Version 1.5
09/17/23

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
- Labs docs are in https://github.com/skilldocs/git4
- Labs doc for workshop

https://github.com/skilldocs/git4/blob/main/git4-3-labs.pdf

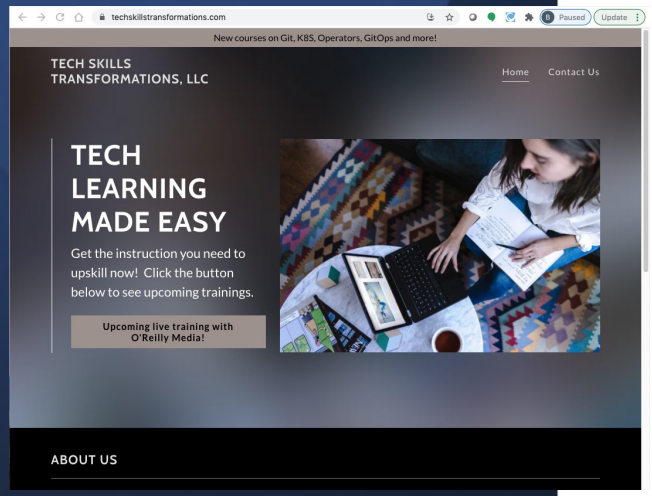
Git in 4 Weeks
Part 3

Presented by

Brent Laster

Tech Skills Transformations LLC

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

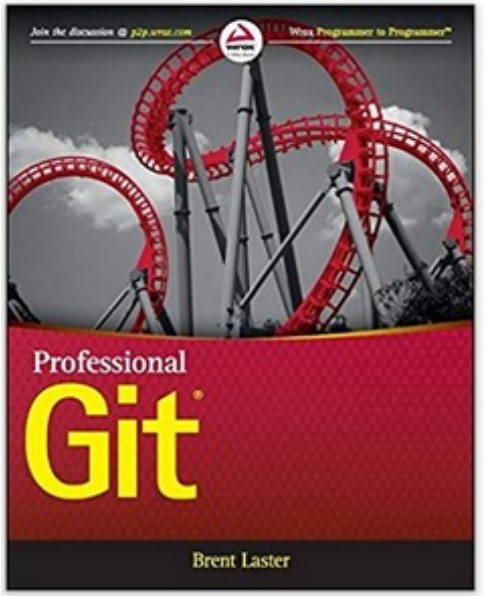
- Extensive Git reference, explanations,
- and examples
- First part for non-technical
- Beginner and advanced reference
- Hands-on labs

Professional Git 1st Edition

by Brent Laster (Author)

7 customer reviews

Look inside



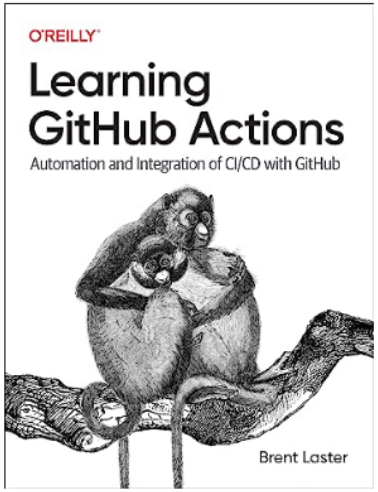
GitHub Actions book

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:

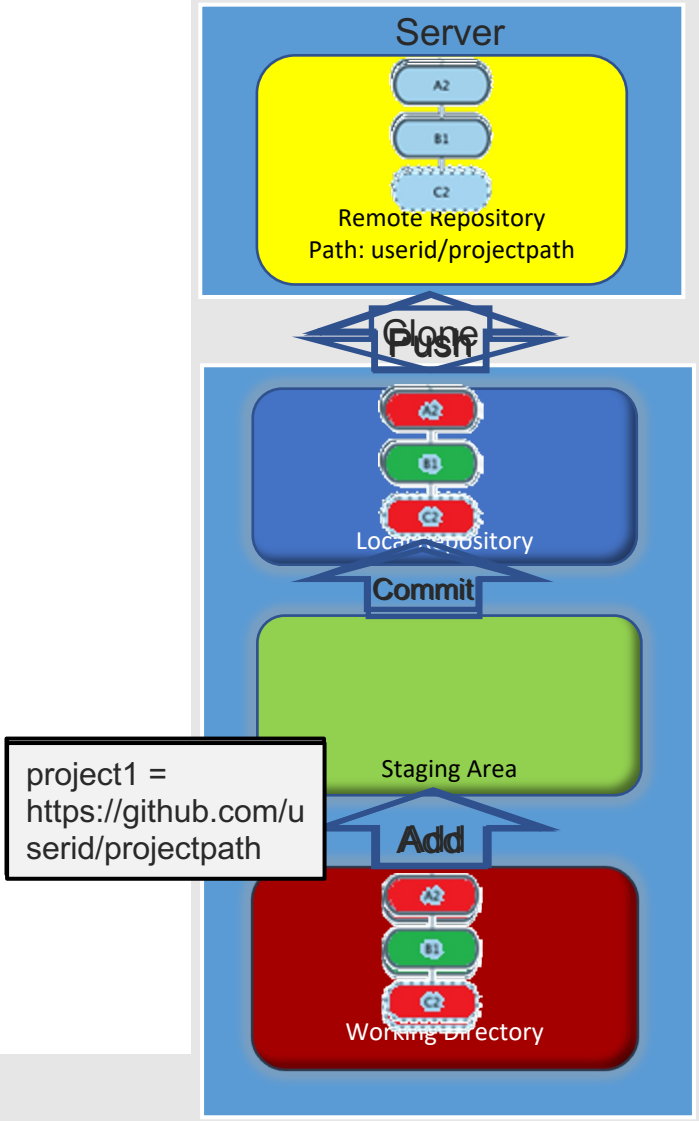
- Learn what GitHub Actions are, the various use cases for them, and how to incorporate them into your processes
- Understand GitHub Actions' structure, syntax, and semantics
- Automate processes and implement functionality
- Create your own custom actions with Docker, JavaScript, or shell approaches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

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)
```



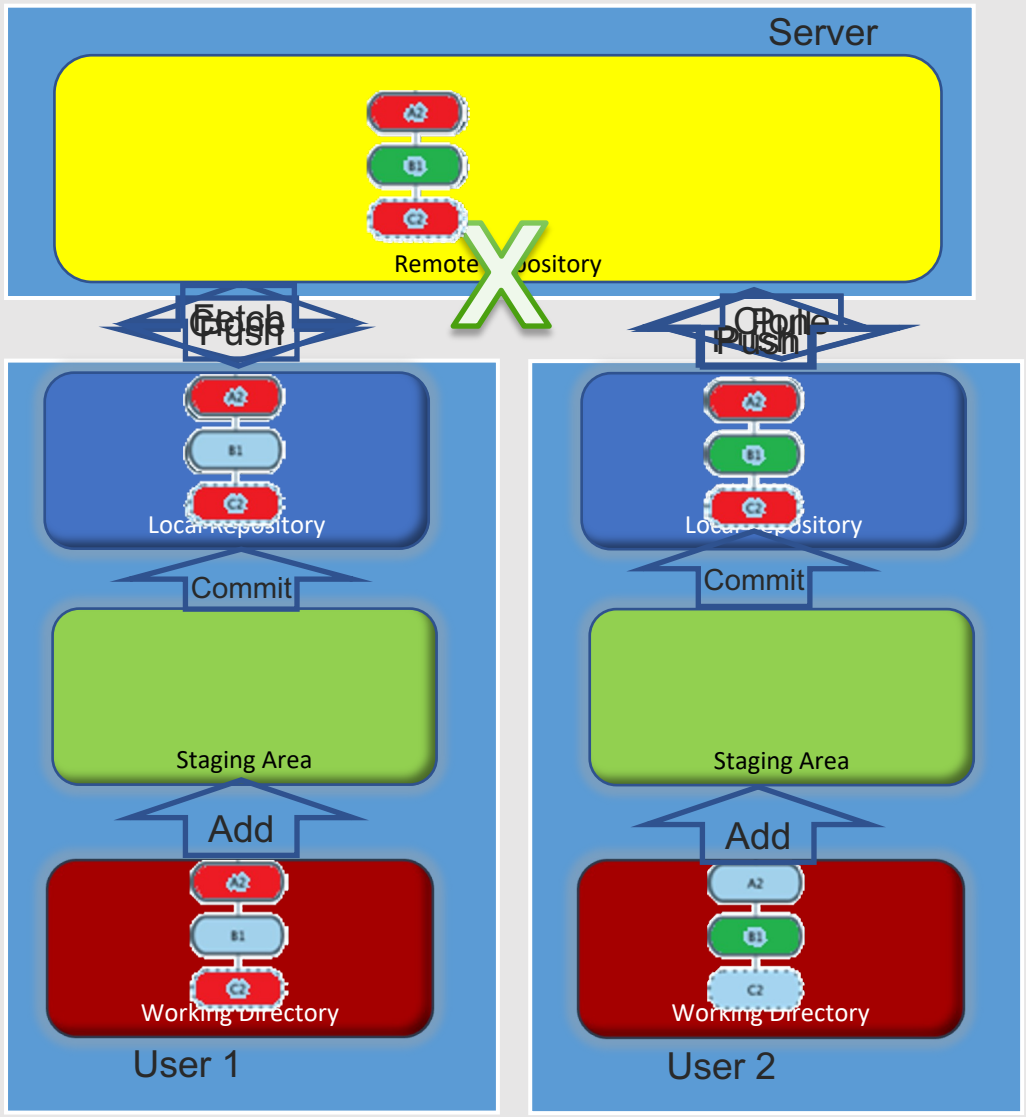
Working with a Remote and Multiple Users

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
```



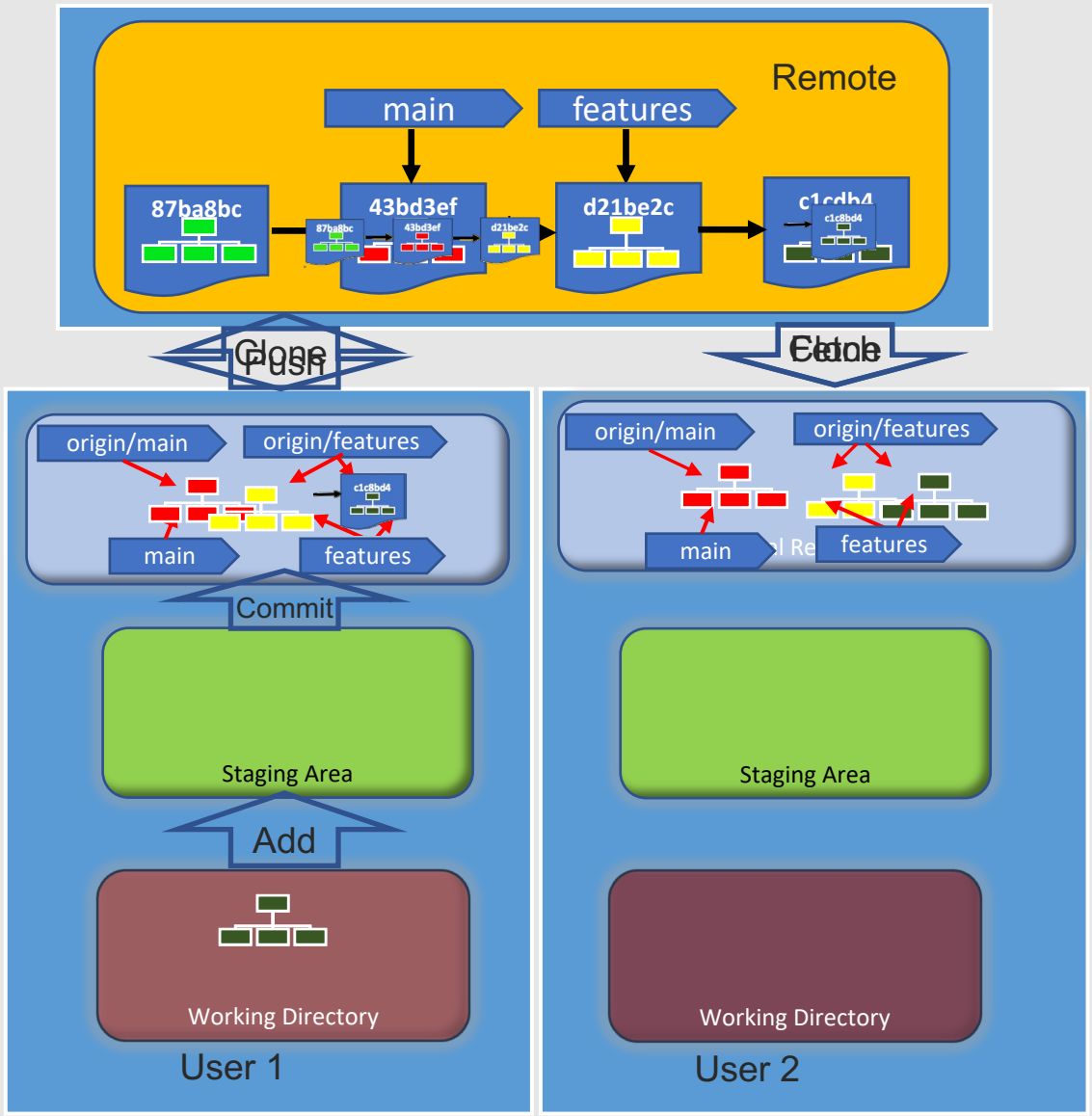
Remote vs Local Branches

User 1

$ git clone …
$ git checkout features
$ git status
On branch features
Your branch is up to date with 'origin/feature'.
<edit file(s)>
$ git commit -am "update"
$ git status
On branch feature
Your branch is ahead of 'origin/feature' by 1 commit.
(use "git push" to publish your local commits)
$ git push

User 2

$ git clone …
$ git checkout features
$ git fetch
$ git status
Your branch is behind 'origin/features' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)
$ git pull OR $ git merge origin/features



command: switch

- Purpose: switch (change) branches
- Use case: more explicit than checkout
- Syntax:

  (-c| -C creates new branch)

```
git switch [<options>] [--no-guess] <branch>
git switch [<options>] --detach [<start-point>]
git switch [<options>] (-c|-C) <new-branch> [<start-point>]
git switch [<options>] --orphan <new-branch>
```

New Content

command: bisect

- Purpose - Use “automated” binary search through Git’s history to find a specific commit that first introduced a problem (i.e. “first bad commit”)
- Use case - Quickly locate the commit in Git’s history that introduced a bug
- Syntax:

```
git bisect start [--term-{old,good}=<term> --term-{new,bad}=<term>]
          [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new|<term-new>) [<rev>]
git bisect (good|old|<term-old>) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect (visualize|view)
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
git bisect help
```

bisect usage

- Use “automated” binary search to find change that first introduced a bug (i.e. “first bad commit”)
- Initiate with git bisect start
- Can pass range to start option to identify bad and good revisions – i.e.

  git bisect start HEAD HEAD~10
- Identify first “bad” commit, usually current one via git bisect bad
- Identify a “good” (functioning) commit via git bisect good
- From there, git will do a binary search and pick a “middle” commit to checkout.
- Try the checked out version and indicate git bisect good or git bisect bad depending on whether it works or not.
- Process repeats until git can identify “first bad” commit/revision.
- Can grab previous good revision by specifying “first bad”^
- Can update, create new branch, rebase, etc. to isolate good revisions.
- Other useful options to git bisect: log, visualize, reset, skip, run

bisect example

FIRST BAD COMMIT

- checkout latest version
- try code
- git bisect start
- git bisect bad
- checkout earlier version (user checks out)
- try code
- git bisect good (bisect checks out version 5)
- try code
- git bisect good (bisect checks out version 7)
- try code
- git bisect bad (bisect checks out version 6)
- try code
- git bisect bad (git reports version 6 as the first bad commit) WORKING DIRECTORY

bisect usage - run

- Can bisect automatically IF you have a script that can tell if current source is good or bad
- Script requirements:
  - Must exit with 0 if current source is good
  - Exit with 1 - 127 (except 125) if source is bad
  - Exit with 125 if source cannot be validated - is skipped
- Example:

  $ git bisect run <script> <arguments>

Lab 7 - Using the Git bisect command

Purpose: In this lab, we'll learn how to quickly identify the commit that introduced a problem using Git bisect.

Porcelain and Plumbing commands

Porcelain

- add
- am
- archive
- bisect
- blame
- branch
- bundle
- checkout
- cherry-pick
- clean
- commit
- config
- describe
- diff
- fetch
- format-patch
- gc
- grep
- gui
- help
- init
- instaweb
- log
- merge
- mv
- notes
- prune
- pull
- push
- rebase
- reflog
- remote
- rerere
- revert
- rm
- show
- stash
- submodule
- switch
- tag
- worktree

Plumbing

- cat-file
- cherry-pick
- commit-tree
- count-objects
- fast-export
- fast-import
- filter-branch
- fsck
- hash-object
- ls-files
- ls-trees
- merge-tree
- pack-refs
- read-tree
- rev-parse
- symbolic-ref
- update-index
- verify-commit
- verify-tag
- write-tree

command: filter-branch

- Purpose - provides a way to revise git revisions (rewrite history); different types of operations can be done by applying different built-in filters against revisions
- Use case – remove files from commits, change email, split areas out, etc.
- Syntax:

git filter-branch [--setup <command>] [--subdirectory-filter <directory>] [--env-filter <command>] [--tree-filter <command>] [--index-filter <command>] [--parent-filter <command>] [--msg-filter <command>] [--commit-filter <command>] [--tag-name-filter <command>] [--prune-empty] [--original <namespace>] [-d <directory>] [-f | --force] [--state-branch <branch>] [--] [<rev-list options>…]

- Notes:
  - Usual prohibitions/cautions against changing things already in the repository apply
  - Uses git rev-list to help identify commits in some cases
  - filter-repo command is preferred now but not included in base Git

rev-list

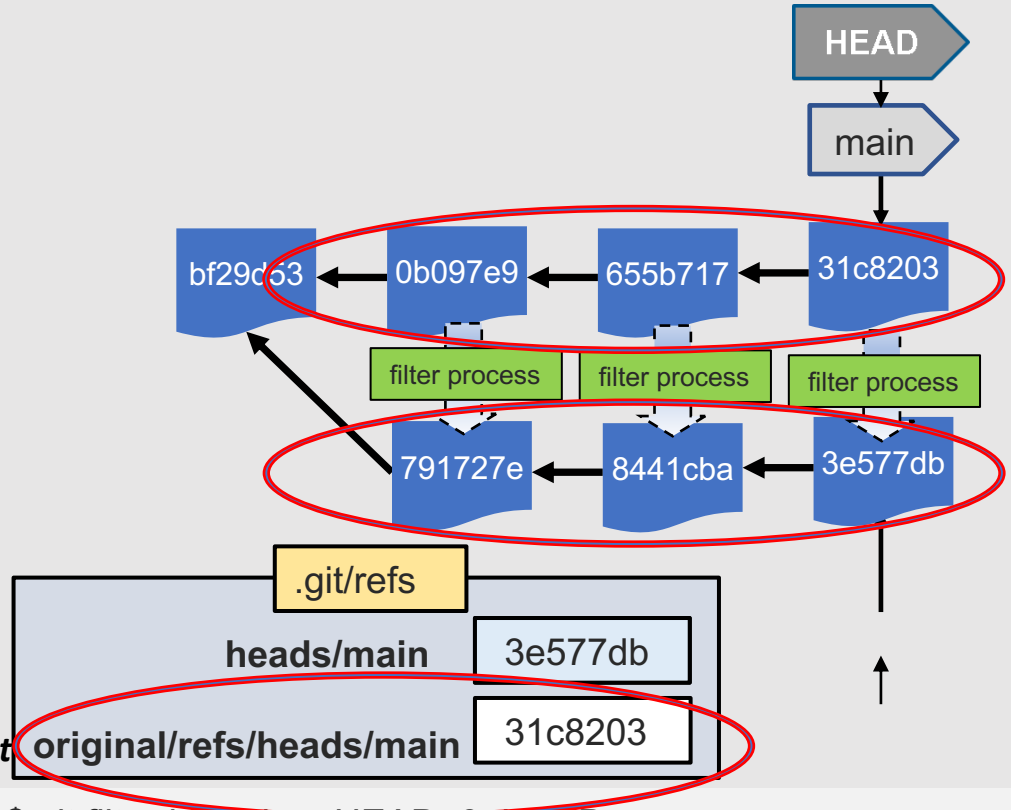
- Another plumbing command
- Used to support some other commands such as filter-branch
- main use is to list a set of commits bounded by some range or criteria; use the rev-list help page to find out more details on specifying ranges if you're interested

git rev-list **[--max-count=<number>]**
[--skip=<number>]
[--max-age=<timestamp>]
. . . (lots of other omitted options)
[--count]
[--use-bitmap-index]
<commit>… [-- <paths>…

- For filter-branch, rev-list options can be used to bound the set of things for filter-branch to operate against.
- In most cases, if you need to supply an option for rev-list, you can just use the --all option to indicate all commits.

working with filter-branch

- After a filter-branch run, the HEAD points at the rewritten revisions
- However,the commits that make up the previous structure are still there—just not accessible in this chain
- Git keeps a backup of the previous reference pointed to by HEAD before the filter-branch.
- Assuming you were working in the main branch, that SHA1 value can be found in the following path:.**git/refs/original/refs/heads/main**
- Can undo the filter-branch by just using a reset to a point before the operation
- To undo filter-branch run the command, ***git reset --hard <sha1 from path above> or ref-path***
- Can also use the *git reflog* command to find the relative reference before the operation and then run the command, ***git reset --hard HEAD@{<relative offset>}.***




```
$ git filter-branch ... HEAD~3..HEAD
Rewrite 0b097e958005a98b08e835cc2fdfeecc1b276cf5 (1/3) ...
Rewrite 655b7172eb4da1272dafe5a21012dcabf794e9a5 (2/3) ...
Rewrite 31c820396cb25c08877a00f9cc0729bde7e7eb71 (3/3) ...

Ref 'refs/heads/main' was rewritten

$ git reset refs/original/refs/heads/main
```

command: reflog

- Logs when tips of branches or other references are updated in the local repo

git reflog [*show*] [log-options] [<ref>]
git reflog expire [--expire=<time>] [--expire-unreachable=<time>] [--rewrite] [--updateref] [--stale-fix] [--dry-run | -n] [--verbose] [--all [--single-worktree] | <refs>…]
git reflog delete [--rewrite] [--updateref] [--dry-run | -n] [--verbose] ref@{specifier}…
git reflog exists <ref>

- Previous references can be specified for various Git ops - such as reset
  - HEAD@{3} = where HEAD used to point three moves ago
  - main@{one.week.ago} - where main branch was one week ago in the local repo
- Default subcommand is "show" and can take git log options

```
$ git reflog
e105c1b HEAD@{0}: checkout: moving from main
to docs
44e709e HEAD@{1}: commit: Update Git
Fundamentals labs
7517636 HEAD@{2}: checkout: moving from docs
to main
e105c1b HEAD@{3}: commit: update Git
Fundamentals Labs
```

```
$ git reflog --date=iso
e105c1b HEAD@{2021-03-23 13:41:55 -0400}:
checkout: moving from main to docs
44e709e HEAD@{2021-03-18 19:00:41 -0400}:
commit: Update Git Fundamentals labs
7517636 HEAD@{2021-03-18 19:00:12 -0400}:
checkout: moving from docs to main
e105c1b HEAD@{2021-03-18 18:58:45 -0400}:
commit: update Git Fundamentals Labs
```

```
$ git reflog HEAD@{one.week.ago}
44e709e HEAD@{Thu Mar 18 19:00:41 2021 -
0400}: commit: Update Git Fundamentals labs
7517636 HEAD@{Thu Mar 18 19:00:12 2021 -
0400}: checkout: moving from docs to main
e105c1b HEAD@{Thu Mar 18 18:58:45 2021 -
0400}: commit: update Git Fundamentals Labs
ace0698 HEAD@{Wed Mar 17 22:33:22 2021 -
0400}: checkout: moving from main to docs
```

```
$ git reflog  --pretty=format:"%h %cd %cn %s"
e105c1b Thu Mar 18 18:58:45 2021 -0400 Brent
Laster update Git Fundamentals Labs
44e709e Thu Mar 18 19:00:41 2021 -0400 Brent
Laster Update Git Fundamentals labs
7517636 Mon Mar 15 23:20:48 2021 -0400 Brent
Laster Update bdpj2 class labs
e105c1b Thu Mar 18 18:58:45 2021 -0400 Brent
Laster update Git Fundamentals Labs
```

filter-branch - subdirectory filter

- Filter-branch expects a subdirectory rather than a command
- Use case: split out subdirectory into repository
- Example:

```
$ git filter-branch -f --subdirectory-filter <subdir> -- --all
Rewrite be42303ffb9356b8e27804ce3762afdeea624c64 (1/6) (0 seconds passed,
remainRewrite c6b5cbd8805bc7b1b411a89be66adccc037df553 (2/6) (1 seconds passed,
remainRewrite f64bd2ce520e9a3df4259152a139d259a763bc31 (2/6) (1 seconds passed,
remainRewrite d8a1065e9709d2c5ee20f62fd4e338fe35666c65 (2/6) (1 seconds passed,
remainRewrite c228ad75ef060707ef2905f8bd46012ed67e718b (5/6) (5 seconds passed,
remainRewrite 4758f9e7a9cbeb8dfea0a37a416b46d823ffa95a (5/6) (5 seconds passed,
remaining 1 predicted)
Ref 'refs/heads/master' was rewritten
```

- Notice that it is going through and processing the set of commits here that have to do with the particular <subdir>
- Ending status of ‘refs/heads/<branch>’ is indication of successful completion

filter-branch - deleting a file from history

- Use case: remove undesired file from revisions in a branch
- Either tree-filter or index-filter can be used
- Because index-filter can use the index instead of the working tree, this option will be faster
- Think removing in the staging area versus checking out and removing
- index-filter option takes a command as an argument to apply when it encounters an instance of the thing you are trying to filter out
- Can use the git rm command here to simplify the task
- Add the --cached option because working in the index
- Add the --ignore-unmatched option to the git rm command
  - Tells the command to exit with a zero return code even if none of the files match
- Command to remove file :

```
$ git filter-branch --index-filter 'git rm --cached --ignore-unmatch <relative
path to file>' <branch name>
```

- Need to run command for each branch (add -f to filter-branch to overwrite backup reference for previous branch)

filter-branch - environment filter

- --env-filter <command>
- Uses environment filter
  - Allows environment variables to be set and used by commands passed to filter-branch
- Use case: changing email address for set of commits
- Set GIT_AUTHOR_EMAIL environment variable
- Must export it to ensure it is set for operation
- Example:

Env Variable Name	Meaning
GIT_AUTHOR_NAME	“author” field name
GIT_AUTHOR_EMAIL	“author” field email
GIT_AUTHOR_DATE	”author” field timestamp
GIT_COMMITTER_NAME	“committer” field name
GIT_COMMITTER_EMAIL	“committer” field email
GIT_COMMITTER_DATE	“committer” field timestamp

```
$ git filter-branch -f --env-filter 'GIT_AUTHOR_NAME="your-name-here",
GIT_AUTHOR_EMAIL=your-email-here; export GIT_AUTHOR_NAME; export
GIT_AUTHOR_EMAIL' HEAD~6..HEAD
```

command: filter-repo

- separate tool for rewriting git history
  - python script from github.com/newren/git-filter-repo
  - needs to be installed separately

  git filter-repo --analyze *git filter-repo* [<path_filtering_options>] [<content_filtering_options>] [<ref_renaming_options>] [<commit_message_filtering_options>] [<name_or_email_filtering_options>] [<parent_rewriting_options>] [<generic_callback_options>] [<miscellaneous_options>]
- replacement for git filter-branch
  - challenges with filter-branch
    - » can be extremely slow for all but simple repos
    - » can be challenging to use
    - » issues cannot be fixed in a way that is backwards-compatible
    - » git community recommends to stop using filter-branch
- cheat sheet for conversions at https://github.com/newren/git-filter-repo/blob/main/Documentation/converting-from-filter-branch.md#cheat-sheet-conversion-of-examples-from-the-filter-branch-manpage

Lab 8 - Working with filter-branch

Purpose: In this lab, we'll see how to work with the filter branch functionality in Git

Specifying Previous Revisions Relatively

- ^ sign has multiple uses in Git
- A ^ sign following a reference means “one before” – example: HEAD^ means the commit before last (termed the “caret parent”)
- Can be extended with more ^ signs – example: HEAD^^^ means 3 commits back (before current)
- Shortcut notation uses ~# - example: HEAD~3 = HEAD^^^
- A ^ sign before a ref means “not in this” or “not before”
  - Usually used as part of a range
  - Example – to see all changes in branch1 that are not in branch 2
    - » git log branch1 ^branch2
  - Can be combined with caret parent notation
    - » git log branch1 ^branch2~2 (show everything that is in branch1 but not in branch2 before 2 commits back)
    - » If branch1 and branch2 are the same, this would just show the last 2 commits in the branch
    - » git log branch1 ^branch1 – What do you think this shows?

(Interactive) Rebase

- Purpose - allows you to run operations against commits in the git history
- Use case - you need to make some kind of modification to one or more commits in the repository (rewrite history)
- Syntax

git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
    [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
    --root [<branch>]

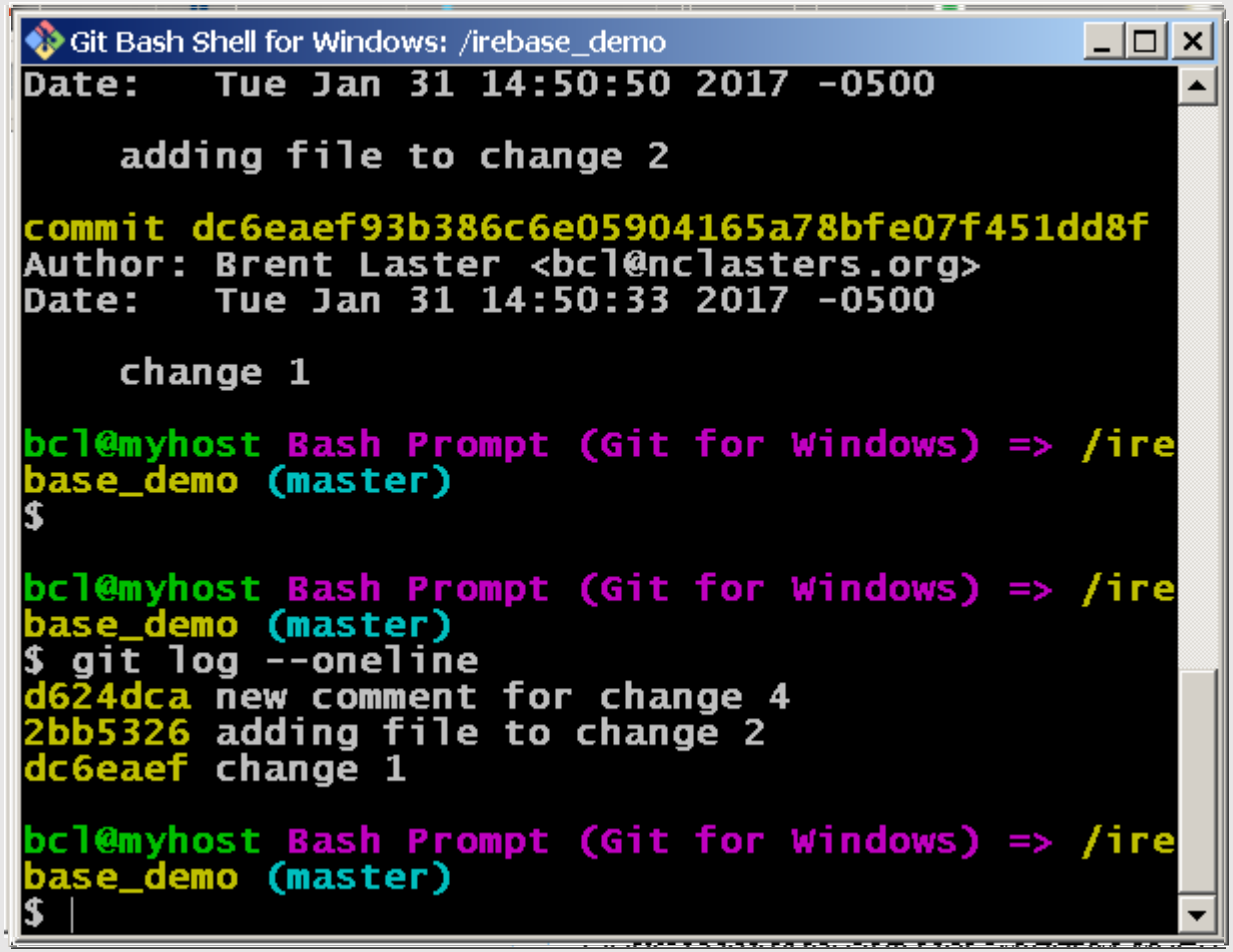
- Notes - creates an interactive script/batch file to modify commits
- Cautions - don’t use this on anything already pushed

Interactive Rebase Commands

```
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
```


Interactive Rebase example

- Choose set of commits
- Initiate interactive rebase
- Git presents initial script
- Modify commands (save and exit)
- Act on prompts
- Commits are updated



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)

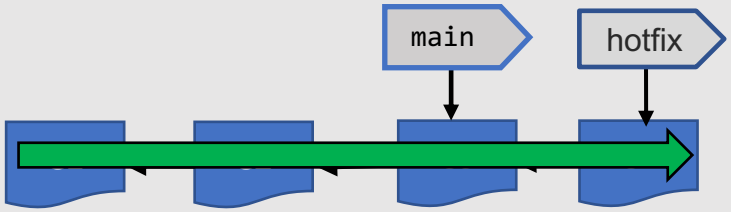
Lab 9 - Working with interactive rebase

Purpose: In this lab, we'll see how to use interactive rebase to squash multiple commits into one

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 (-)
```

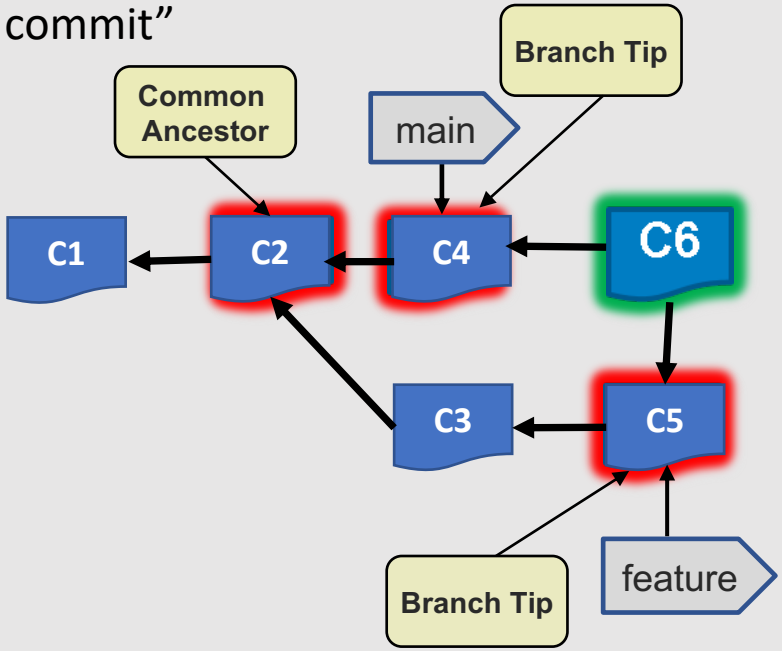


About “Fast Forward” – because commit pointed to by branch merged was directly “upstream” of the current commit, Git moves the pointer forward

(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"

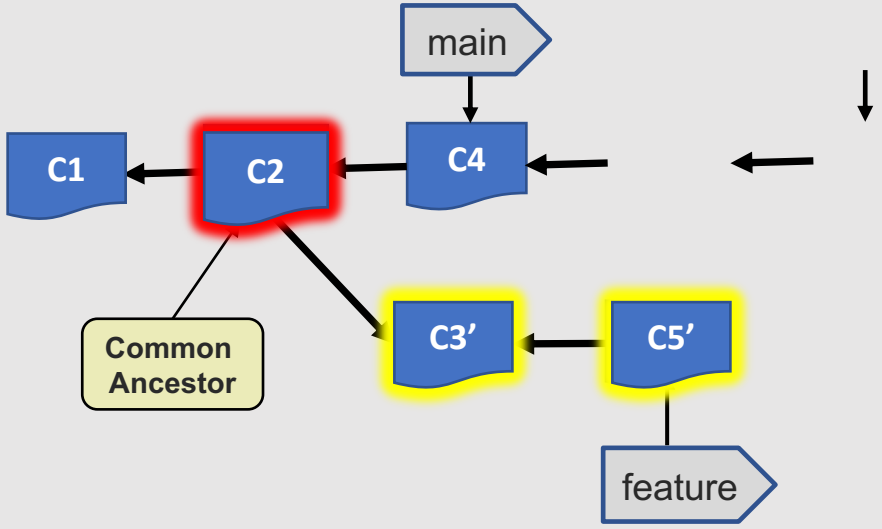



```
$ git checkout main
$ git merge feature
```

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
```



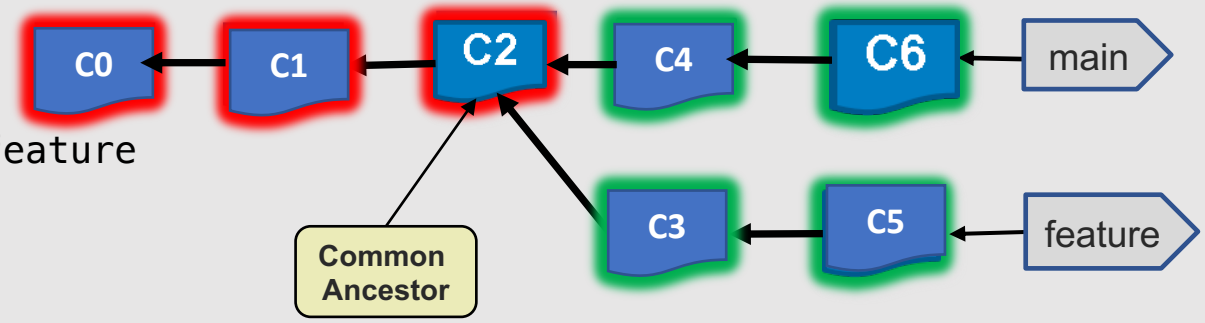
Understanding Ranges

- Specifying a single commit gets the set of commits reachable from the given commit
  - reachable - commit itself and commits in its ancestry chain
- Exclude commits via caret (^)
  - ^R1 R2 - commits reachable from R2 but exclude ones reachable from R1 (stop at R1)
- Dotted Range Notations
  - .. - same as caret notation above (^R1 R2 = R1..R2)
  - ... - symmetric difference - R1...R2 - set of commits reachable from either R1 or R2 but not both (changes in their chains since they last were in common)
  - omitting either side defaults to HEAD
    - » HEAD..origin = what changes made on HEAD since forked

```
$ git log --oneline ^main feature
dc666ad (feature) C5
8c175fa C3
```

```
$ git log --oneline main..feature
dc666ad (feature) C5
8c175fa C3
```

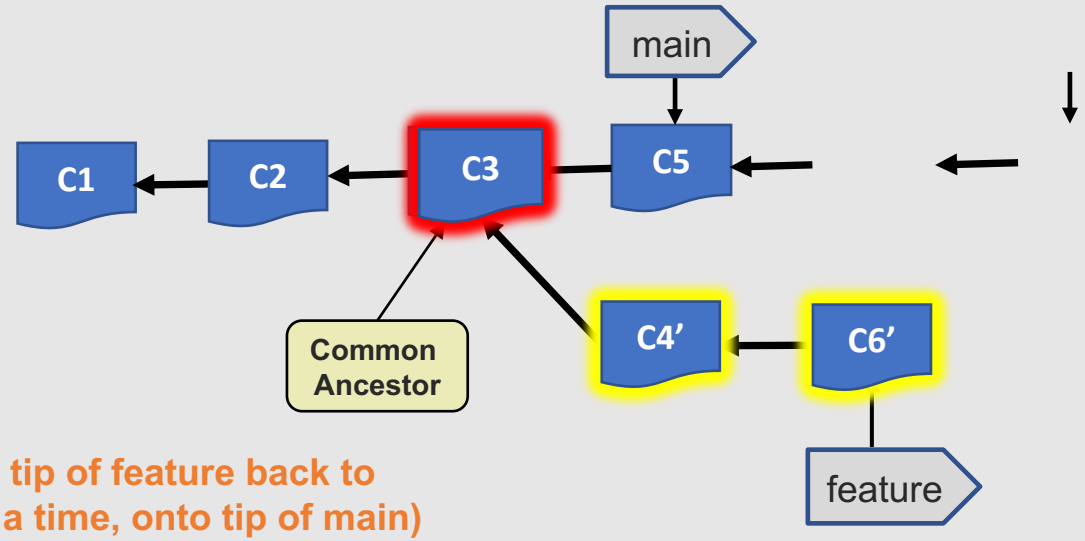
```
$ git log --oneline main...feature
cf8db03 (HEAD -> main) C6
dc666ad (feature) C5
5586921 C4
8c175fa C3
```



command: cherry-pick

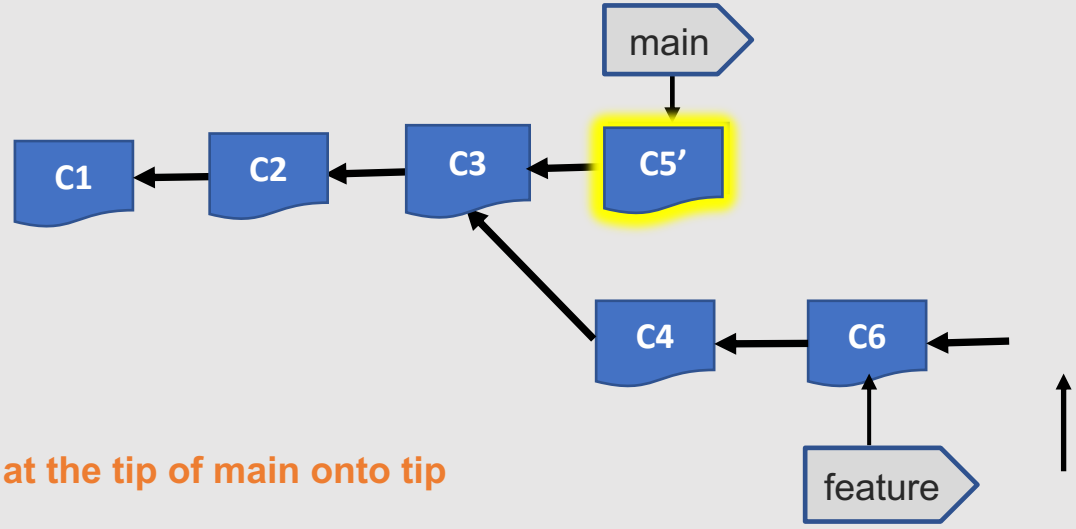
- "Given one or more existing commits, apply the change each one introduces, recording a new commit for each"
- Allows you to apply changes from specific commits
- Move a chunk of code from one branch to another
- Introduces the same change with a different parent
- Syntax: git cherry-pick <commit SHA1 or tag/ref>
- cherry-picks onto current branch
- cherry-pick does not record original reference like rebase, merge, etc.
- Other useful options:
  - -n - no-commit – only does it to your working directory and staging area
  - --edit – edit commit message before applying
  - --abort – end the cherry-pick in progress
  - -x - add line in commit message that records which commit cherry-picked from

rebase versus cherry-pick



$ git checkout feature
$ git rebase main

(merge each commit from tip of feature back to common ancestor, one at a time, onto tip of main)

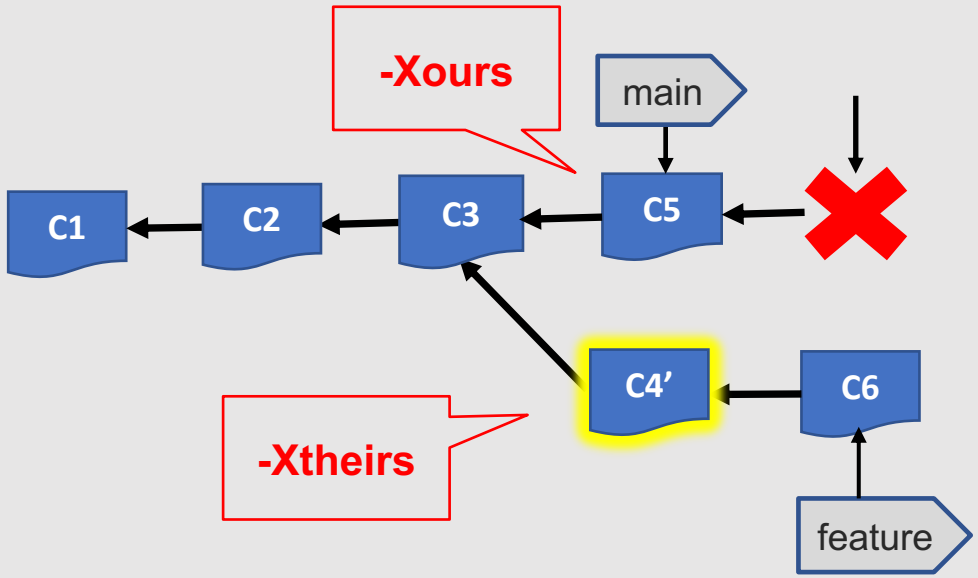


$ git checkout feature
$ git cherry-pick main

(merge the single commit at the tip of main onto tip of feature)

Merge Strategies

- If not fast-forward
- Can pass in merge strategy to git commands via --strategy. Options are:
  - resolve – merges two branches via 3-way merge
  - ort (default) - replacement for recursive (faster) - similar options to recursive
  - recursive – merges two branches via 3-way merge (if more, does additional pre-merges)
    - » pass in options to merge strategy via –X<option>
      - » ours – favors current branch when conflicts are found
      - » theirs – favors other branch when conflicts are found
      - » patience – does extra processing to try and avoid conflicts – best for very diverged branches
      - » other options including ones to ignore space/eol
  - octopus – merges multiple branches
  - ours – always use current results
  - subtree – modified recursive – makes subtrees match structure first




```
$ git checkout main
$ git cherry-pick feature~1

error: could not apply cb6c3e3... C4'
hint: after resolving the conflicts, mark
the corrected paths
hint: with 'git add <paths>' or 'git rm
<paths>'
hint: and commit the result with 'git
commit'

$ git cherry-pick -Xtheirs feature~1
```

Lab 10 - Working with cherry-pick

Purpose: In this lab, we'll see how to work with the cherry-pick command, merge conflicts with it and an alternative merge strategy

Command: grep

- Purpose - provides a convenient (and probably familiar) way to search for regular expressions in your local Git environment.
- Use case - self-explanatory
- Syntax:

```
git grep [-a | --text] [-I] [--textconv] [-i | --ignore-case] [-w | --
word-regexp]
           [-v | --invert-match] [-h|-H] [--full-name]
           [-E | --extended-regexp] [-G | --basic-regexp]
           [-P | --perl-regexp]
           [-F | --fixed-strings] [-n | --line-number] [--column]
           [-l | --files-with-matches] [-L | --files-without-match]
           [(-O | --open-files-in-pager) [<pager>]]
           [-z | --null]
           [ -o | --only-matching ] [-c | --count] [--all-match] [-q | --
quiet]
           [--max-depth <depth>] [--[no-]recursive]
           [--color[=<when>] | --no-color]
           [--break] [--heading] [-p | --show-function]
           [-A <post-context>] [-B <pre-context>] [-C <context>]
           [-W | --function-context]
           [--threads <num>]
           [-f <file>] [-e] <pattern>
           [--and|--or|--not|(|)|-e <pattern>…]
           [--recurse-submodules] [--parent-basename <basename>]
           [ [--[no-]exclude-standard] [--cached | --no-index | --
untracked] | <tree>…]
           [--] [<pathspec>…]
```

- Notes
  - Several options are similar to OS grep options

grep

- Default behavior - search for all instances of an expression across all tracked files in working directory
- Search for all instances off expression “database” across all java files (note use of --)

```
$ git grep database -- *.java
api/src/main/java/com/demo/pipeline/status/status.java:      @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

- -p option tells Git to try and show header of method or function where search target was found
- --break - make output easier to read
- --heading - prints filename above output

```
$ git grep -p --break --heading database -- *.java

api/src/main/java/com/demo/pipeline/status/status.java
13=public class V1_status {
31:     @Path("/database")

dataaccess/src/main/java/com/demo/dao/MyDataSource.java
18=public class MyDataSource {
64:                 logger.log(Level.SEVERE, "Could not access database via
connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

- boolean operators
- search in staging area
- search in specific commit(s)

```
$ git grep -e 'database' --and -e 'access' -- *.java
```

```
$ git grep -e 'config' --cached -- '*.txt'
```

```
$ git grep -e 'database' HEAD  -- *.java
$ git grep -e 'database' b2e575a  -- *.java
```

Command: notes

- Create a note

```
$ git notes add -m "This is an example of a note" 2f2ea1e
```

- Create a note in a custom namespace (add --ref)

```
$ git notes --ref=review add -m "Looks ok to me" f3b05f9
```

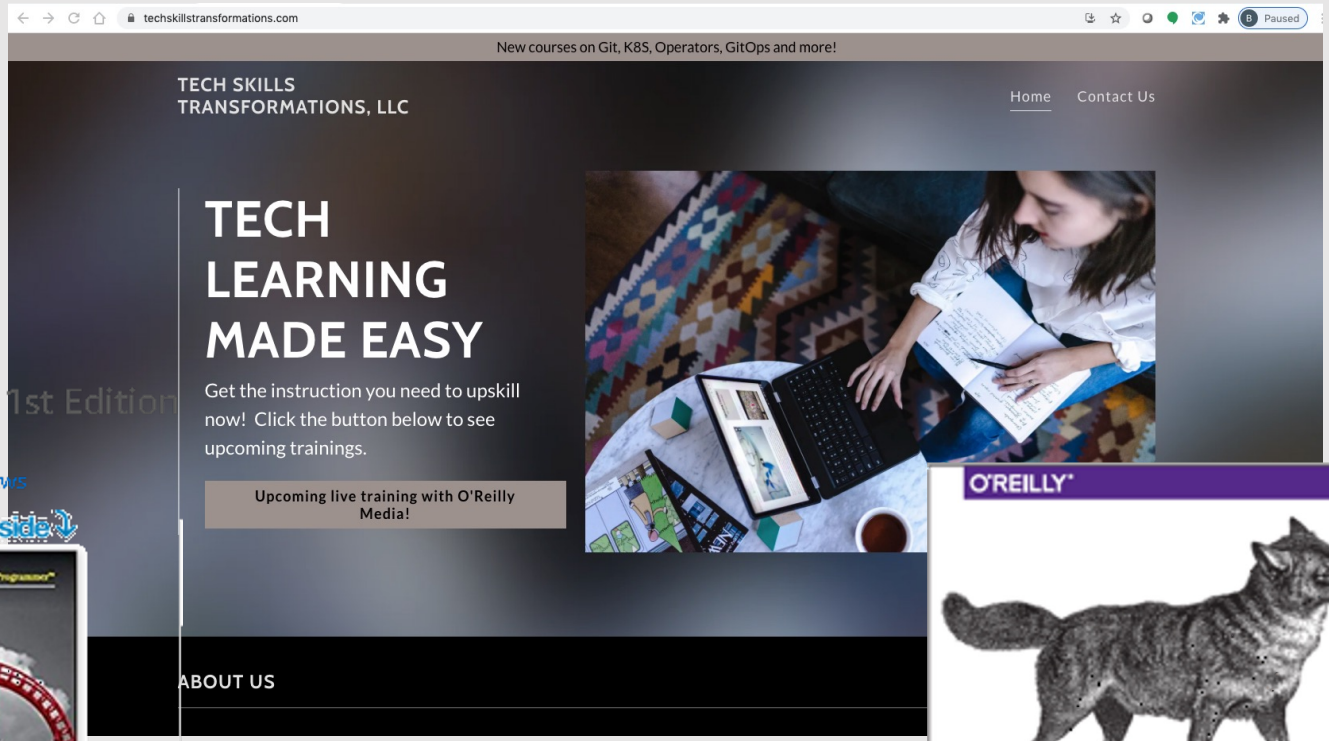
- View a note (for a specific revision)

```
$ git notes show 2f2ea1e
This is an example of a note
```

- List notes in log

```
$ git log --show-notes=*
commit 80e224b24e834aaa8915e3113ec4fc635b060771
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Jul 1 13:01:58 2016 -0400

commit ef15dca5c6577d077e38a05b80670024e1d92c0a
Author: unknown <bcl@nclasters.org>
Date:   Fri Apr 24 12:32:50 2015 -0400

    Removing test subdir on master

commit f3b05f9c807e197496ed5d7cd25bb6f3003e8d35
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 19:56:39 2015 -0400

    update test case

Notes (review):
    Looks ok to me

commit 2f2ea1e30fe46306294773 38a0ab8618569f0f5e
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 17:34:57 2015 -0400

    Add in testing example files

Notes:
    This is an example of a note
```

That's all - thanks!



techskillstransformations.com
getskillsnow.com