Version 1.6
08/30/23

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 this week's workshop

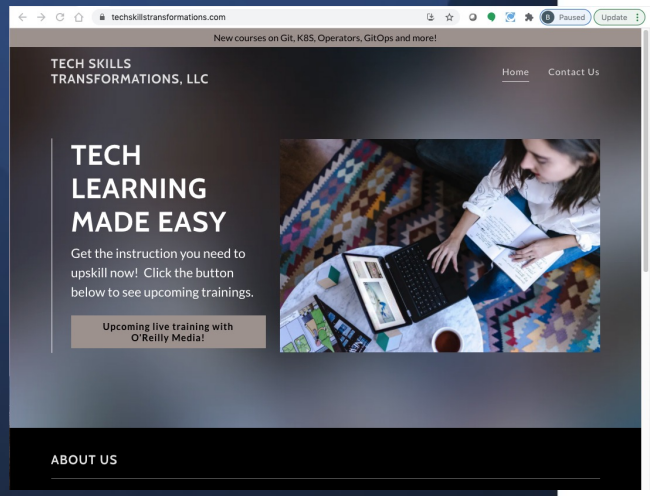
https://github.com/skilldocs/git4/blob/main/git4-1-labs.pdf

Git In Four Weeks
Week 1

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 -
  - 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

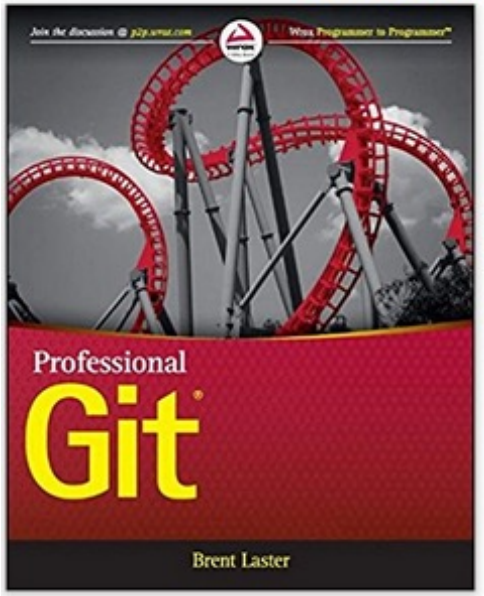
- Extensive Git reference, explanations,
- and examples
- First part for non-technical
- Beginner and advanced reference
- Hands-on labs

Professional Git 1st Edition

by Brent Laster (Author)

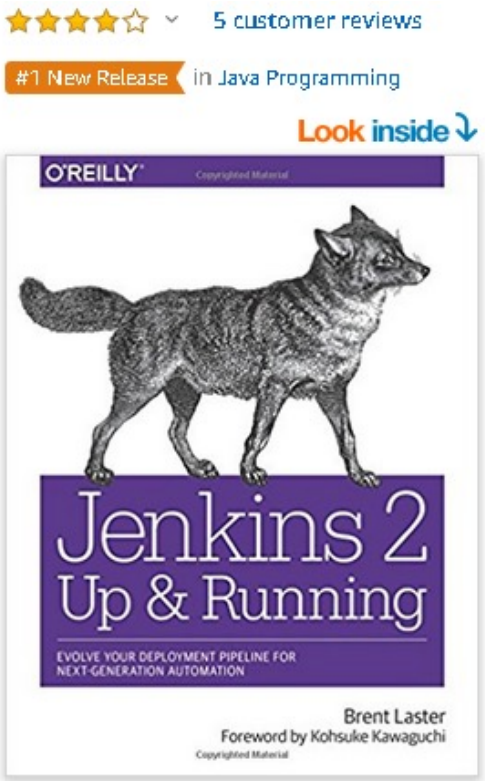
7 customer reviews

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***



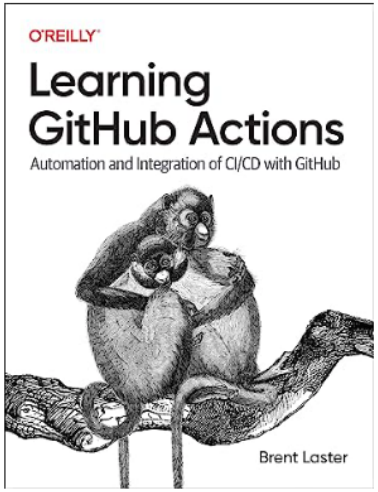
GitHub Actions book

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.

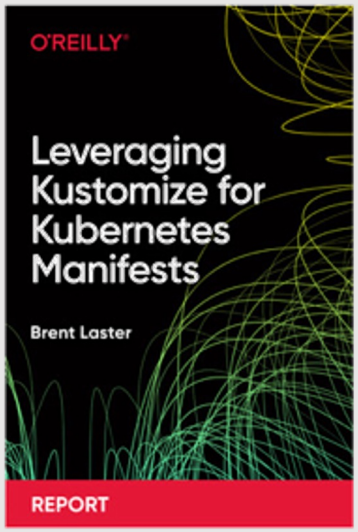
This book explains the platform, components, use cases, implementation, and integration points of actions, so you can leverage them to provide the functionality and features needed in today's complex pipelines and software development processes. You'll learn how to design and implement automated workflows 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.

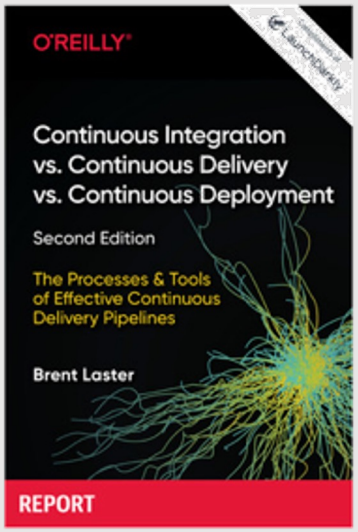
With this book, you will:

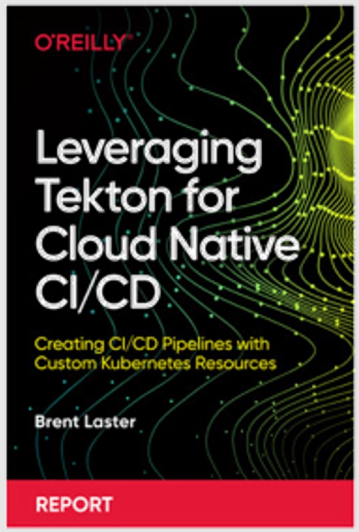
- Learn what GitHub Actions are, the various use cases for them, and how to incorporate them into your processes
- Understand GitHub Actions' structure, syntax, and semantics
- Automate processes and implement functionality
- Create your own custom actions with Docker, JavaScript, or shell approaches
- 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

O'Reilly Reports







O'Reilly Training

April 26 & 27, 2021

Git Fundamentals

Join Brent Laster to gain the knowledge and skills you need to leverage Git to greatly simplify and speed up managing all of the changes in your source code. Once you ...

June 3, 10, 17 & 24, 2021

Git in 4 Weeks

Join author, trainer, and DevOps director Brent Laster to learn how Git works—and discover all the things you can do with it. Over four sessions, Brent walks you through everything you ...

LIVE ONLINE TRAINING

Git Troubleshooting

How to solve practically any problem that comes your way

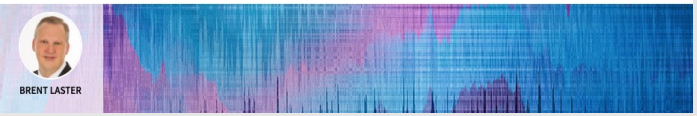
Topic: Software Development

LIVE ONLINE TRAINING

Next Level Git - Master your content

Use powerful tools in Git to simplify merges, rewrite history, and perform automatic updates

Topic: Software Development

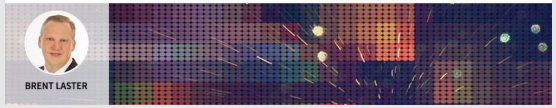


LIVE ONLINE TRAINING

Next Level Git - Master your workflow

Use Git to find problems, simplify working with multiple branches and repositories, and customize behavior with hooks

Topic: Software Development



LIVE ONLINE TRAINING

Getting started with continuous delivery (CD)

Move beyond CI to build, manage, and deploy a working pipeline

Topic: System Administration

LIVE ONLINE TRAINING

Building a deployment pipeline with Jenkins 2

Manage continuous integration and continuous delivery to release software

Topic: System Administration

June 28, 2021

Troubleshooting Kubernetes

In this 3-hour course, global trainer, author, and DevOps director Brent Laster will show you how to respond to the most common problem situations you may encounter with Kubernetes. You'll learn ...

LIVE ONLINE TRAINING

Helm Fundamentals

Deploying, upgrading, and rolling back applications in Kubernetes

Topic: System Administration

LIVE ONLINE TRAINING

Containers A-Z

An overview of containers, Docker, Kubernetes, Istio, Helm, Kubernetes Operators, and GitOps

Topic: System Administration

May 24, 2021

Continuous Delivery in Kubernetes with ArgoCD

Join expert Brent Laster to explore GitOps and learn how to use Argo CD to implement GitOps in your Kubernetes deployments. APAC time friendly - You're a Kubernetes admin who wants ...

May 17, 2021

Building a Kubernetes Operator

Join expert Brent Laster to learn how the Operator pattern helps address these kinds of situations by allowing you to create custom controllers that extend the functionality of the Kubernetes API ...

What is Git?

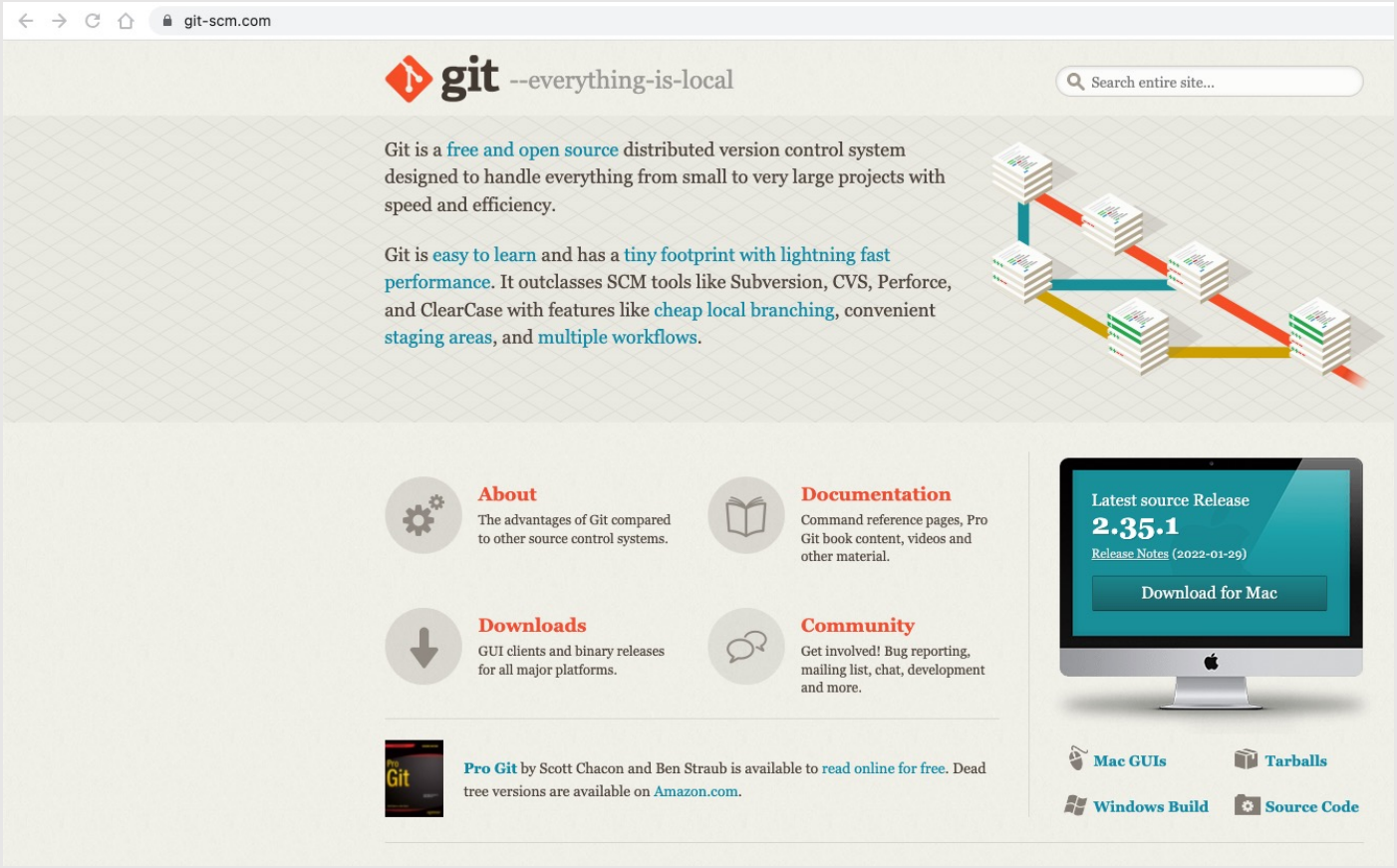
- Distributed Version Control System
- Open source
- Available for multiple platforms
- Roots in source control efforts for Linux Kernel
- Primary developer – Linus Torvalds
- Been around since 2005

Git Design Goals

- Speed
- Simple design
- Strong support for branching
- Distributed nature
- Ability to handle large projects efficiently
- Disconnected support

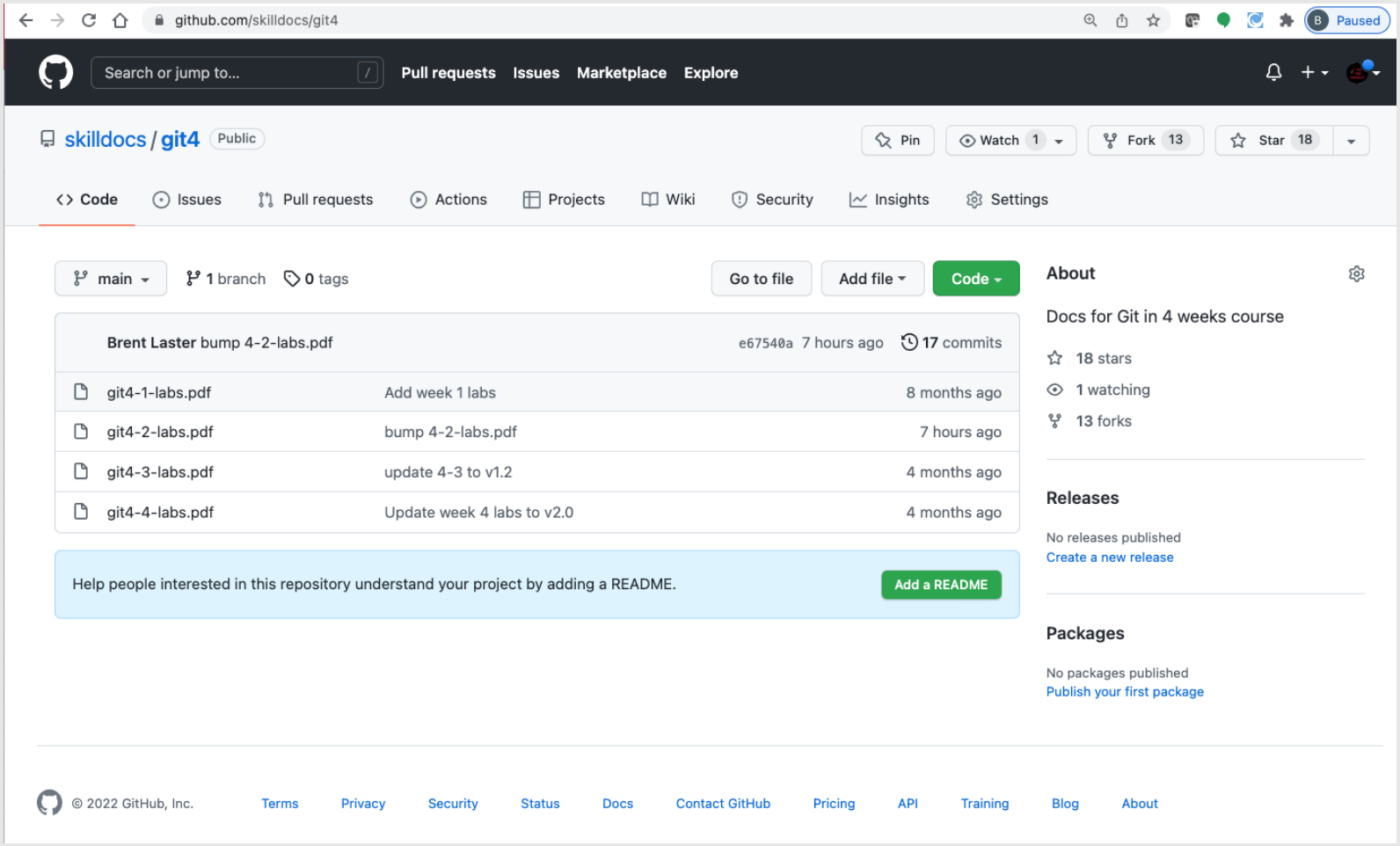
Installation

- Installs available for common platforms through package managers/downloads
- Can also go to git-scm.com



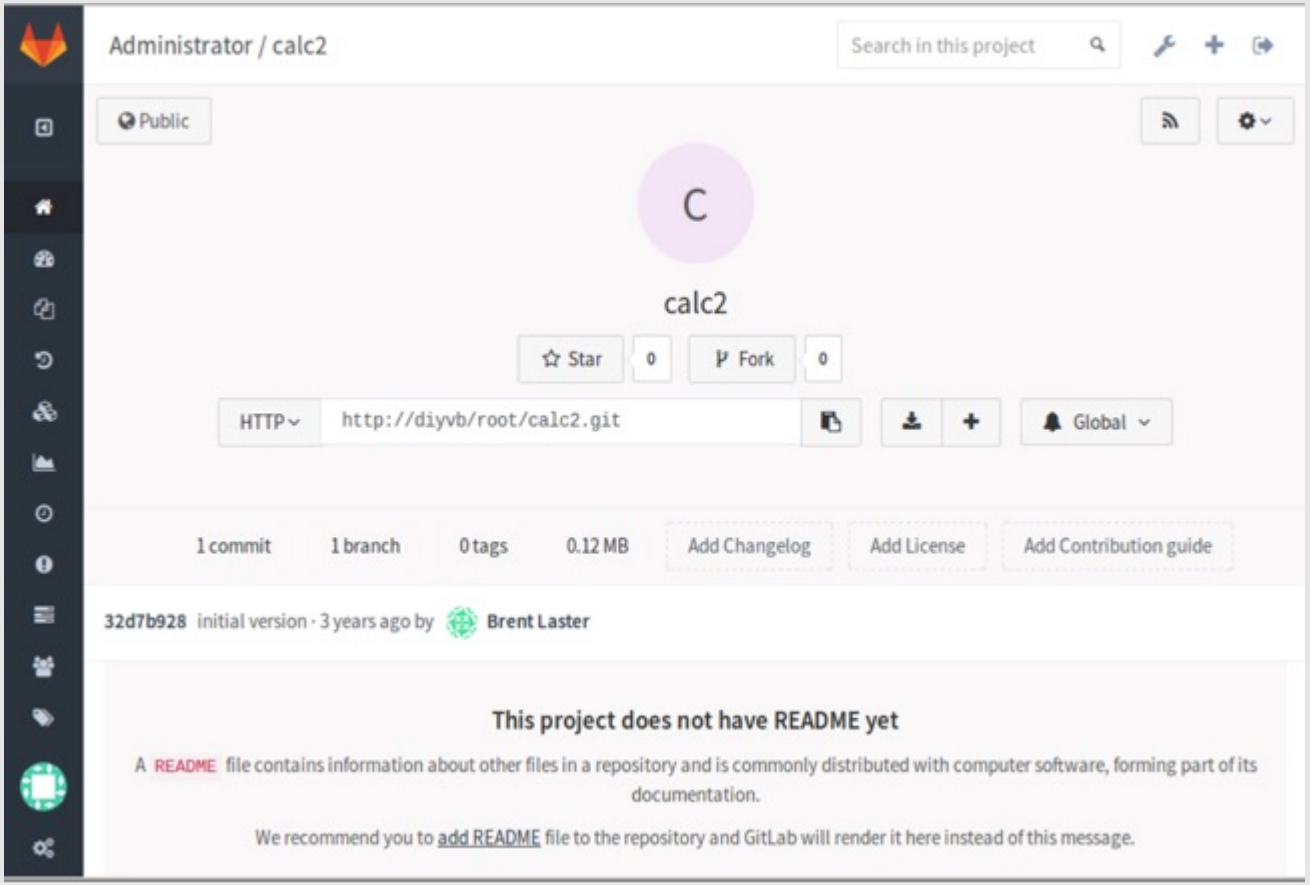
Ecosystem: Public hosting

- GitHub, Bitbucket, etc.

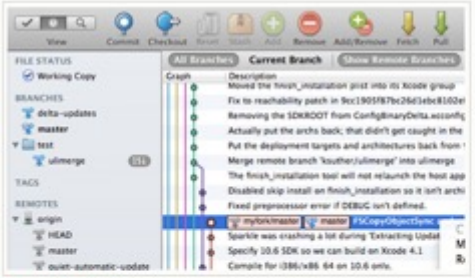


Ecosystem: Self-hosting

- GitLab, Enterprise GitHub



Ecosystem: Ease-of-use Packages

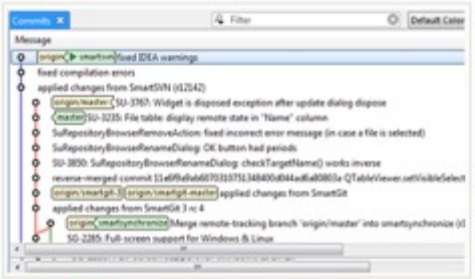


SourceTree

Platforms: Mac, Windows
Price: Free

Git Extensions

Platforms: Windows
Price: Free



SmartGit

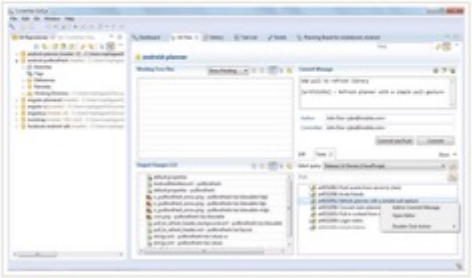
Platforms: Windows, Mac, Linux
Price: $79/user / Free for non-commercial use

git-cola

Platforms: Windows, Mac, Linux
Price: Free

GitUp

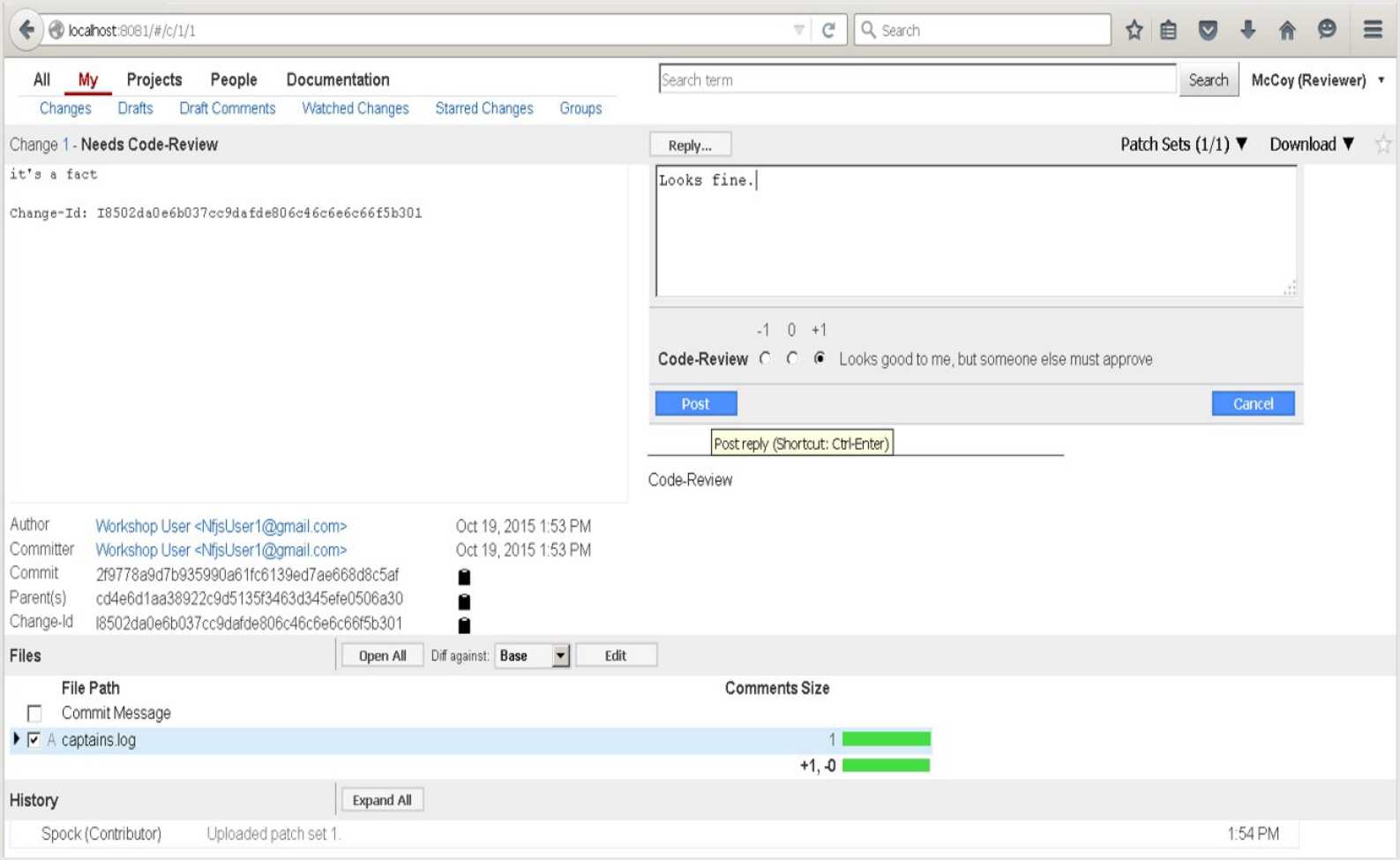
Platforms: Mac



GitEye

Platforms: Windows, Mac, Linux

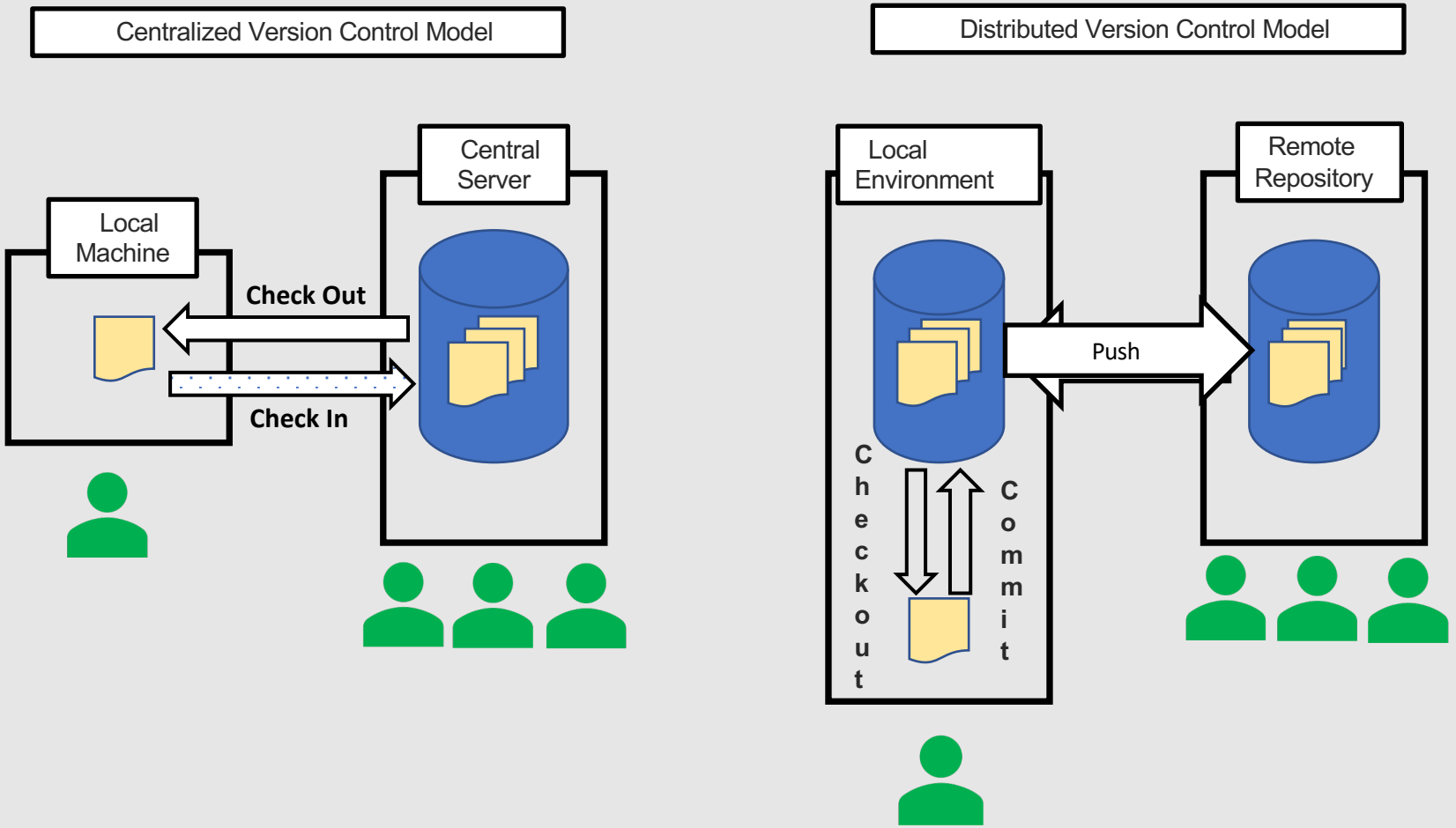
Ecosystem: Tools that Incorporate Git



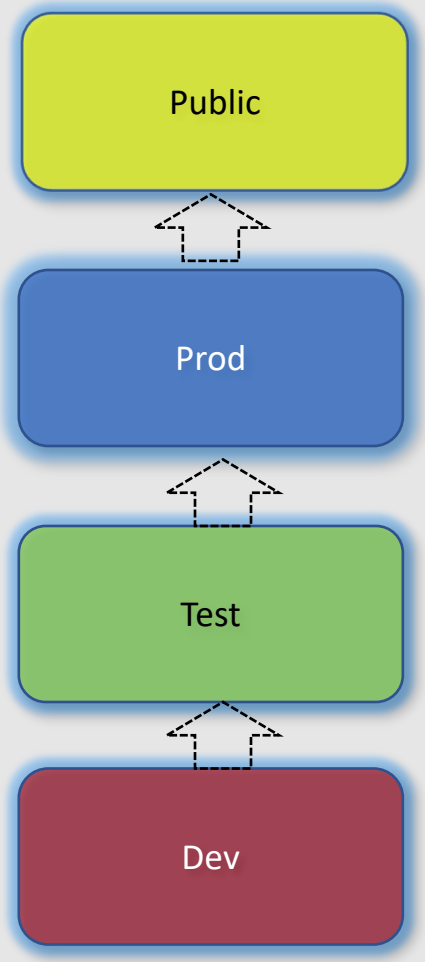
Git Interfaces

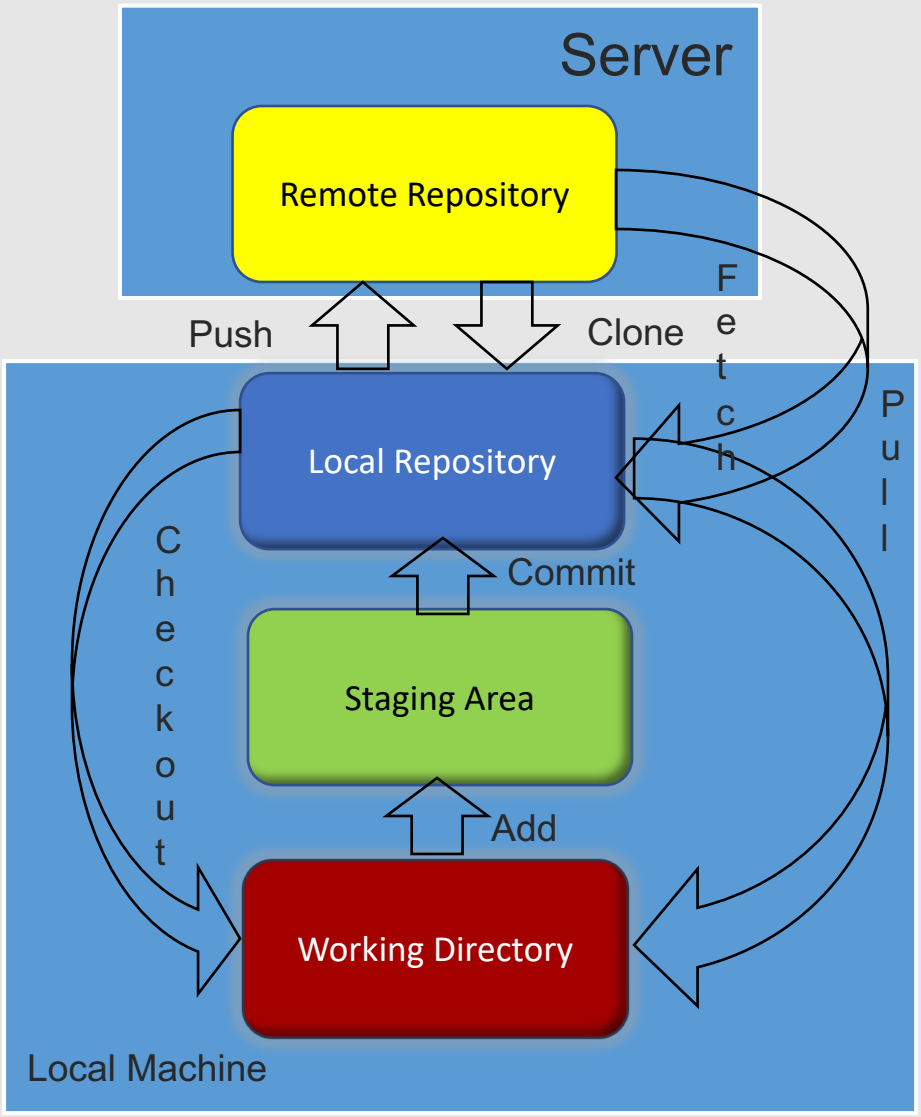
- Many different interfaces exist for working with Git
- Many GUI interfaces
- Command line interface
- Products that script the command line
- Other products such as Gerrit that wrap around Git
- We teach the command line interface because
  - GUI interfaces can all be different - there is not a standard
  - GUI interfaces may change or go away
  - If a GUI interface does not provide some piece of Git functionality, the functionality can be done with the command line
  - If you understand the command line, you can usually find which command in a GUI is used to do the operation

Centralized vs. Distributed VCS



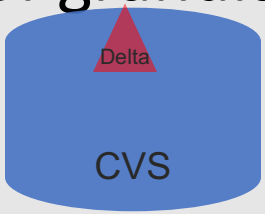
Git in One Picture



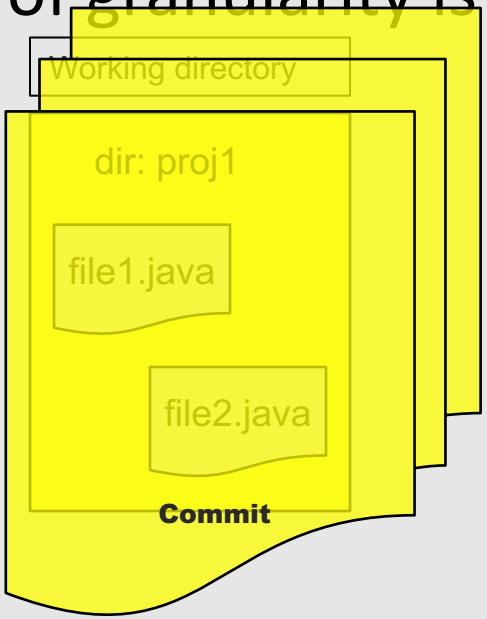


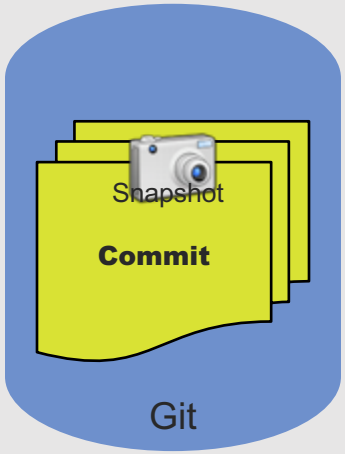
Git Granularity (What is a unit?)

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

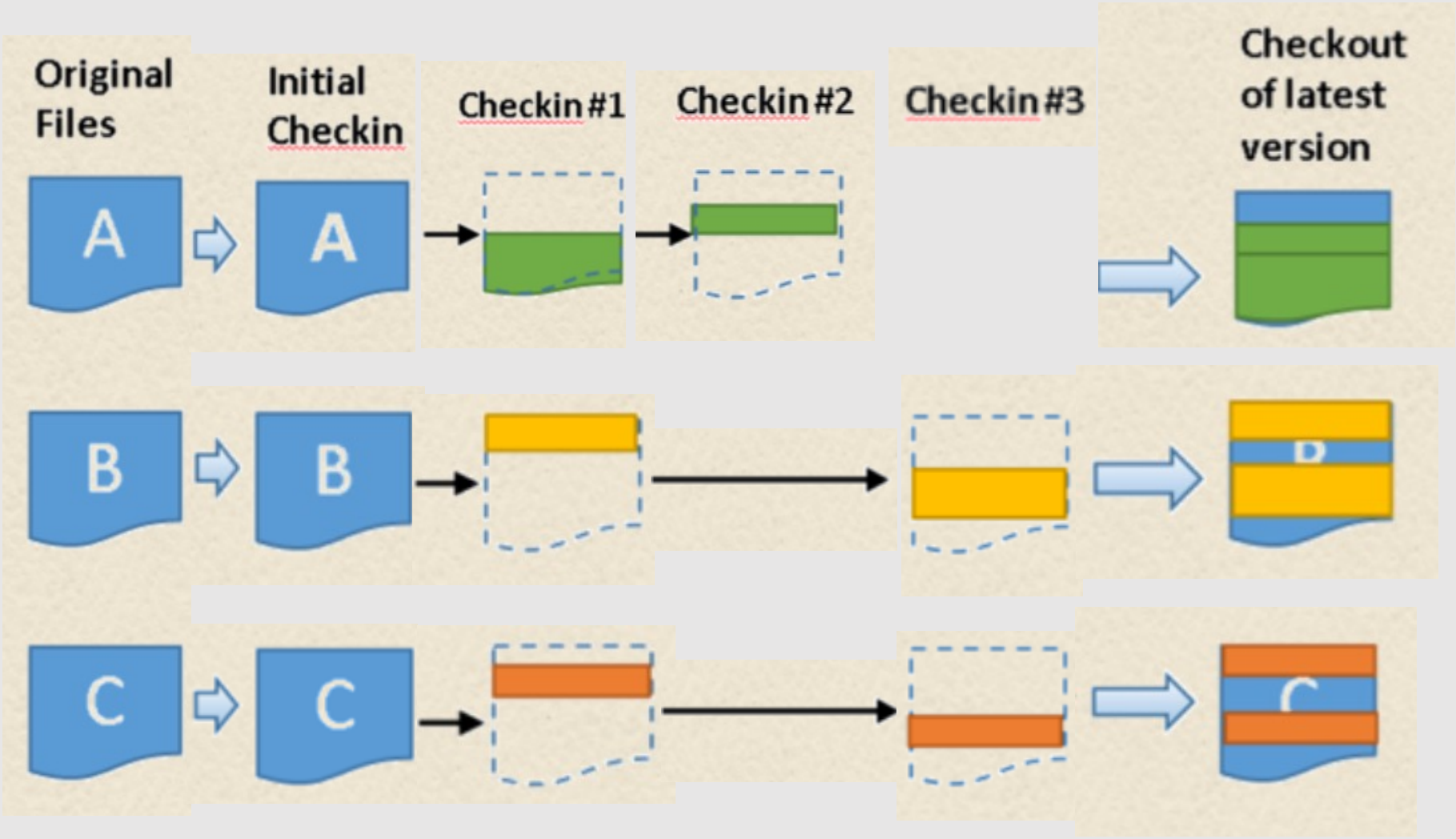


- In Git, the unit of granularity is usually a tree

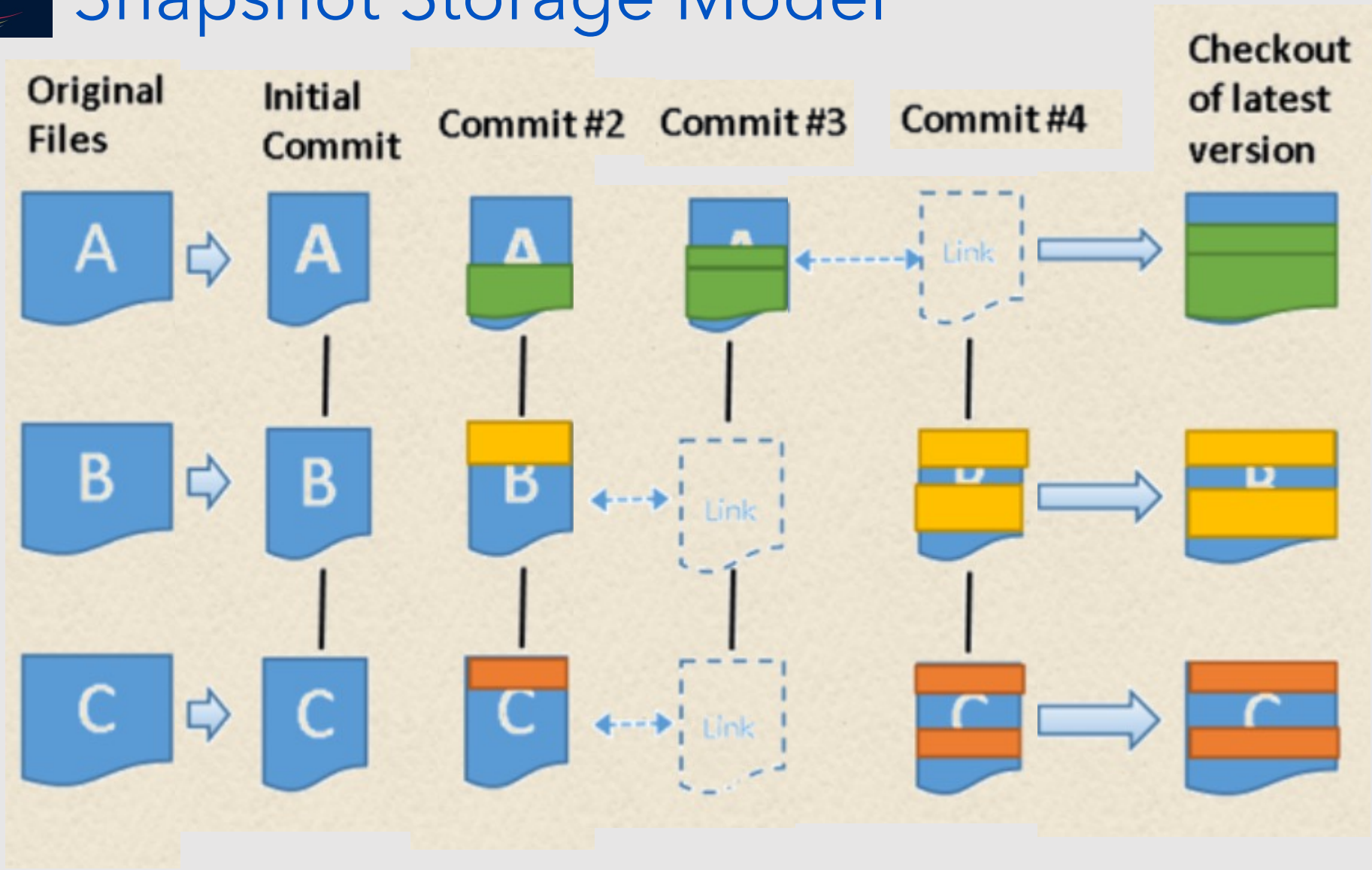




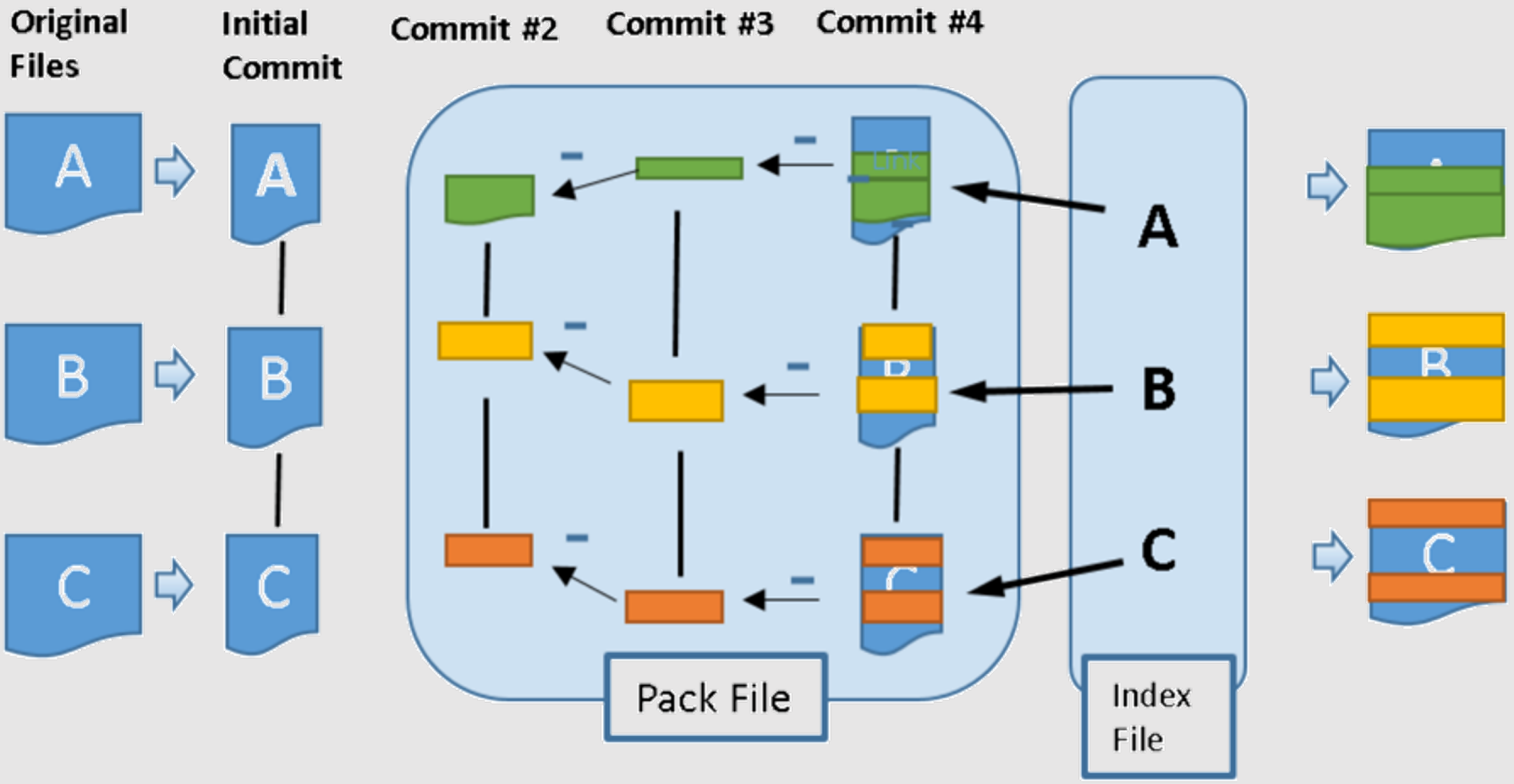
Delta Storage Model



Snapshot Storage Model



Compressing Files



Staging Area: Why?

Prepare

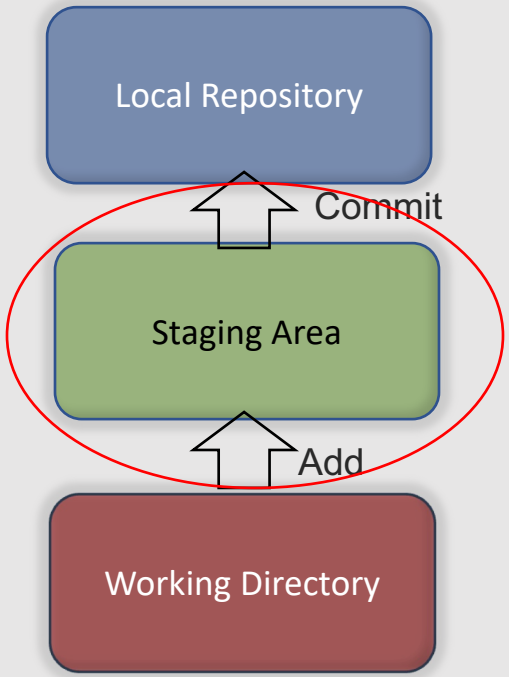
- *Stores info about what will go into next commit*
- *Allows you to control what part of the working tree go into the next commit*
- *Allows you to queue up individual changes*
  - *Bundle a collection of specific changes to go into a commit*
  - *Get them out of your working directory (i.e. "check them off")*
- *Helps you split up one large change into multiple commits*

Repair

- *Allows you to "amend" last commit – redo*

Merging with conflicts

- *When merge happens, changes that merge cleanly are updated both in working directory and in staging area*
- *Changes that did not merge cleanly are left in your working directory*
- *Diff between working directory and staging area shows differences*



Example Workflow and Terms

1. Modify files in working directory
2. Stage files, adding snapshot to staging area
3. Commit – promotes files from the staging area into permanent snapshot in the local repo (stored in .git directory)

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

SHA1


```
$ git commit -m "initial add of file1.c"
[main b41b186] initial add of file1.c
 1 file changed, 3 insertions(+)
 create mode 100644 file1.c
```

- Everything in Git is check-summed before it is stored and is then referred to by that checksum. Built into Git low-level functions.
- That means it's not possible to change the contents of any file or directory without Git knowing about it. i.e. avoids file corruption or tampering.
- Mechanism that Git uses for this checksum is called a SHA-1 hash.
- 40-character string of hex characters.
- Calculated based on contents of a file or directory struture.
- Example: b41b186552252987aa493b52f8696cd6d3b00373
- Used throughout Git.
- Git stores/points to everything by hash value of the contents in its database.

Remember: You can always get a handle to a snapshot via a SHA1**.**

Committing Changes

- Commits to local repo only - not remote
- Records a snapshot of your local working area that you can switch back to or compare to later
- Requires commit message
  - Pass by –m “<message>”
  - If no –m, brings up configured editor so you can type it
- Commit output

  - which branch you committed to
  - what SHA-1 checksum the commit has
  - how many files were changed
  - statistics about lines added and removed in the commit.

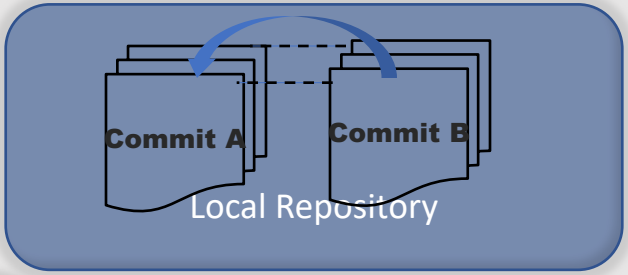
Git Workflow

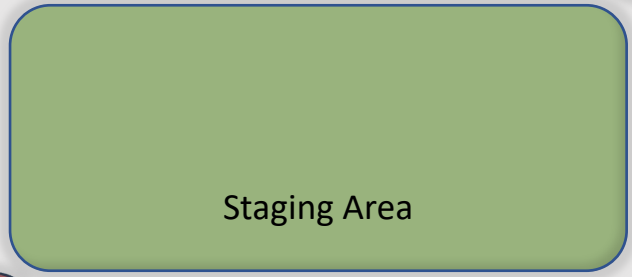
Staging Area Use: Prepare…

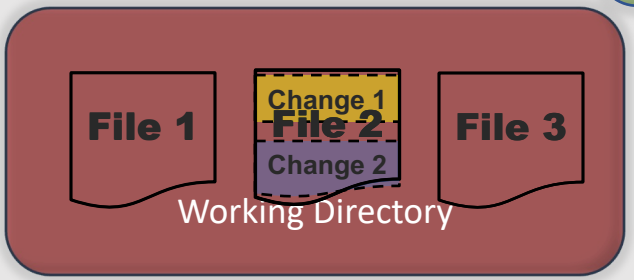
git commit -m "<comment>"

git add .

git add -p







Undoing/Updating Things

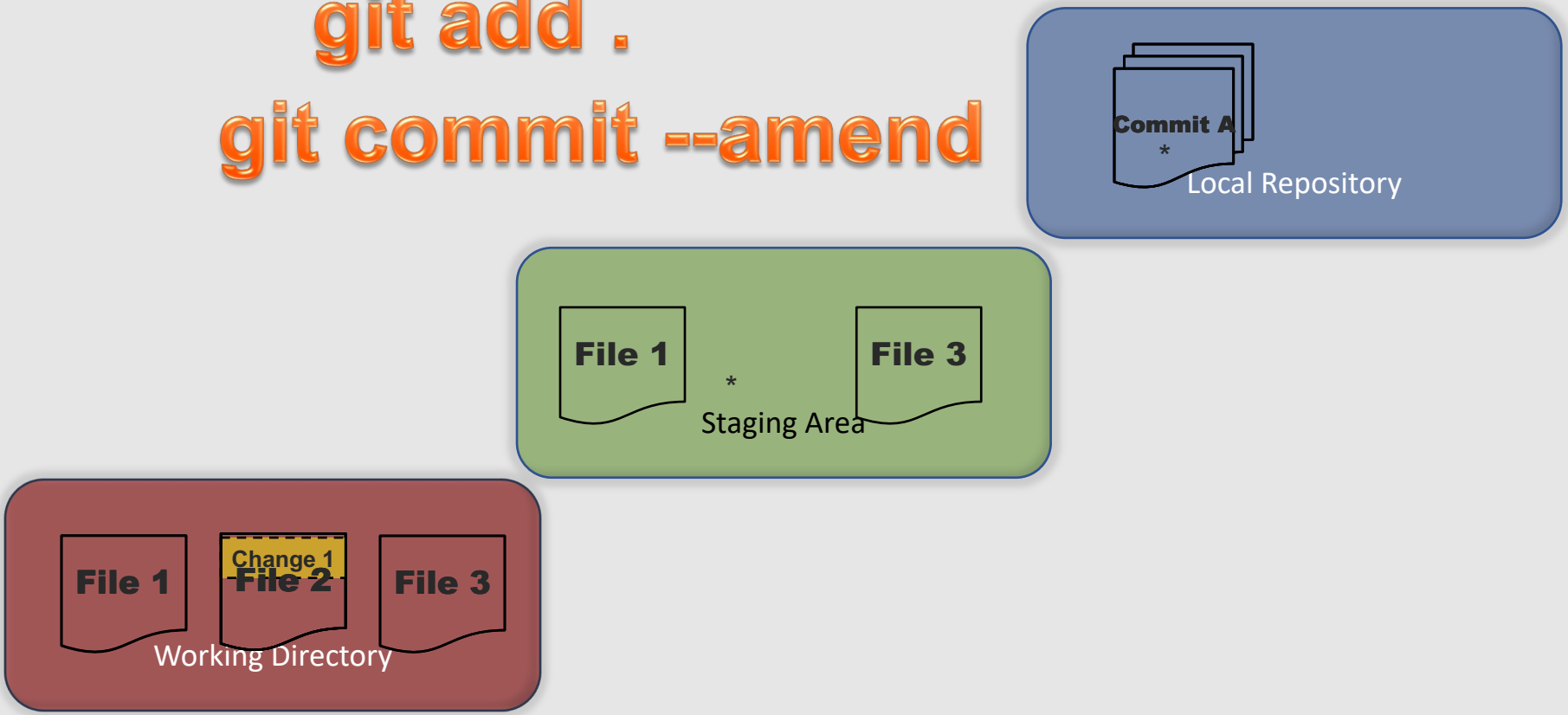
- git commit --amend
  - Allows you to “fix” last commit
  - Updates last commit using staging area
  - With nothing new in staging area, just updates comment (commit message)
  - Example:
    - » git commit –m “my first update”
    - » git add newfile.txt (add command stages it into staging area)
    - » git commit --amend
    - » This updated commit takes the place of the first (updates instead of adding another in the chain)

Git Workflow

Staging Area Use:Repair...

git add .

git commit --amend



Updating a file that Git knows about

- How do we update a file that git knows about?
  - Stage it (git add .)
  - Commit it (git commit)
  - Shortcut syntax : git commit –am “<comment>”
    - » Doesn’t add new files (untracked ones)

Initial User Configuration

- Command: git config
- Main things to set
  - User name

    $ git config --global user.name "John Doe"
  - Email address

    $ git config --global user.email johndoe@example.com

- Notes
  - Use --global to persist options for all repos of this user
  - Check config value with
    - *$ git config --list (or git config –l)*
    - *$ git config <property> as in git config user.name*
  - Can also set many other things – like diff tool and end-of-line settings

Configuring your default editor in Git

- Editor defaults (generally)
  - Windows : notepad, Linux : vim
- Can set via OS (for example export EDITOR variable)
- Setting for Git only
  - Set core.editor in your config file: **git config --global core.editor “vim”**
  - Set the GIT_EDITOR environment variable: **export GIT_EDITOR=vim**
- Examples
  - git config core.editor vim (Linux)
  - git config --global core.editor “nano” (mac)
  - git config --global core.editor "'c:\program files\windows nt\accessories\wordpad.exe'"(windows)
  - git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe'"
  - (Git bash shell for Windows)

Labs Logistics

Work through the steps – some questions for thought included

Creating Files

In the workshop, we're not concerned about contents of files.

We just need some files to play with.

Lazy way to create a file – in a terminal:

$ echo stuff > file1.c (even lazier way $ date > file1.c)

Lazy way to append to a file – in a terminal:

$ echo "more stuff" >> file1.c

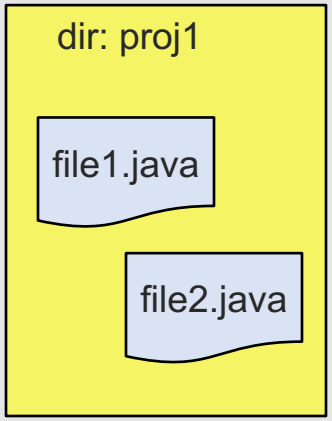
Can use editor if you prefer…

Lab 1 - Creating and Exploring a Git Repository and Managing Content

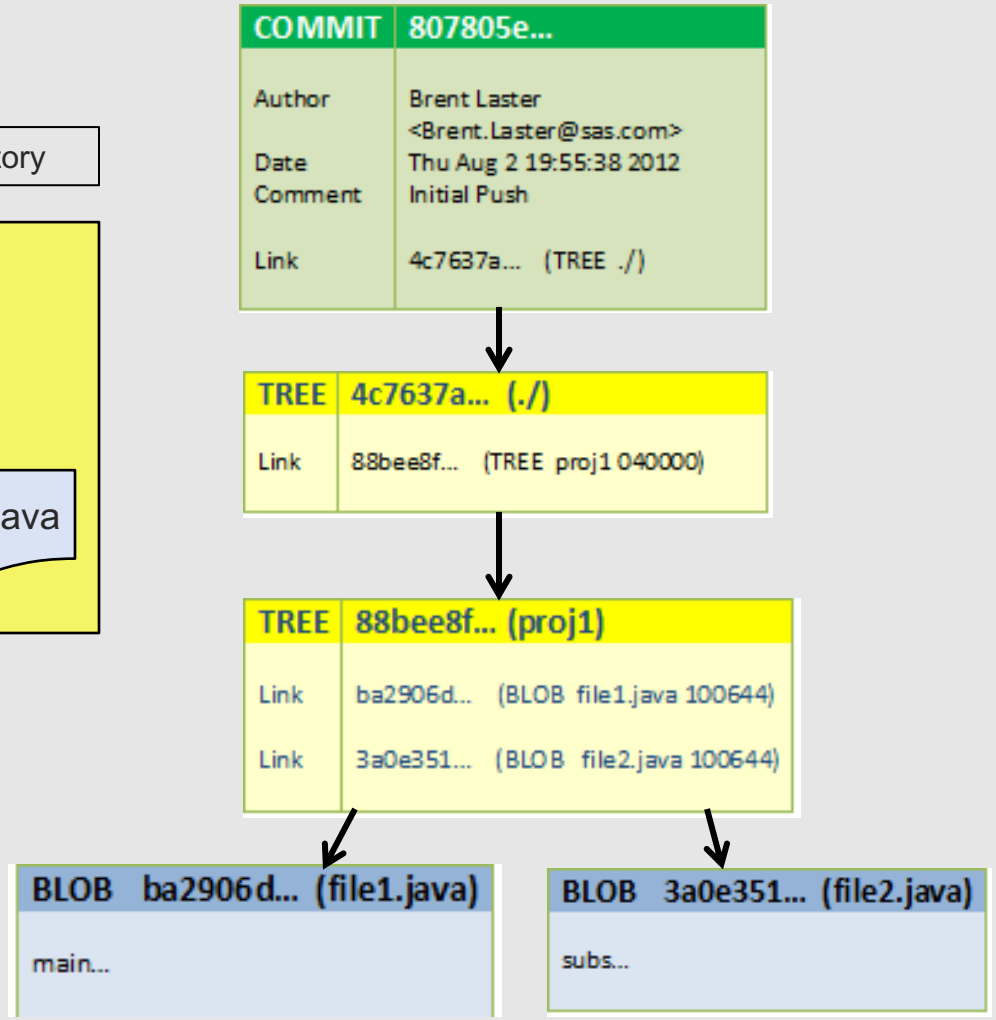
Purpose: In this lab, we'll create an empty Git repository on your local disk and stage and commit content into it.

Git and File/Directory Layouts

Working directory



Git snapshot

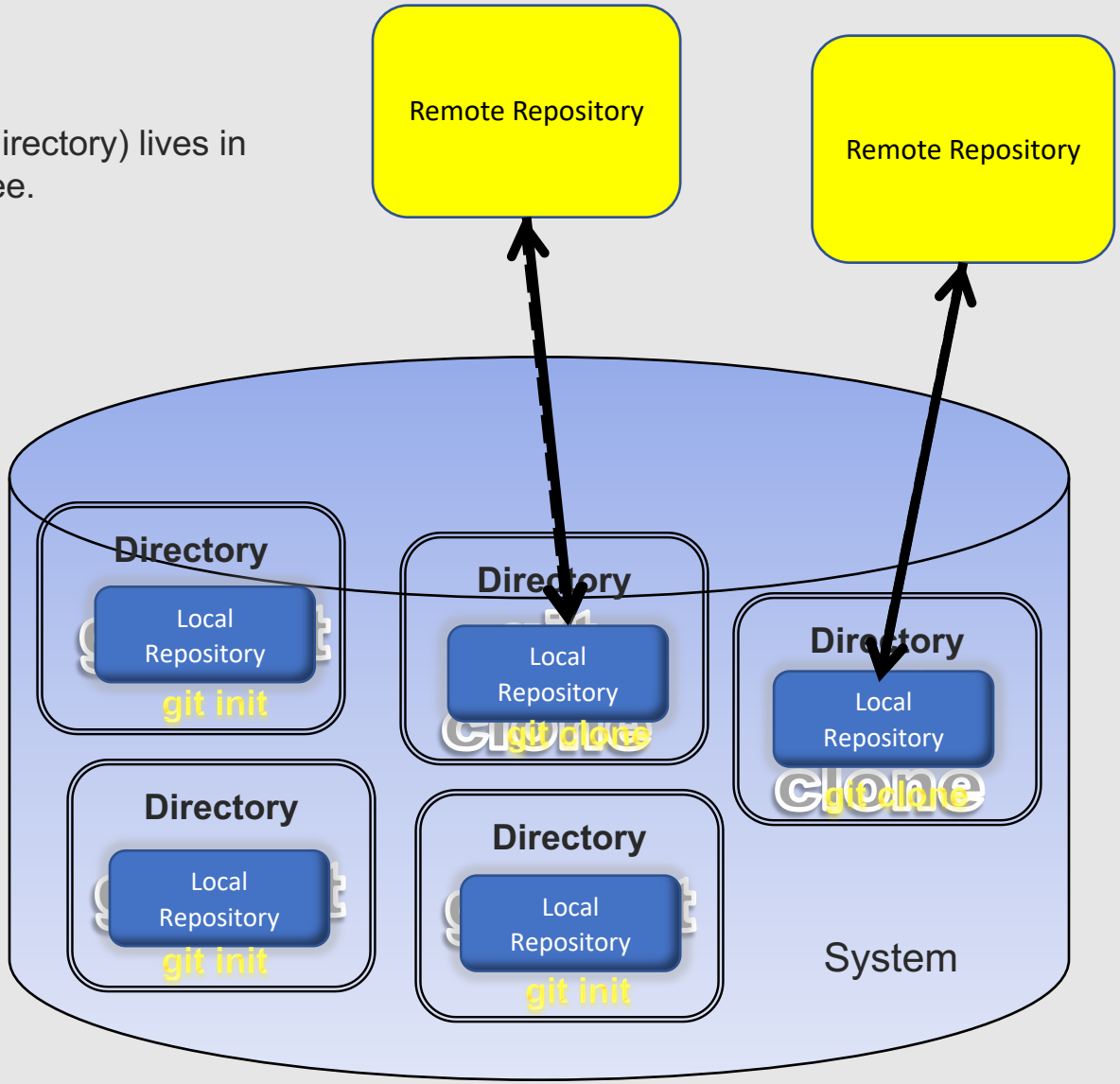


Local Repository File Layout

```
$ 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
                *
```

Starting Work with Git – Multiple Repositories

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



Autocomplete

- Enabled in BASH shell
- Can be enabled in other environments via file in git source
- Git <start of command><tab> (may have to hit it twice if multiple options)
- Git command --<start of option><tab>
- Examples:
  - git co <tab><tab>
  - git com <tab>
  - git list --s<tab> (note double –'s)

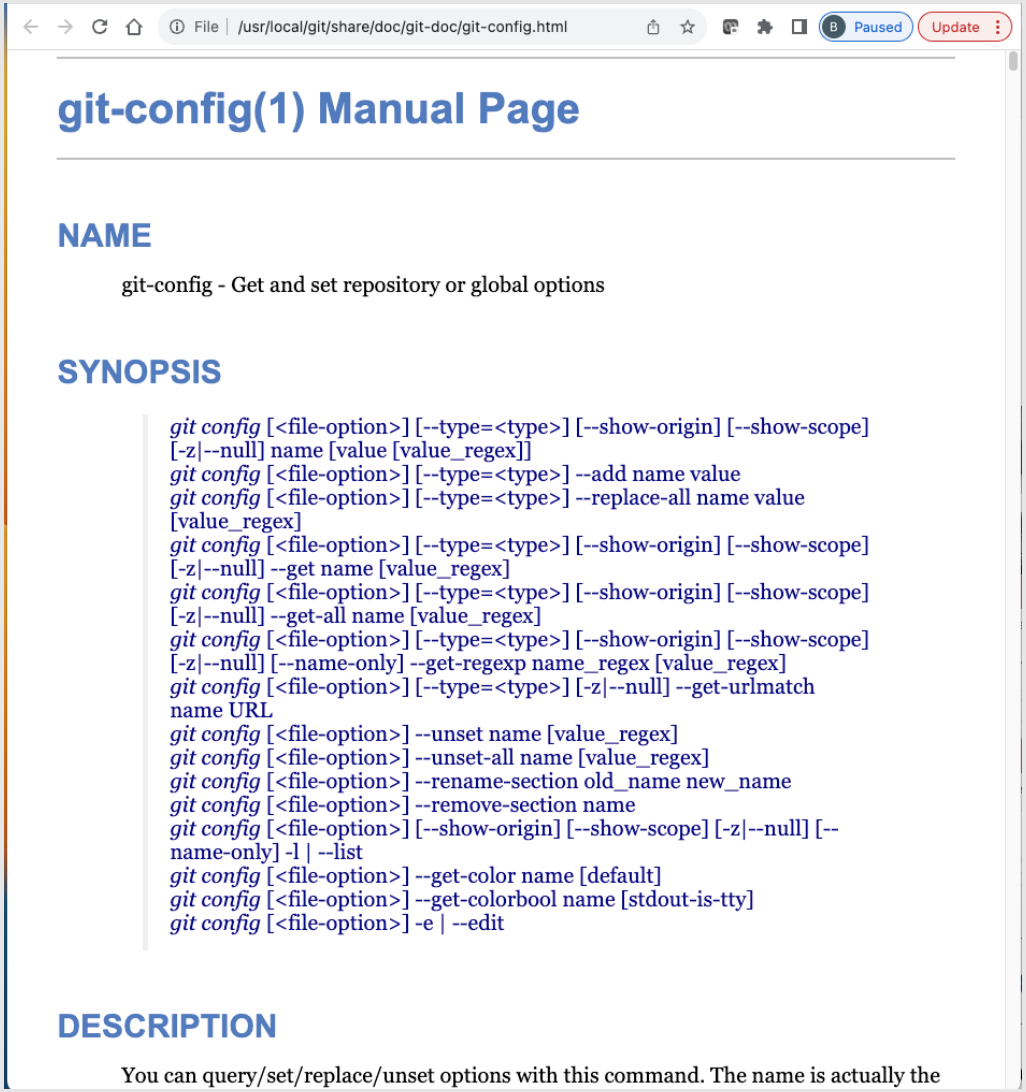
Getting Help

- git <command> -h

Brings up on screen list of options

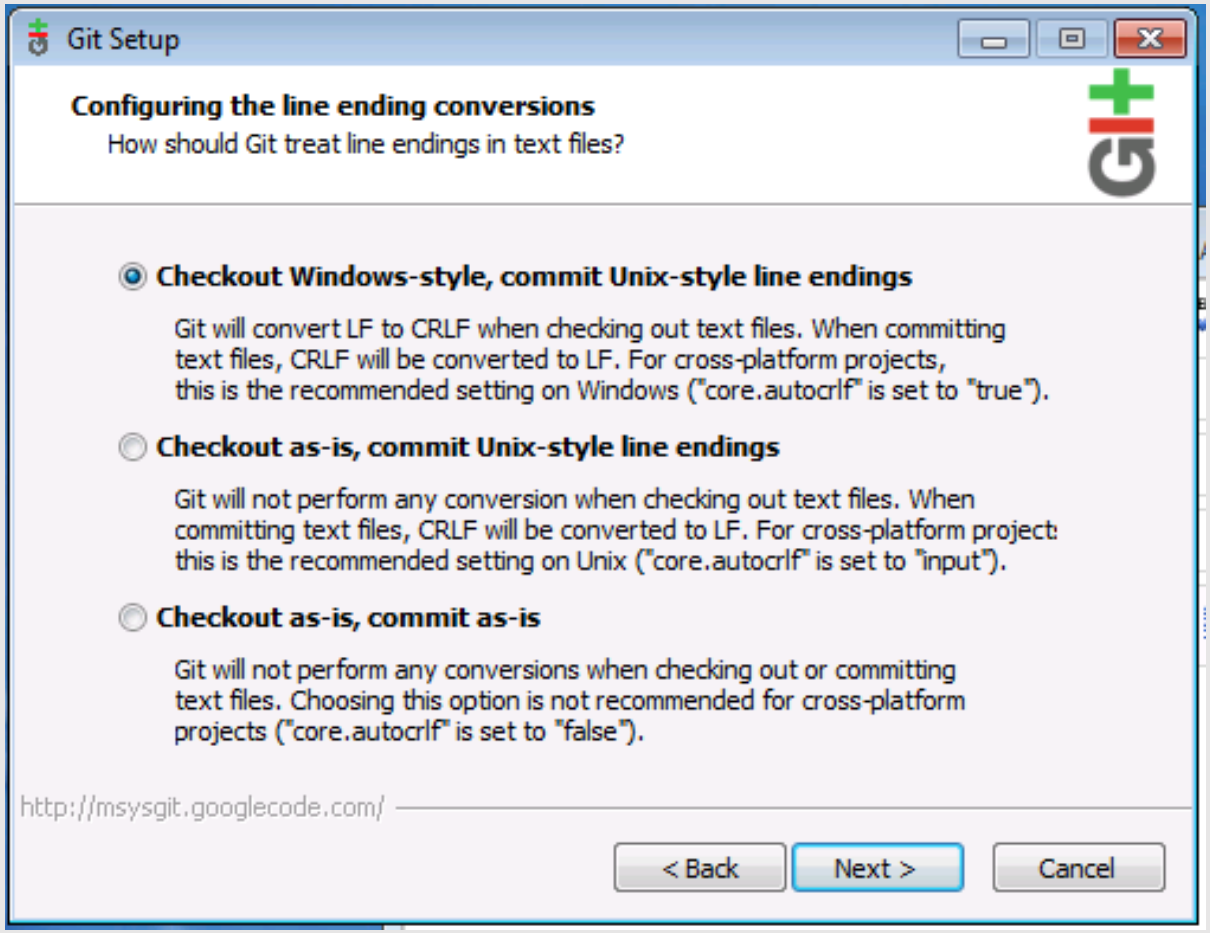
- git <command> --help
- git help <command>

Brings up html man page

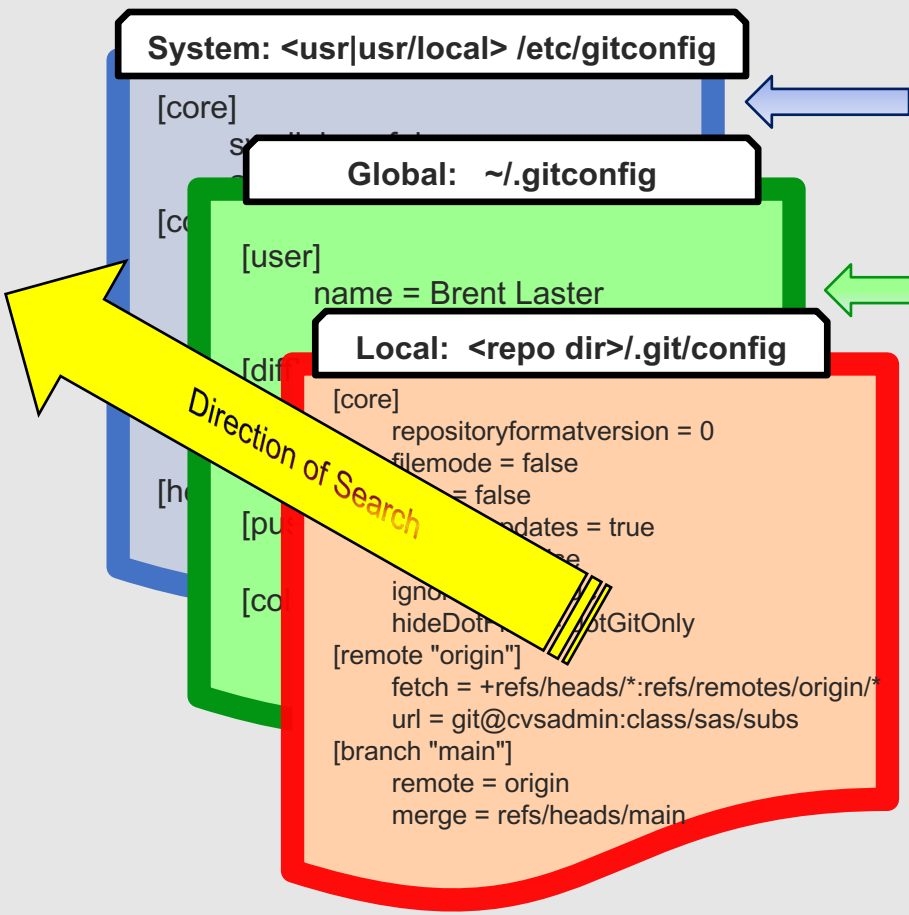


Line Endings

$ git config core.autocrlf



Git Configuration Files



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

git config --system

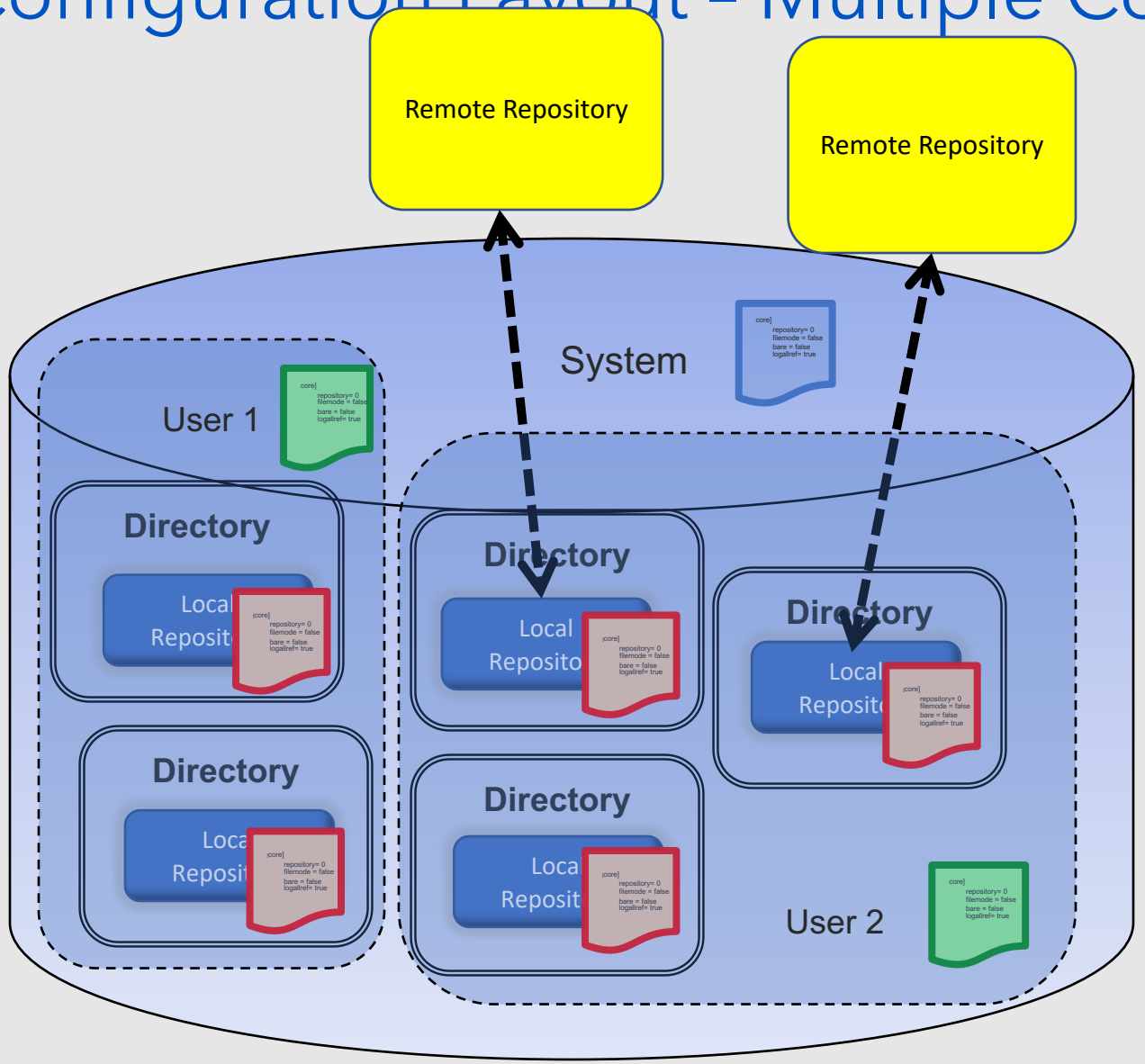
Global Configuration File
Scope: Current User

git config --global

Local Configuration File
Scope: Local repository

git config or git config --local

Git Configuration Layout – Multiple Configs



How Git reports status on files

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?

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

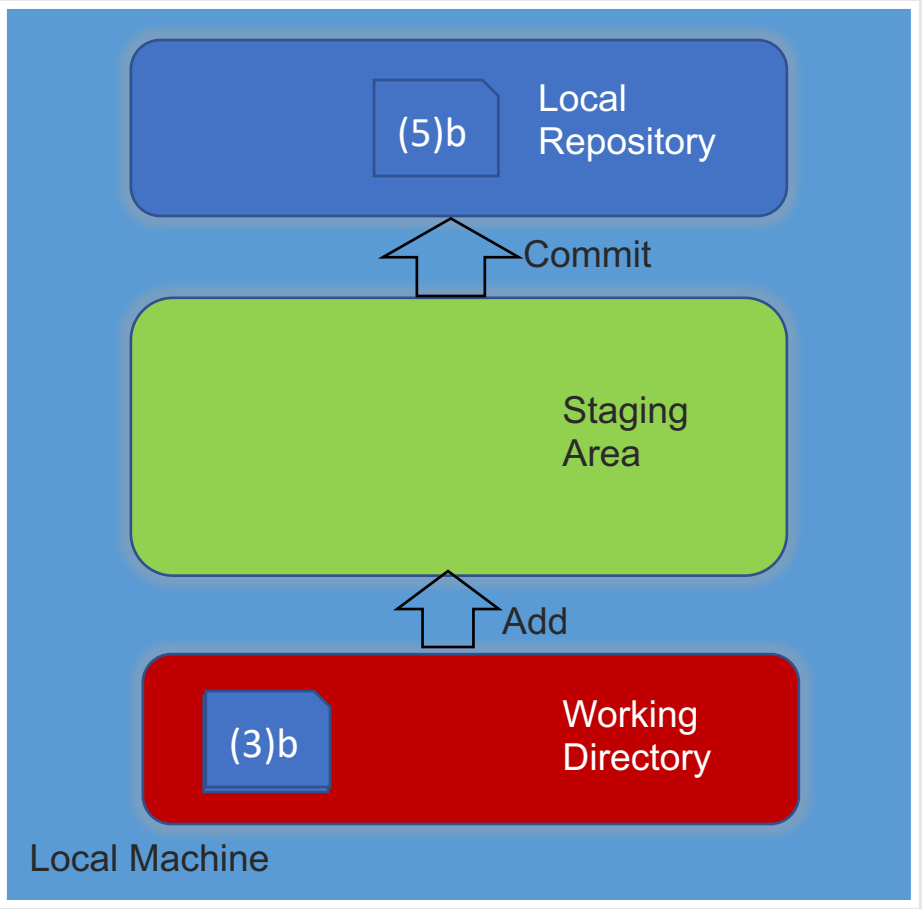
2. (Git) Add file.
Git aware: Yes – **tracked**
Anything staged? Rev a staged = Changes **to be committed**
Git compared to WD: Same - **unmodified**

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

Git Status

- Command: git status
  - Primary tool for showing which files are in which state
  - -s is short option - output <1 or more characters><filename>
    - *??* = untracked
    - *M* = modified
    - *A* = added
    - *D* = deleted
    - *R* = renamed
    - *C* = copied
    - *U* = updated but unmerged
  - -b option – always show branch and tracking info
- Common usage: git status -sb

Git Special References: Head and Index

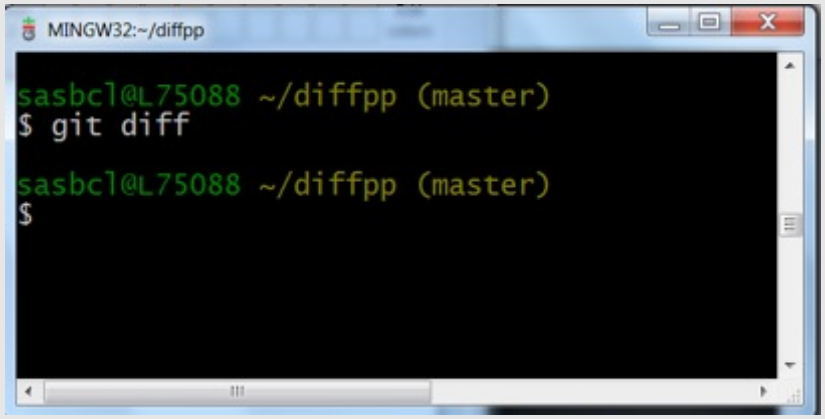
- Head
  - Snapshot of your last commit
  - Next parent (in chain of commits)
  - Pointer to current branch reference (reference = SHA1)
  - Think of HEAD as pointer to last commit on current branch
- Index (Staging Area)
  - Place where changes for the next commit get registered
  - Temporary staging area for what you're working on
  - Proposed next commit
- Cache - old name for index
- Think of cache, index, and staging area as all the same

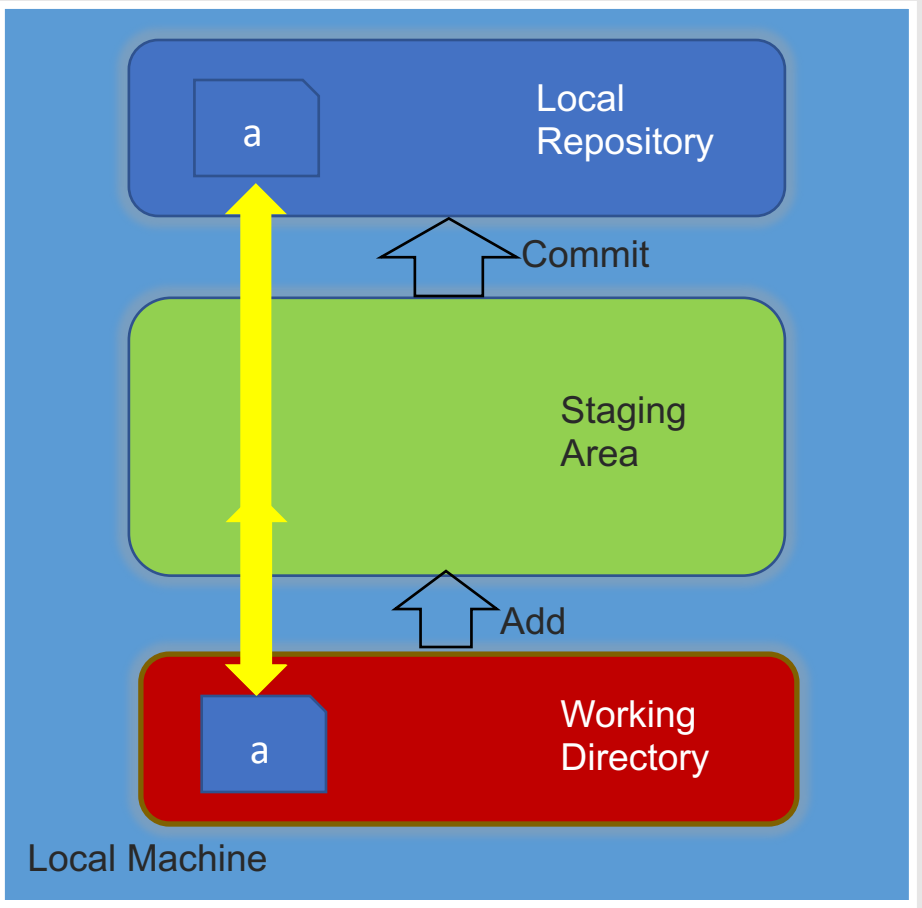
Showing Differences

- 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”

Understanding File Diffs 1

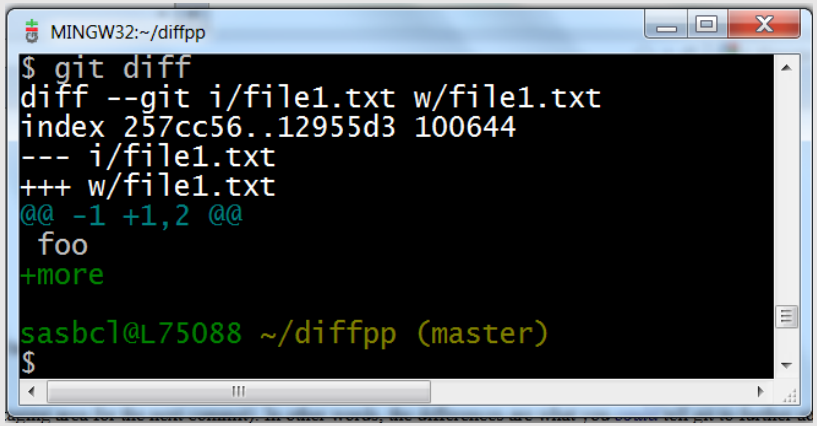
1.Assume file (rev a) created, added,and commited in local repo. WD version unchanged.
Git diff compares:
WD to SA (nothing there),
So compares to Local Repo
Results: Same
Output: Nothing

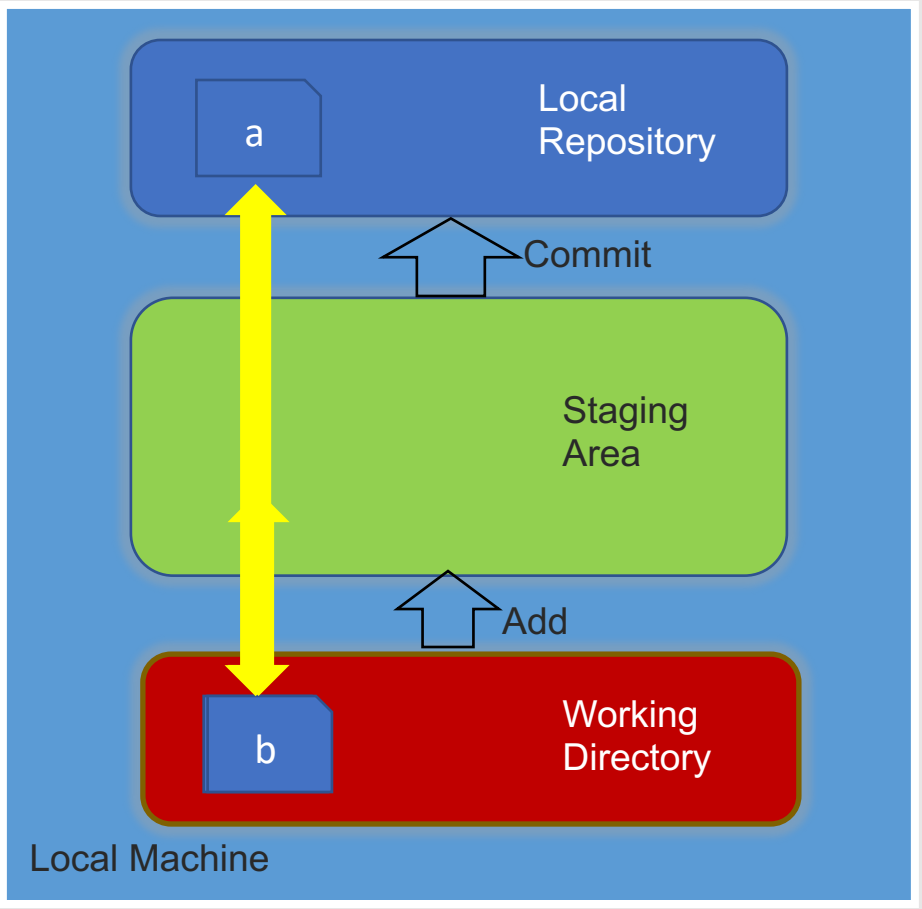




Understanding File Diffs 2

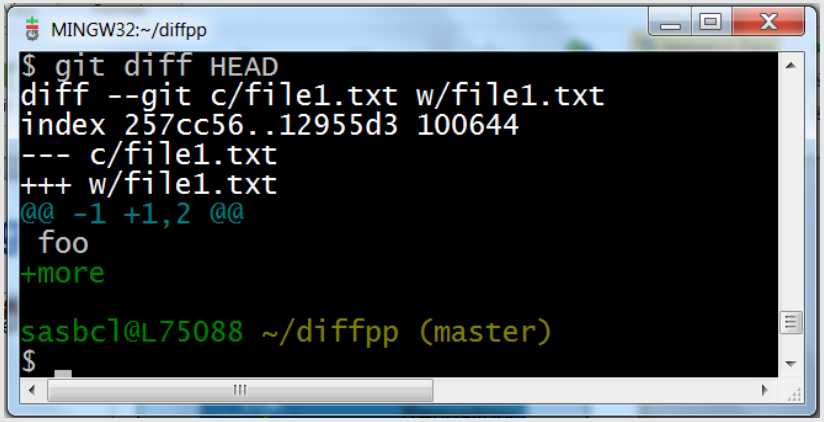
2. Now, we modify the file in the WD – rev b.
Git diff compares:
WD to SA (nothing there),
So compares to Local Repo
Results: Different
Output: Differences between WD and Local Repo

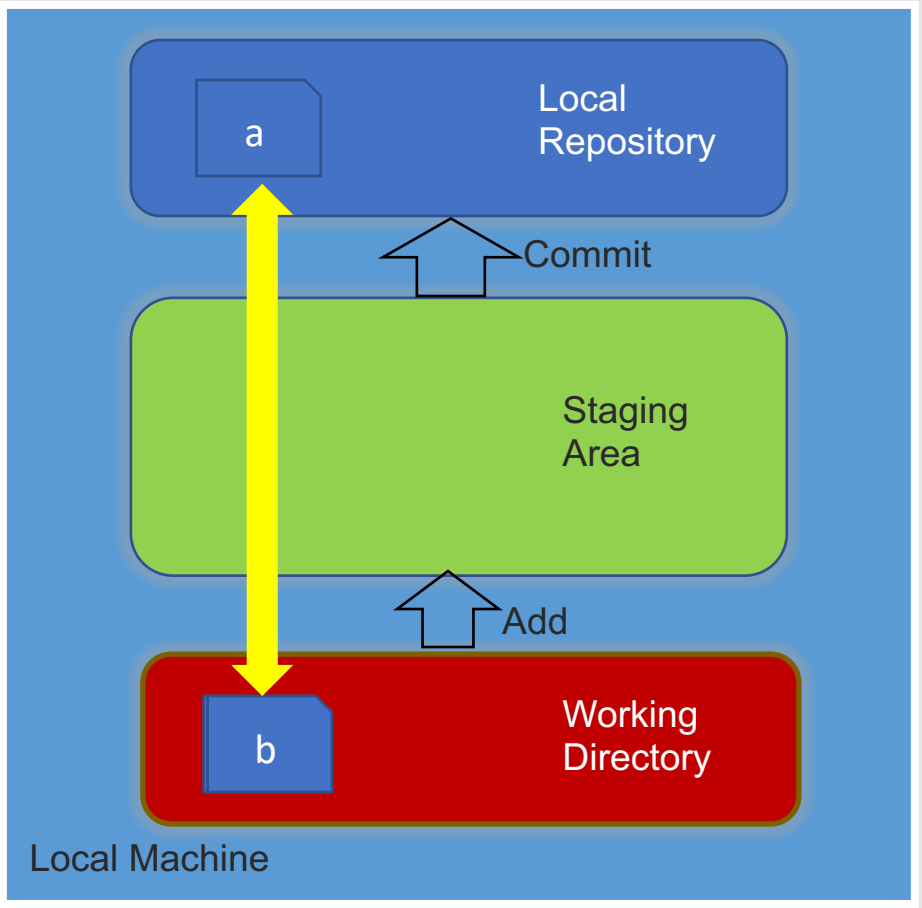




Understanding File Diffs 3

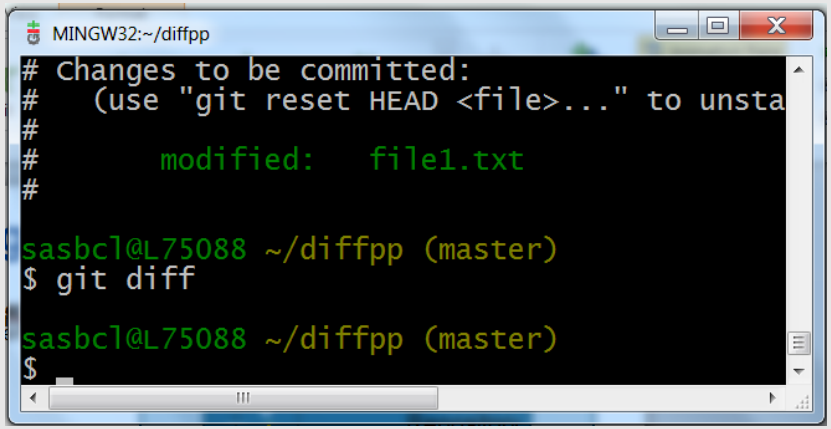
3. We can also compare explicitly against the Local Repo.
Git diff HEAD compares:
WD to Local Repo
Results: Different
Output: Differences between WD and Local Repo

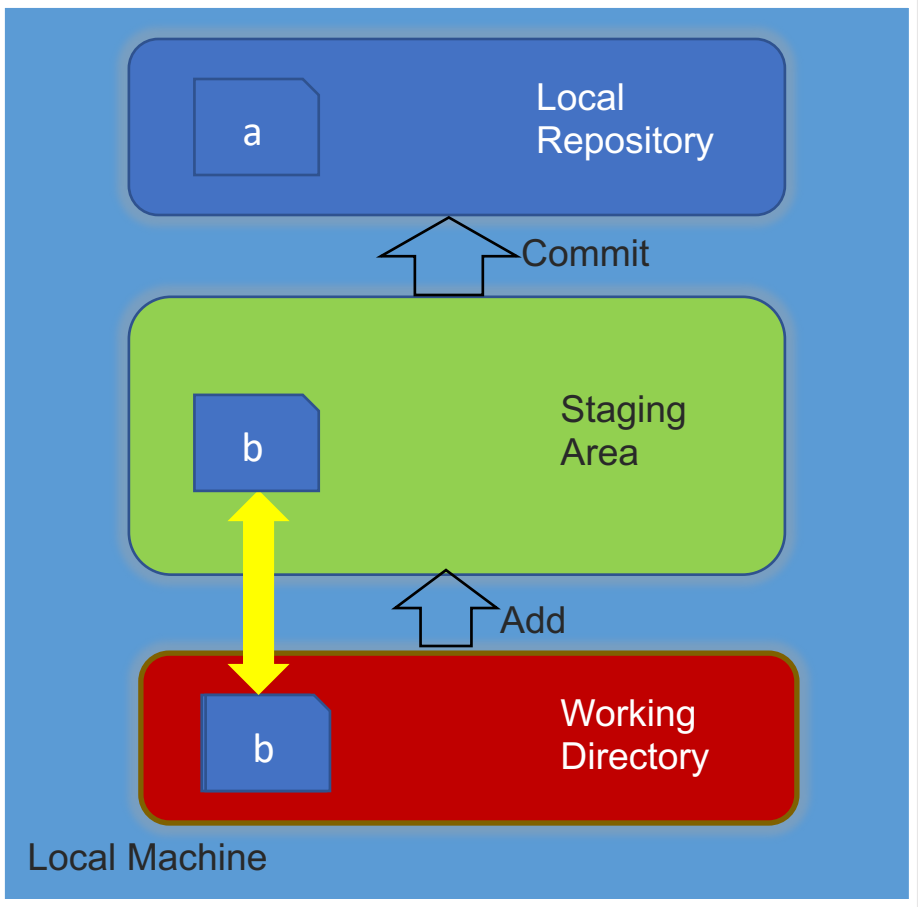




Understanding File Diffs 4

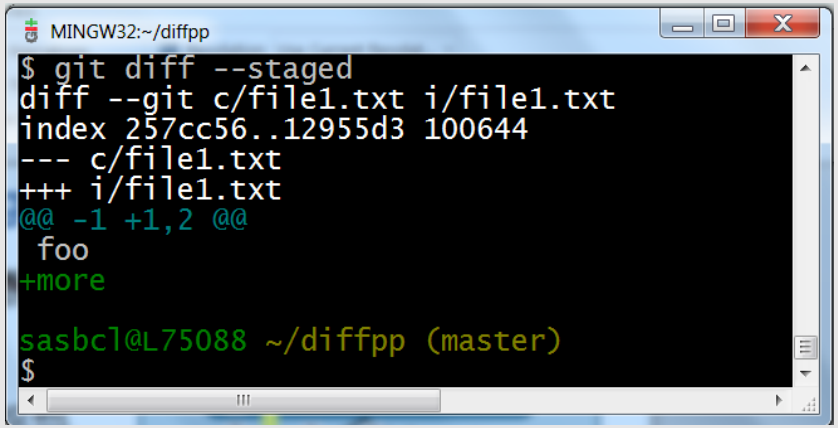
4. Let's add our change to the Staging Area (index).
Git diff compares:
WD to Staging Area
Results: Same
Output: Nothing

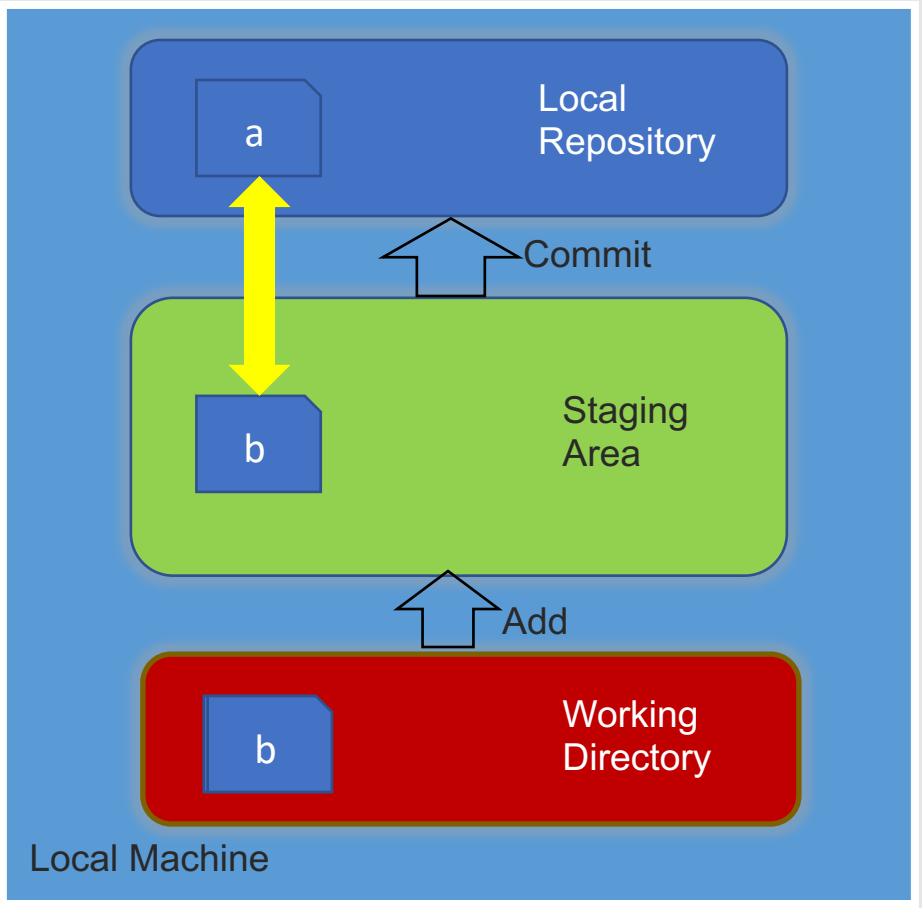




Understanding File Diffs 5

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





Lab 2 - Tracking Content through the File Status Lifecycle

Purpose: In this lab, we'll work through some simple examples of updating files in a Local Environment and viewing the status and differences between the various levels along the way.

History

- Command: git log
- With no options, shows name, sha1, email, commit message in reverse chronological order (newest first)
- -p option shows patch – or differences in each commit
  - Shortcut : git show
- -# -shows last # commits
- --stat – shows statistics on number of changes
- --pretty=oneline|short|full|fuller
- --format – allows you to specify your own output format
- --oneline – show only one line for each entry
- Time-limiting options --since|until|after|before (i.e. --since=2.weeks or --before=3.19.2011)
- gitk tool also has history visualizer

Sample git log format

- git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short
- In detail…
  - --pretty="..." defines the output format.
  - %h is the abbreviated hash of the commit
  - %d commit decorations (e.g. branch heads or tags)
  - %ad is the commit date
  - %s is the comment
  - %an is the name of the author
  - --graph tells git to display the commit tree in the form of an ASCII graph layout
  - --date=short keeps the date format short and nice

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

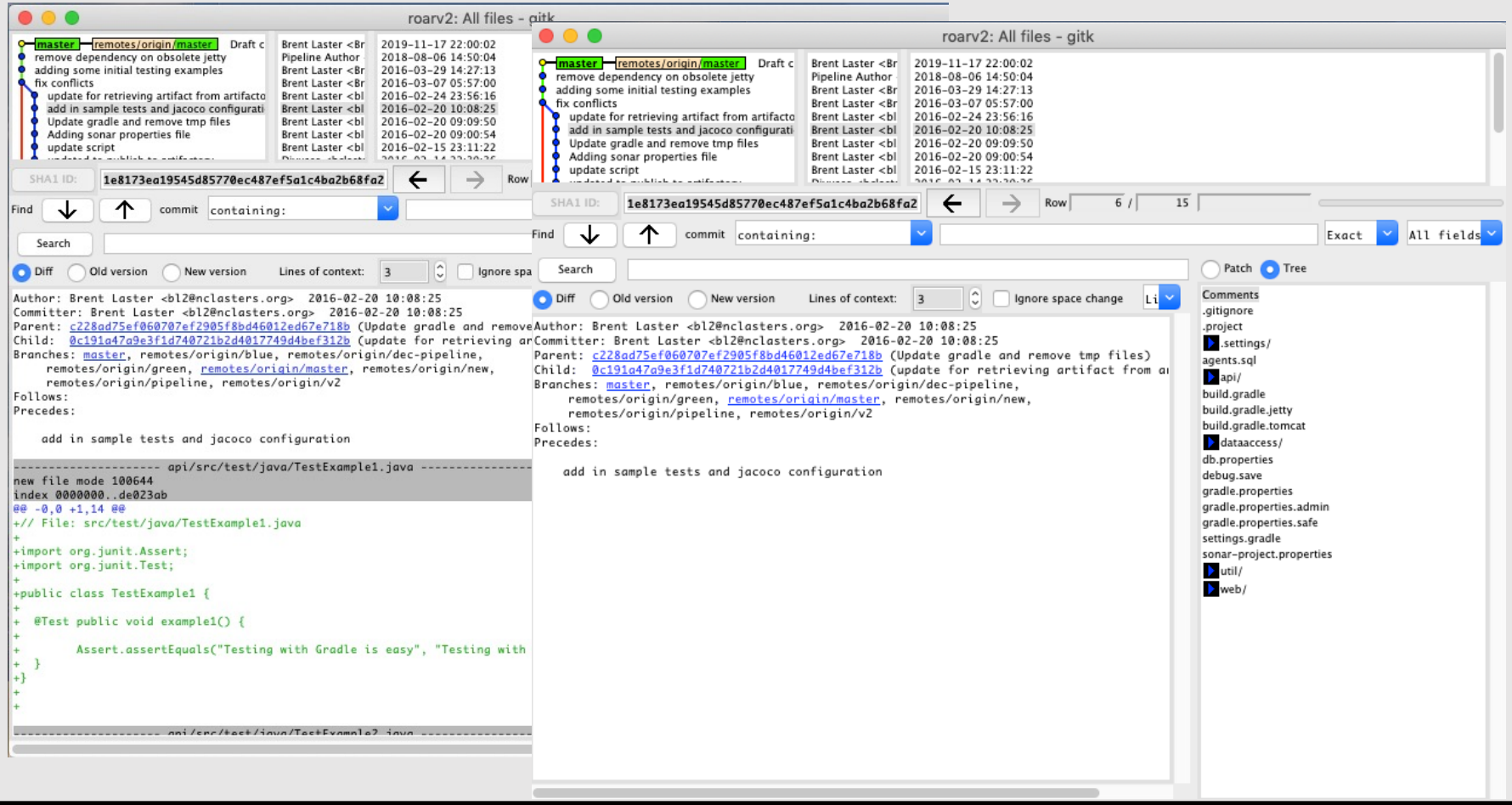
Git Aliases

- 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
```

Seeing History Visually

- gitk – graphical interface that comes with most git
- Shows history of local repository (not remote) in working directory
- Also useful to be able to see how changes look graphically

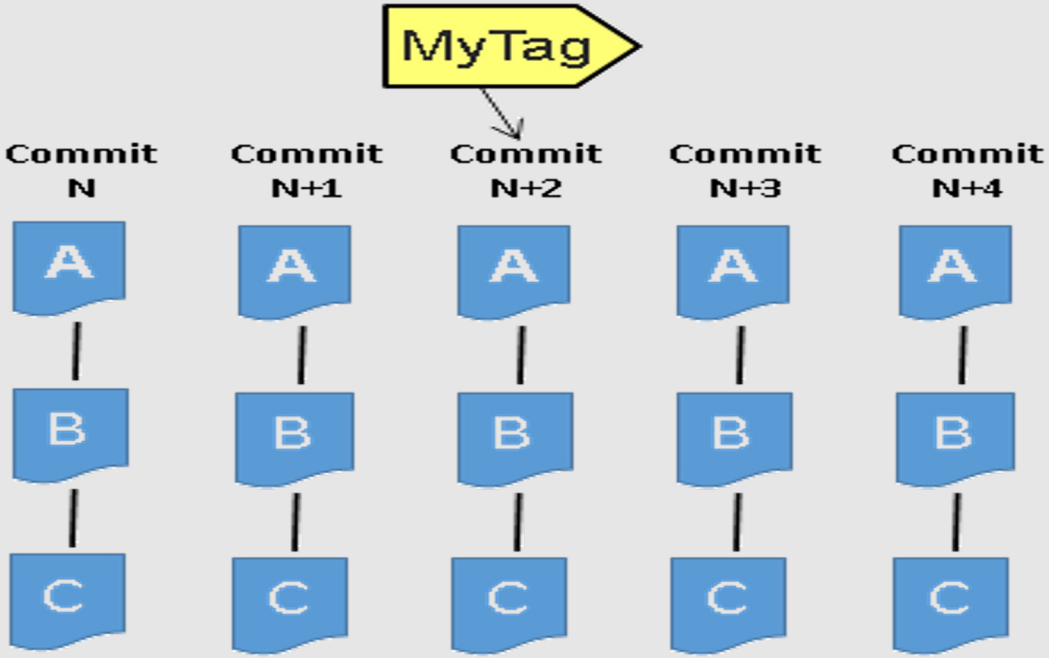


Tags in Git

- 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

Notes about Tags

- You tag commits (hashes), not files
- A tag is a reference (pointer) to a commit that stays with that commit
- To reference a file via a tag, you need to qualify with the file name (since the tag refers to entire commit)



Lab 3 - Working with Changes Over Time and Using Tags

Purpose: In this lab, we'll work through some simple examples of using the Git log commands to see the flexibility it offers as well as creating an alias to help simplify using it. We'll also look at how to tag commits to have another way to reference them.

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.

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

That's all - thanks!

techskillstransformations.com
getskillsnow.com