Collection of specific versions of a group of files tagged in a common way
  - cvs rtag -a -D <date/time> -r DERIVED_FROM -b NEW_BRANCH PATHS_TO_BRANCH
- End result is I have an easy “handle” to get all of the files in the repository associated with that identifier – the branch 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)?

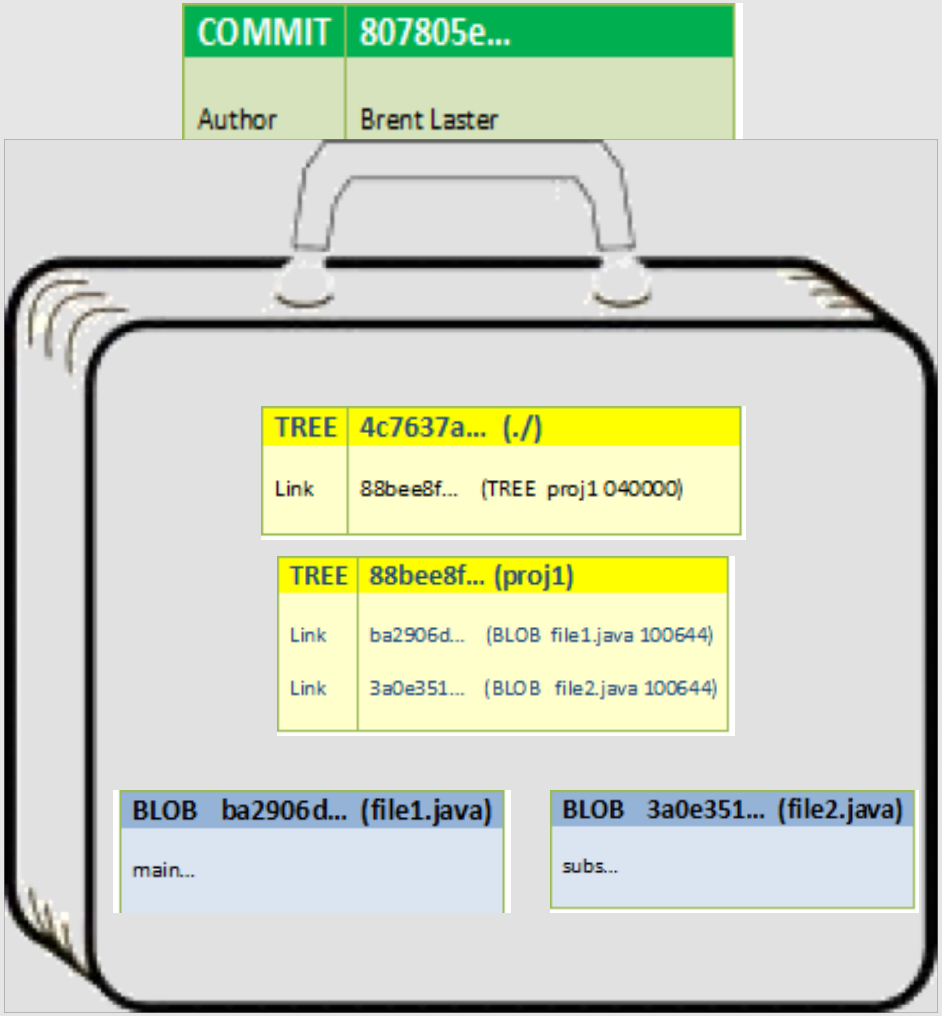
- Line of development associated with a specific change
- Collection of specific versions of a group of files associated with a specific commit
- End result is I have a “handle” to get all of the versions of files in the repository associated with that commit – handle = SHA1 for that commit
- What you 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

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

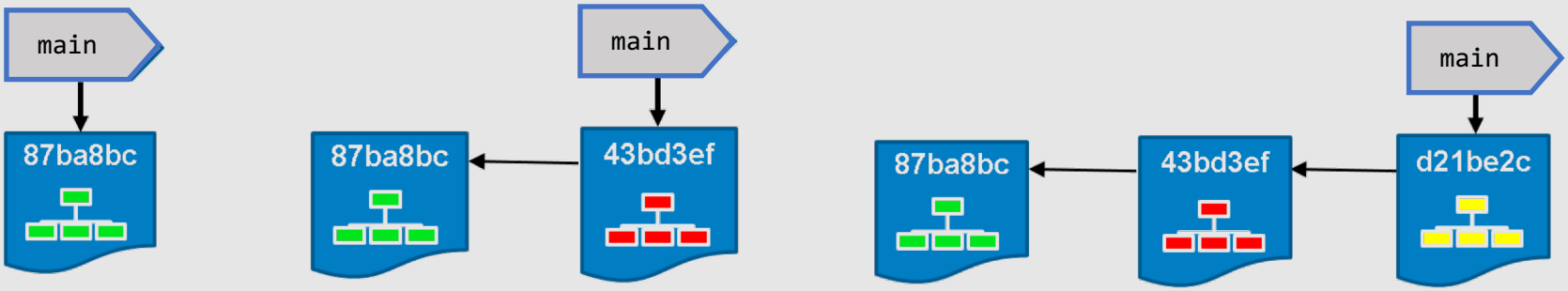


Commit SHA1's are a handle to a snapshot



Lightweight Branching

- 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.

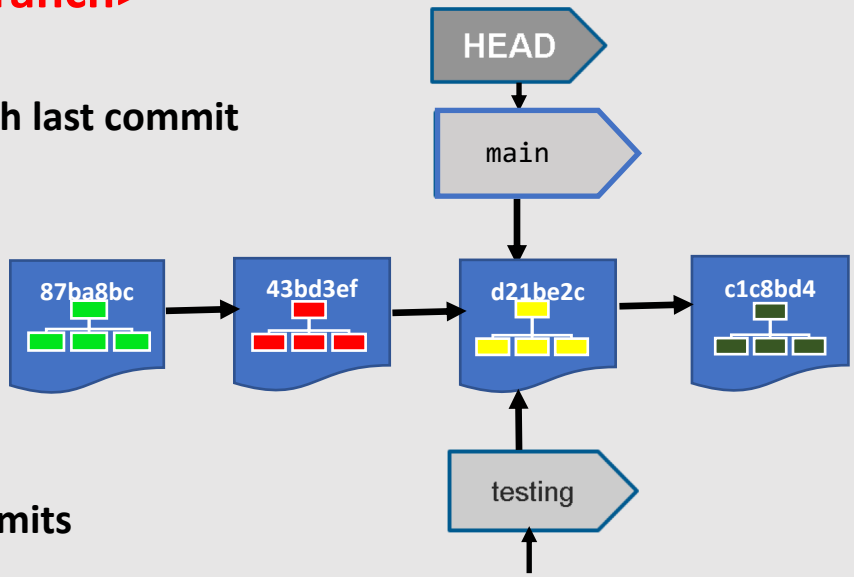
Creating and using a new branch

- **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

Command: git checkout <branch>

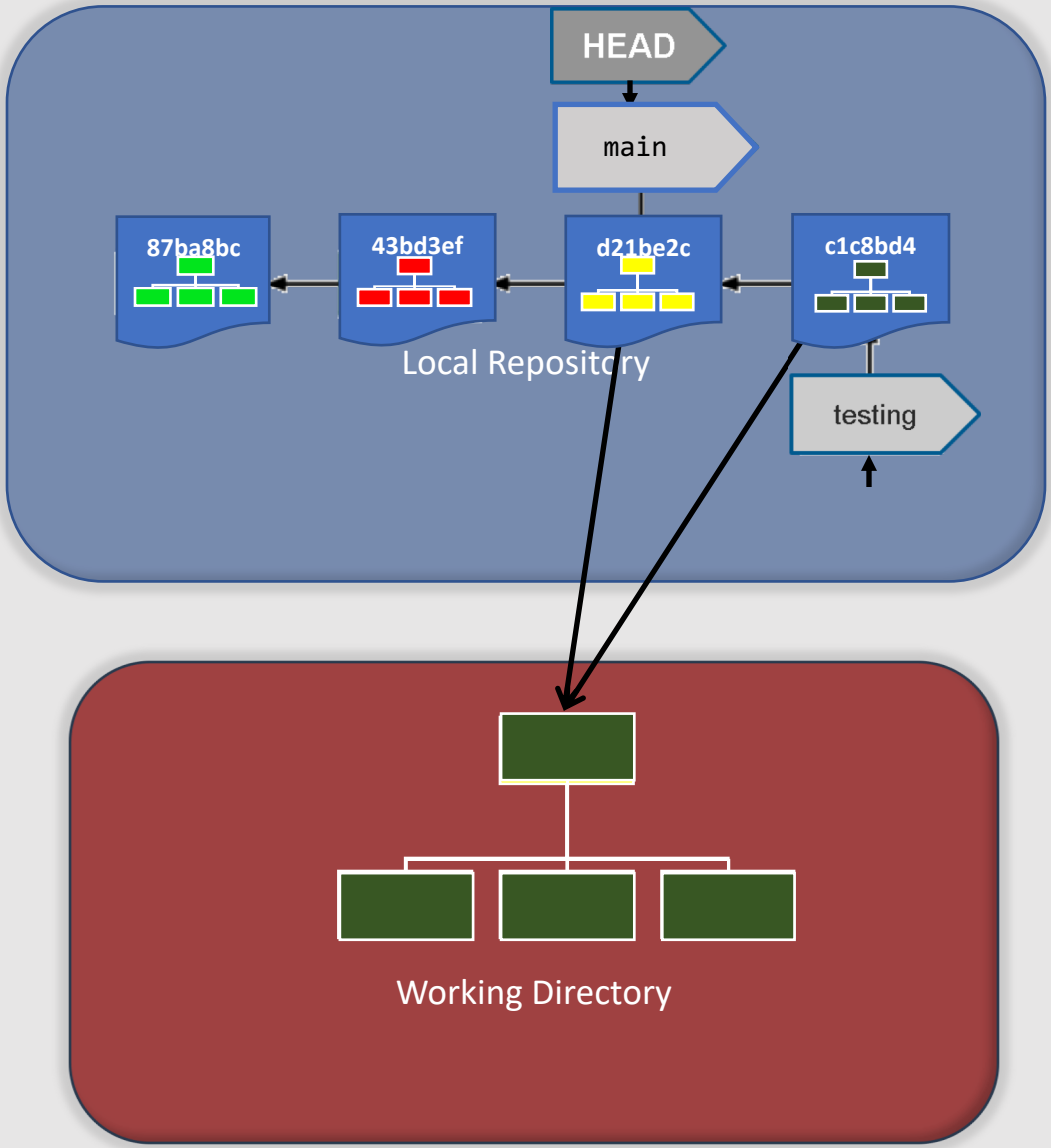
git checkout main

- **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?

- 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"

- 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.

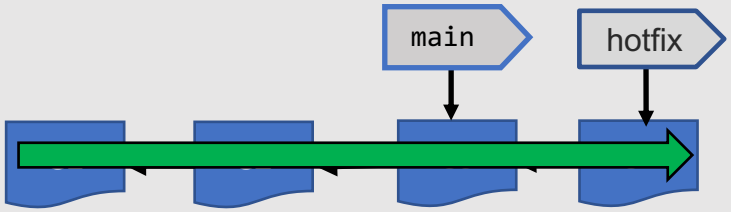
Merging Branches

- 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?

- 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 (-)
```

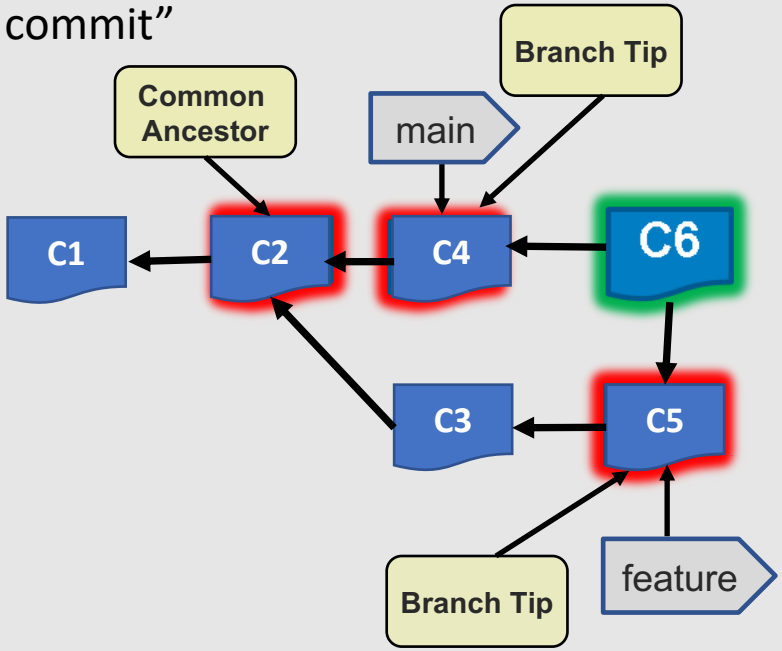


About “Fast Forward” – because commit pointed to by branch merged was directly “upstream” of the current commit, Git moves the pointer forward

(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?

- 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
```

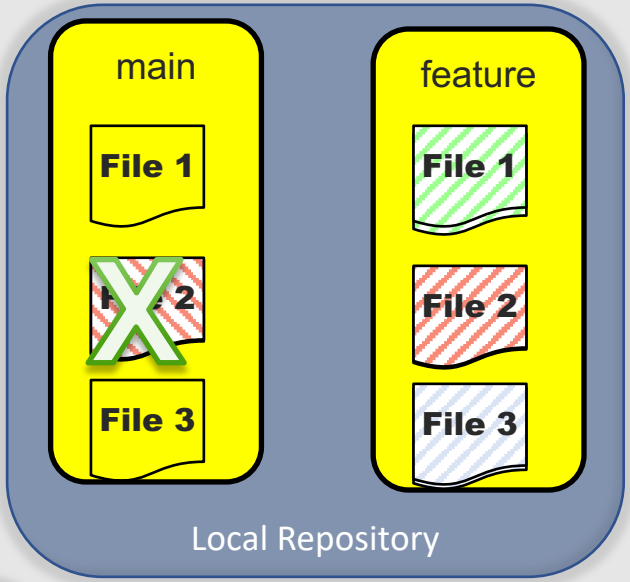
Merge Conflicts

- 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

47

```
$ 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”
```



Staging Area

Working Directory

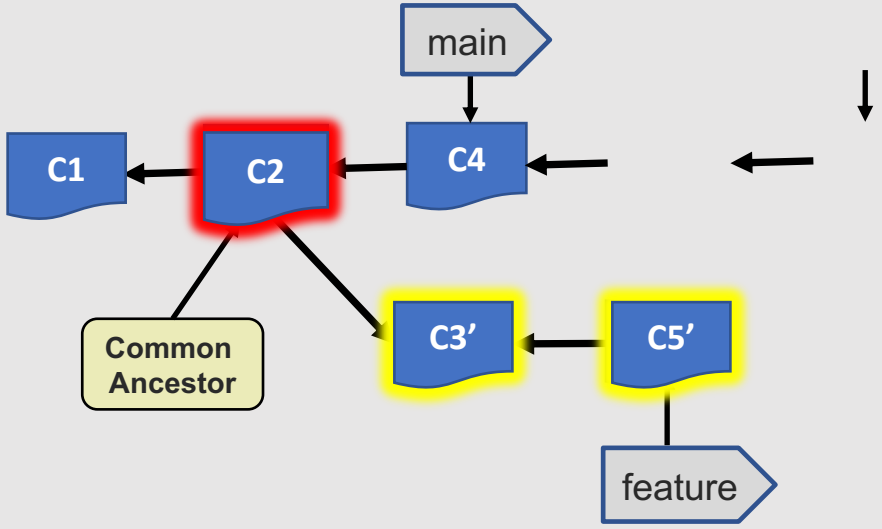
Lab 5 - Practice with Merging

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

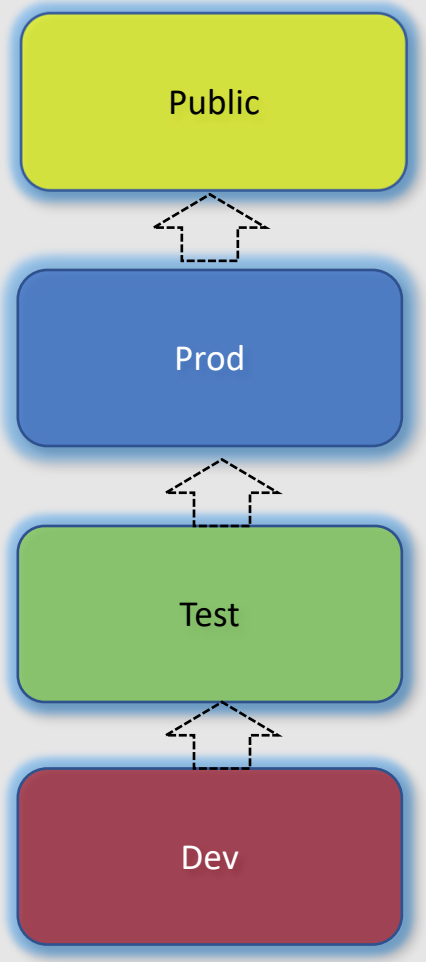
Merging: What is a Rebase?

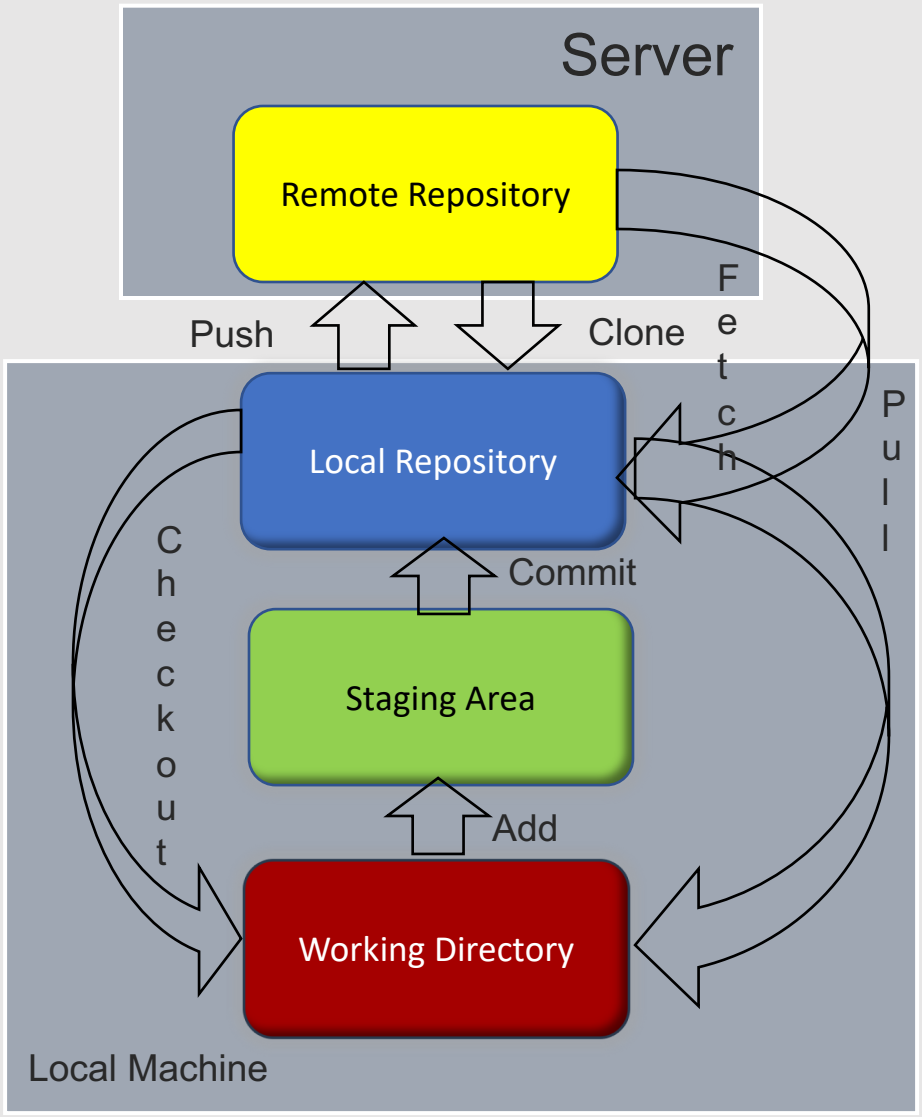
- Rebase – take all of the changes that were committed on one branch and replay (merge) them (one at a time in sequence) on another one.
- Process:
  - Goes to the common ancestor of the two branches (the one you are on and the one you are rebasing onto)
  - Gets the diff introduced by each commit of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





Git Remote Repositories

- 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)

Git Remote References "remotes" 52

- 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>

Working with Handles to Remotes

- 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)

Cloning a Remote Repo to get a Local Repo

- 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 commands for working with remotes

- **git clone**
  - Used to get a fresh copy of a repo locally
  - Grabs a copy of remote repo at that point in time, puts it in .git locally, checks out default branch

Post-clone commands - sync between local and remote - connection only active when doing a command

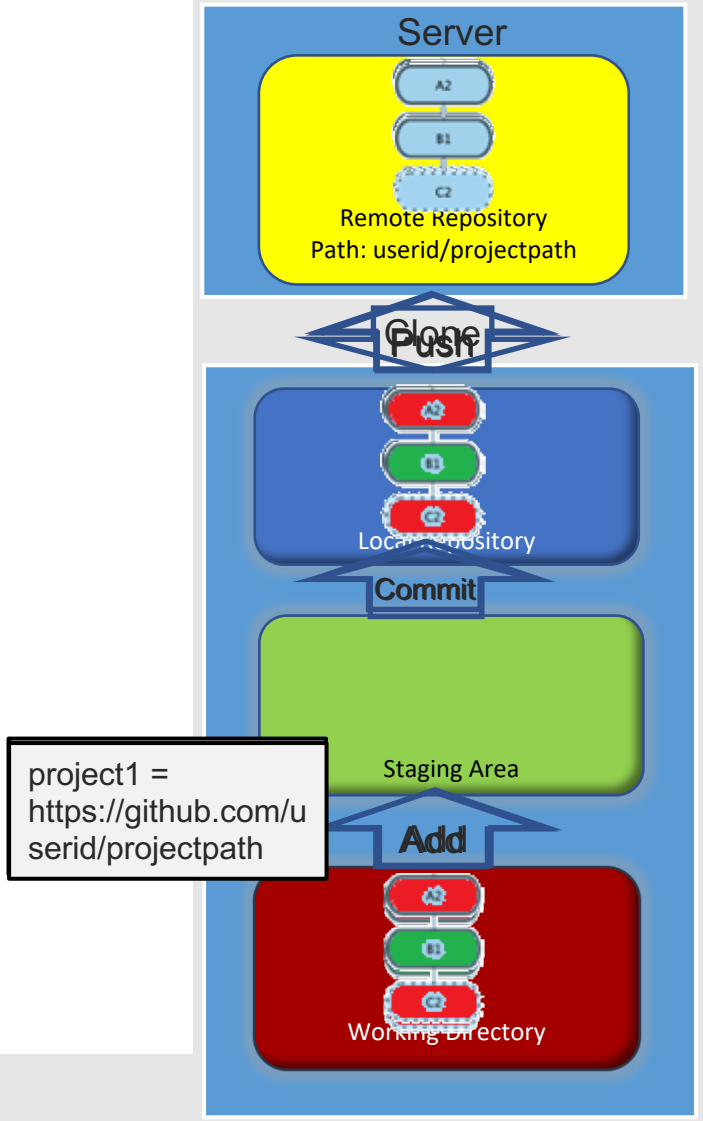
- **git fetch**
  - git fetch origin
  - Synchronize/mirror local repository from remote repo
  - Don't change anything in working directory

- **git pull**
  - Equivalent of git fetch + attempt to merge changes into working directory
  - Effectively a “fetch and merge”
  - git pull origin main (aka git pull)
  - Offers ability to define how merge can occur locally (fast forward, rebase, etc.)

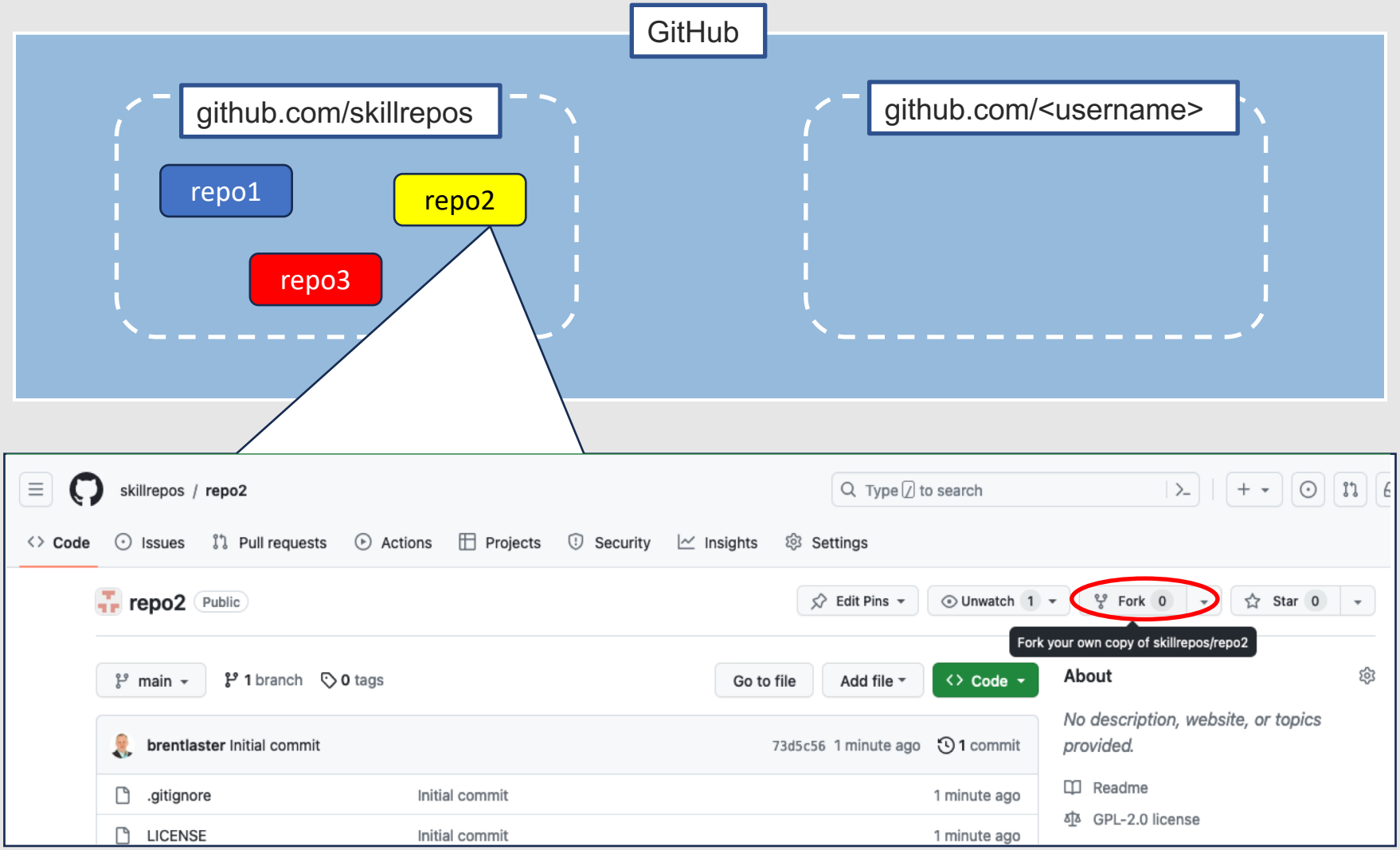
- **git push**
  - Synchronize/mirror content from local repo to remote repo
  - Git wants to do a fast forward merge on remote to cleanly take changes w/o conflicts
  - Only works if you have correct access and remote content hasn’t been changed
  - If can't do fast forward, then rejected and must fetch and merge or pull, resolve conflicts locally and push back out

Git Remotes Example

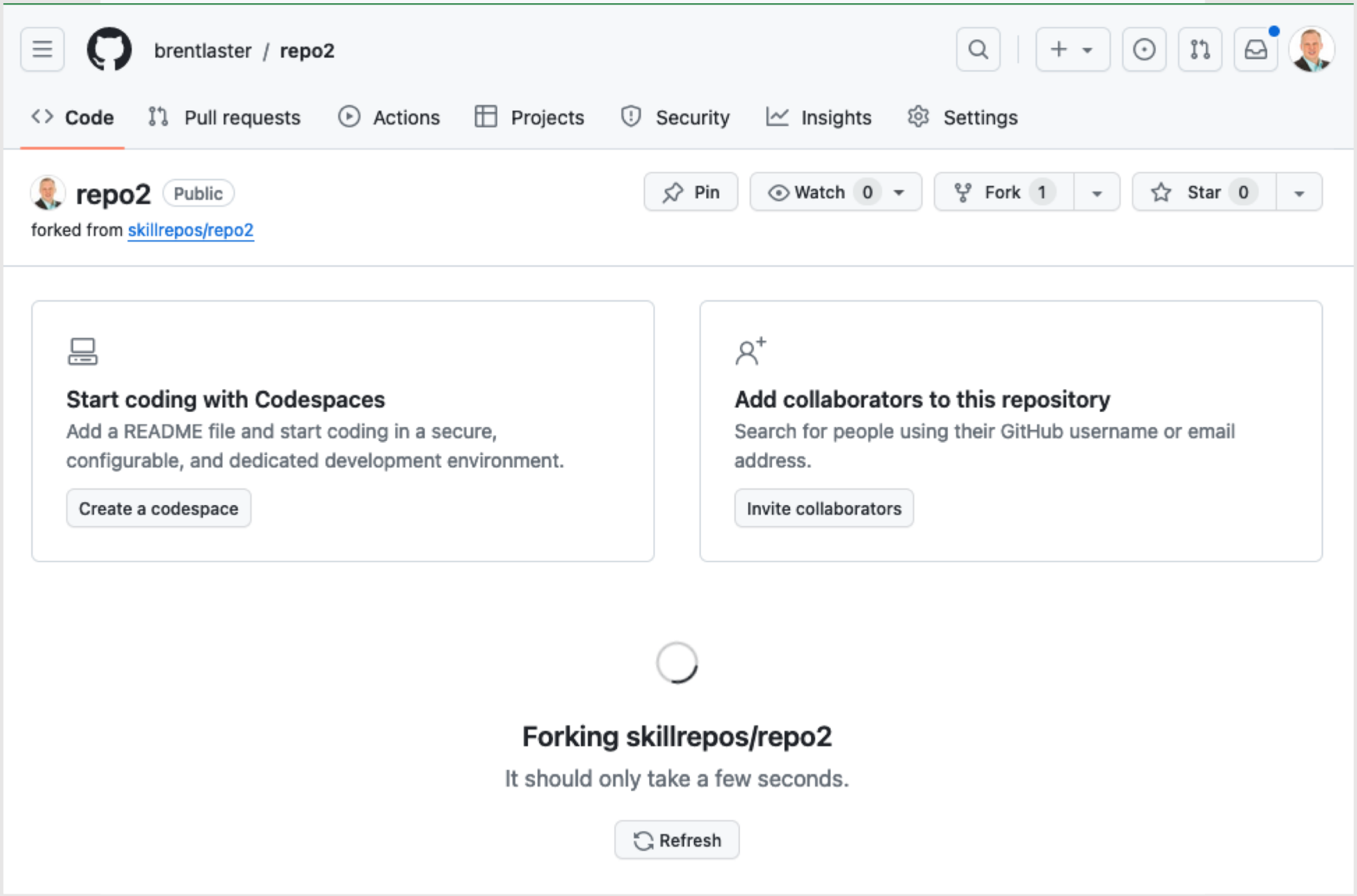

```
$ git clone https://github.com/userid/projectpath
$ git remote -v
origin https://github.com/userid/projectpath(fetch)
origin https://github.com/userid/projectpath (pull)

$ <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)
```



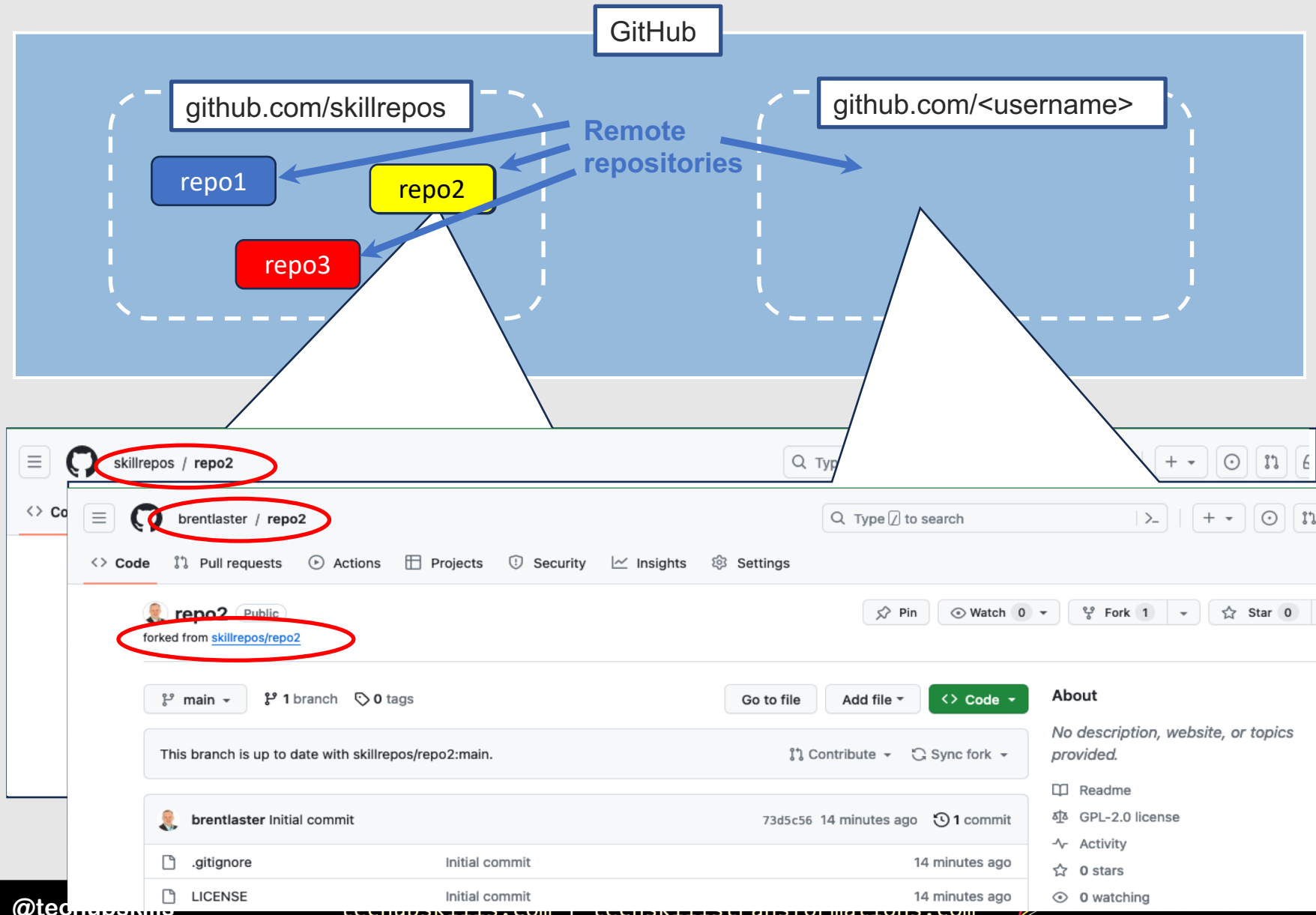
Fork model in GitHub



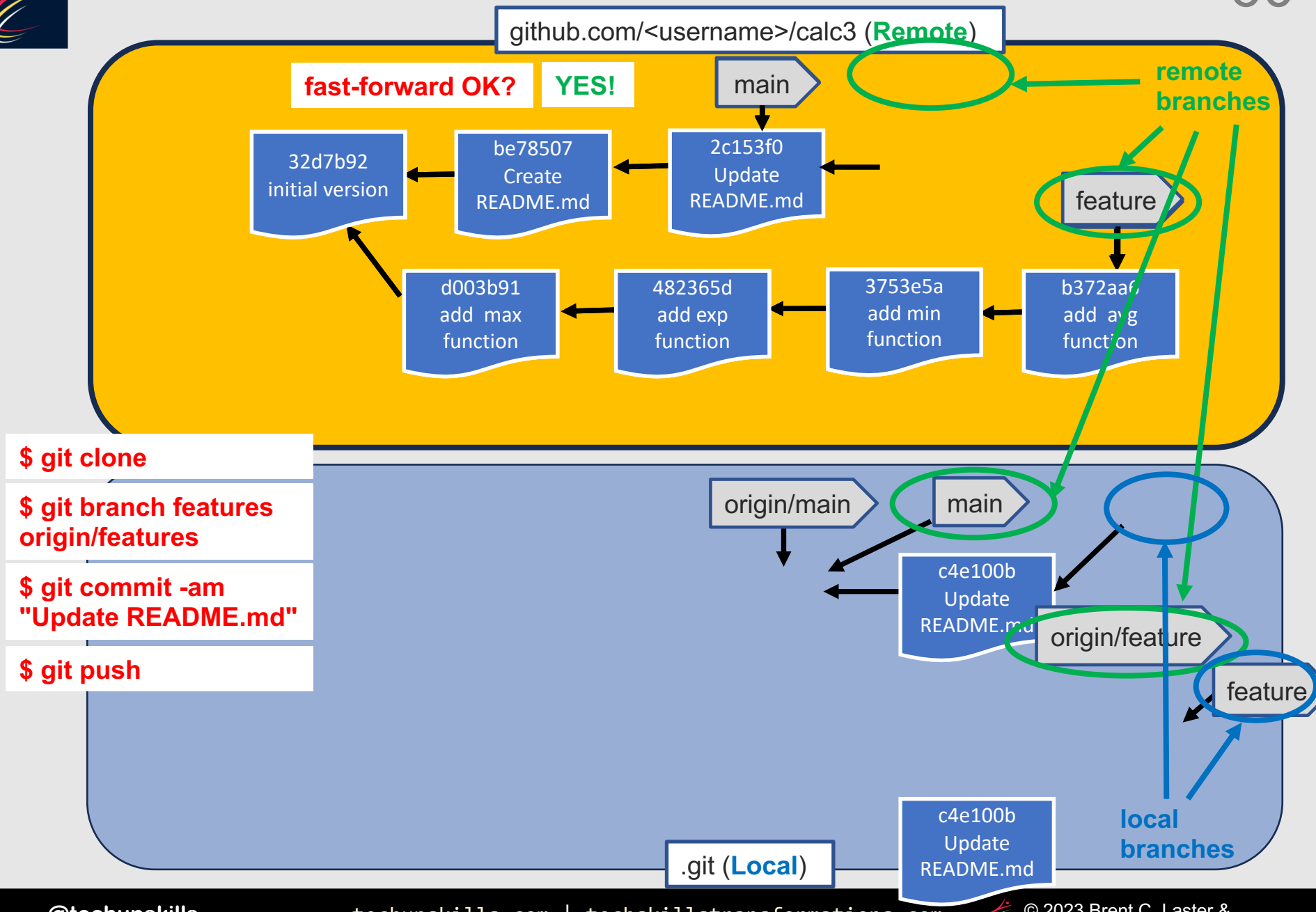
Forking a repository



Fork model in GitHub



Remote vs local branches



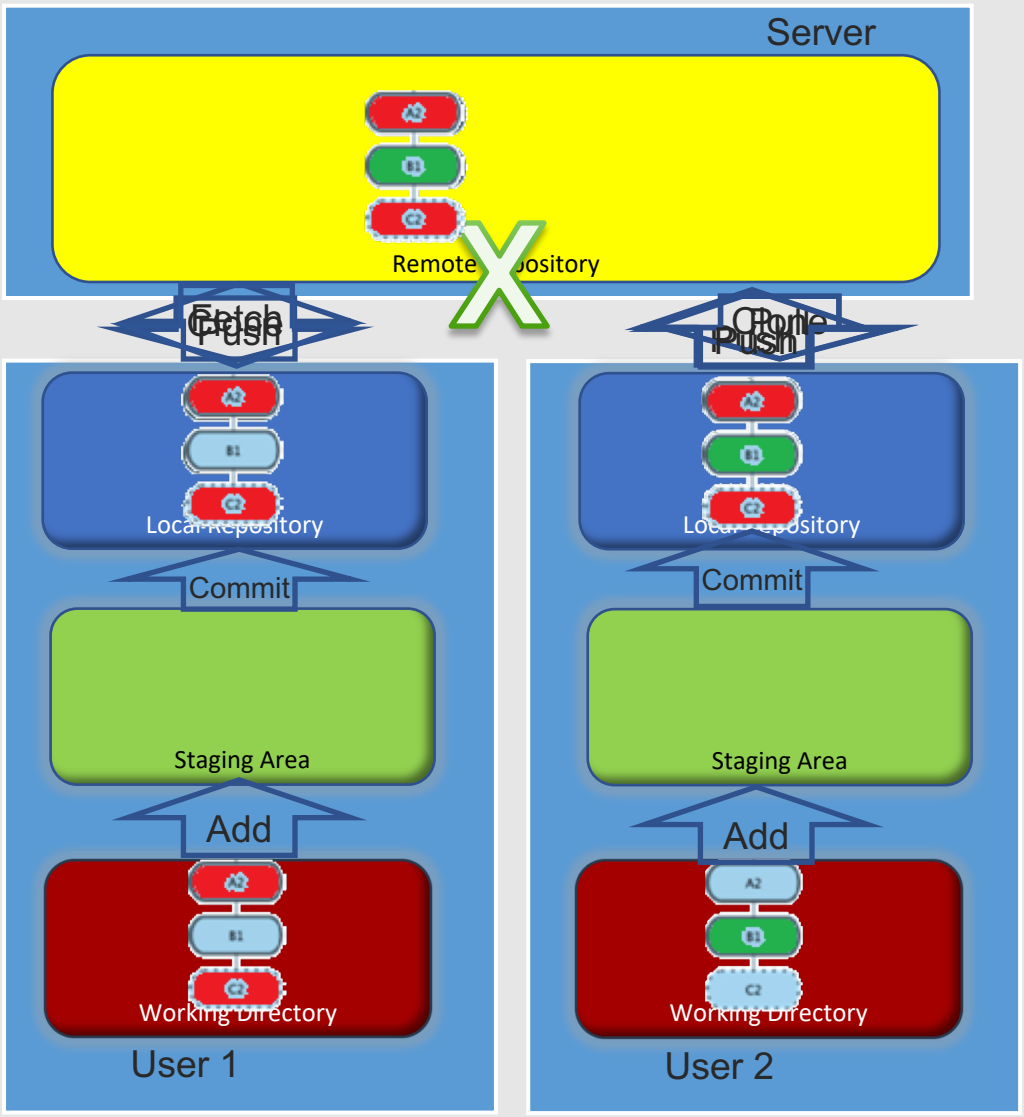
Working with a Remote and Multiple Users

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.

Other useful commands

Removing files

- 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

Renaming Files

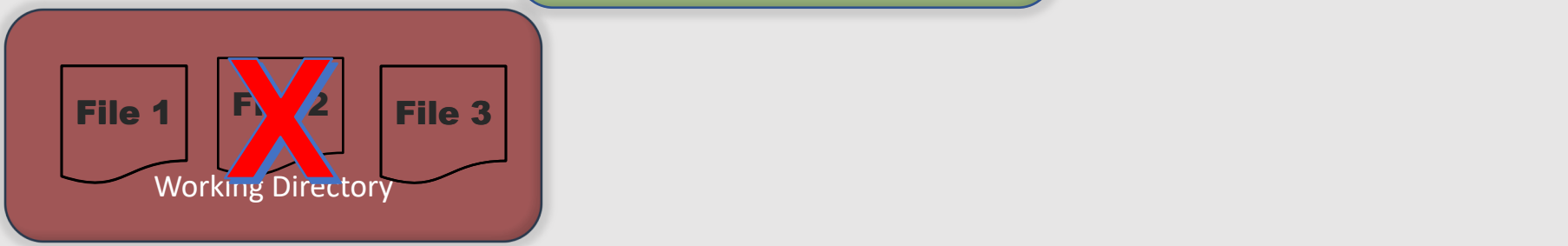
- Git doesn't track metadata about renames, but infers it
- git mv command renames a file locally and stages the change
- Then you commit it
- Git will show "renamed" in status after a mv
- The mv command is equivalent to:
  - mv (old local file) (new local file)
  - git rm old file
  - git add new file

Git Workflow for operations like rm

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

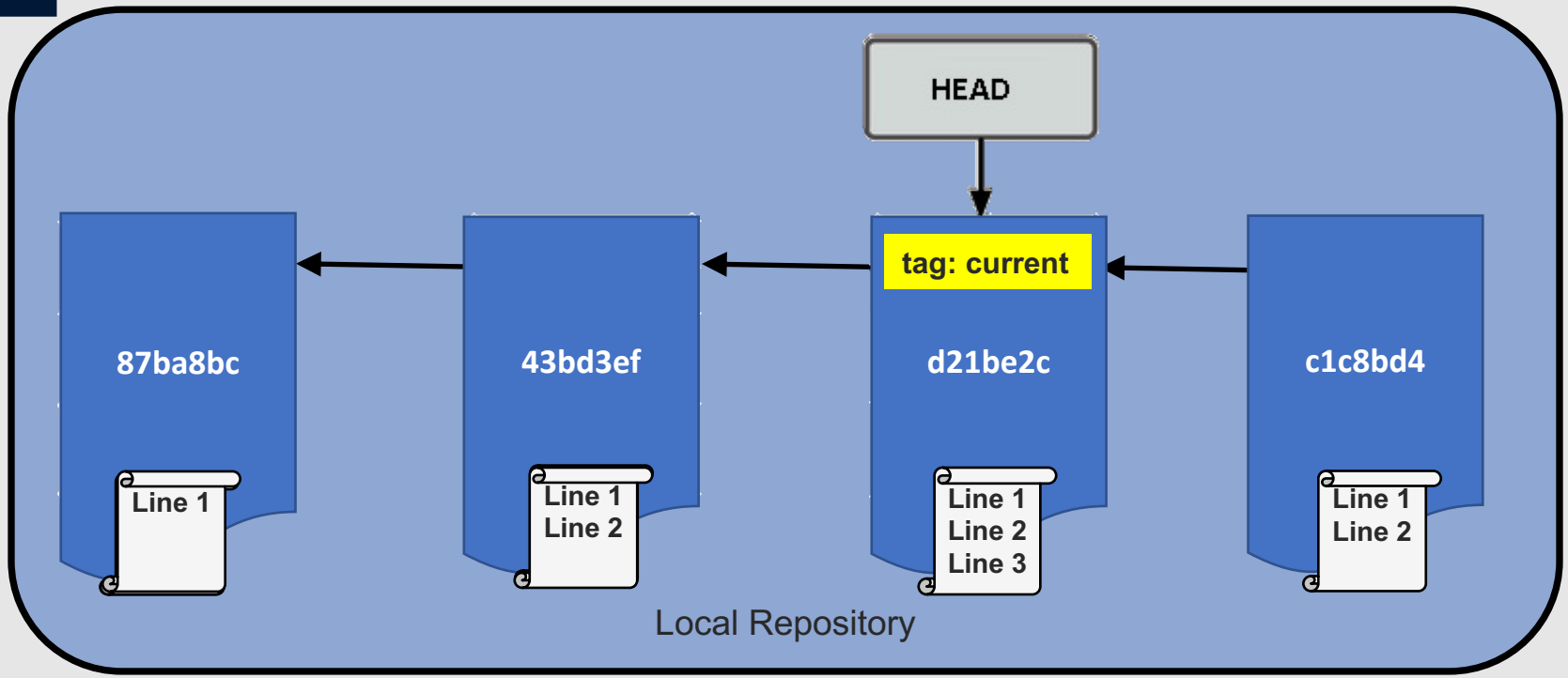
git commit -m "finalize rm"



Rolling back/undoing - Reset and Revert

- 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.

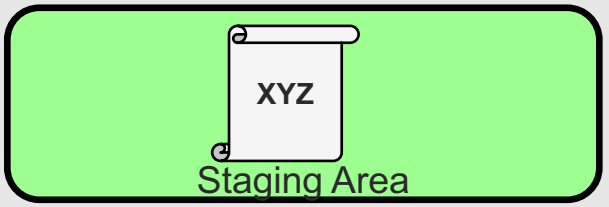
Reset and Revert

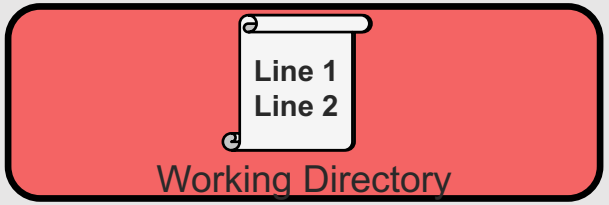


git reset --hard 87ba8bc

git reset current~1 [--mixed]

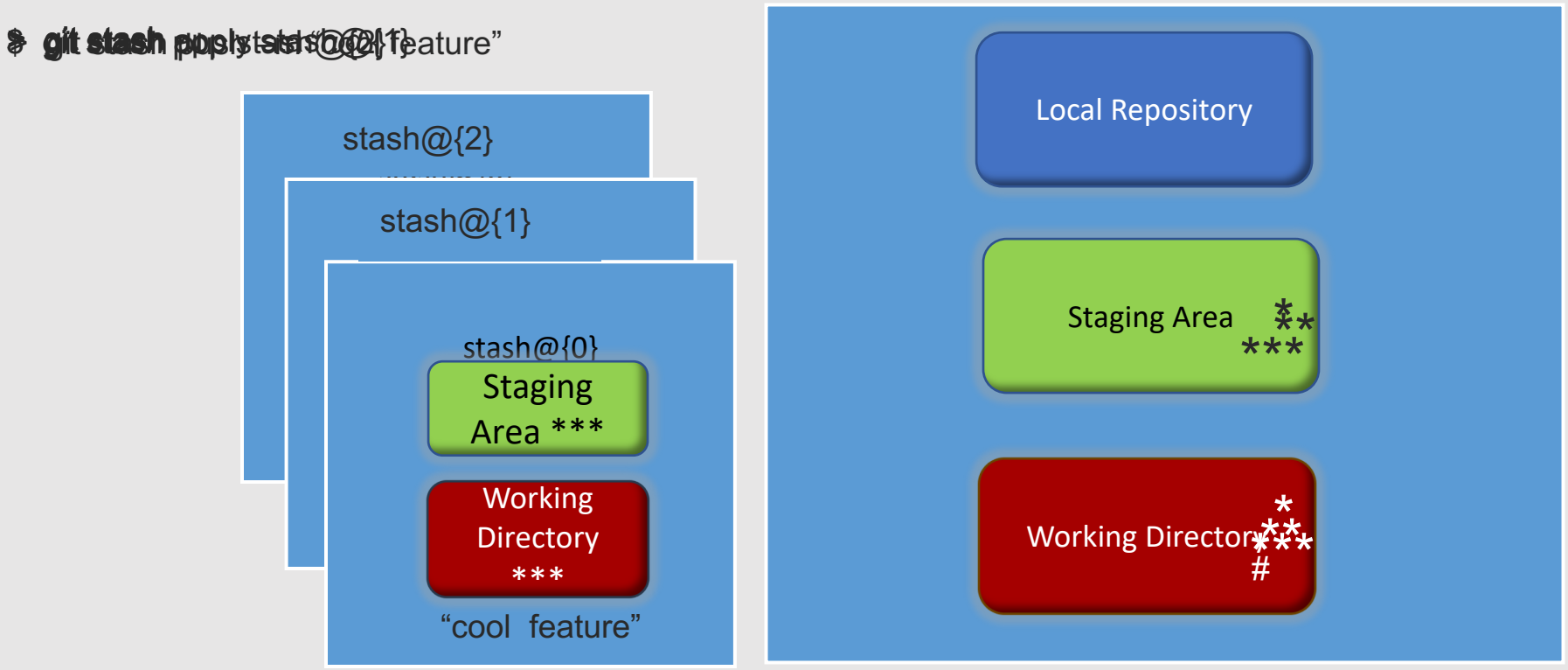
git revert HEAD





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**



That's all - thanks!

techskillstransformations.com
getskillsnow.com