Version 1.7
12/12/23

Git in 4 Weeks

(Part 4)

Presented by Brent Laster &

Tech Skills Transformations LLC

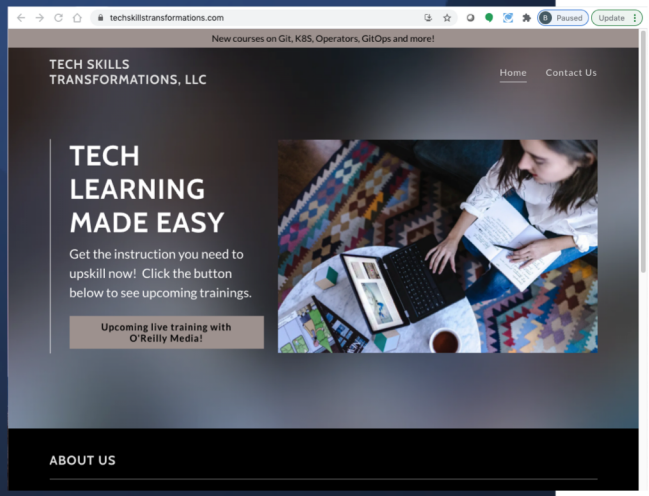
TECHUPSKILLS ®

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/git4-4-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

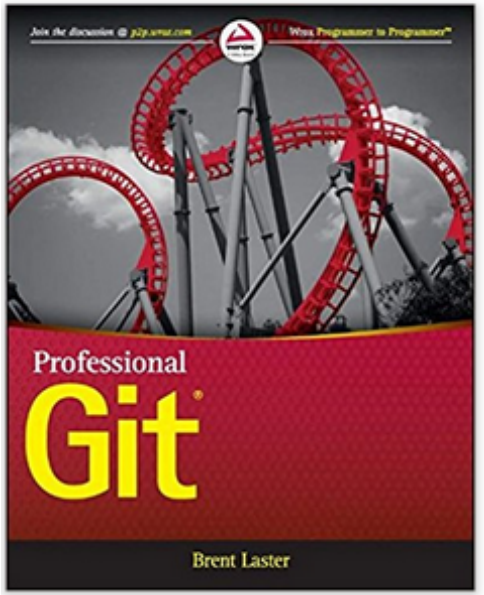
- Extensive Git reference, explanations,
- and examples
- First part for non-technical
- Beginner and advanced reference
- Hands-on labs

Professional Git 1st Edition

by Brent Laster (Author)

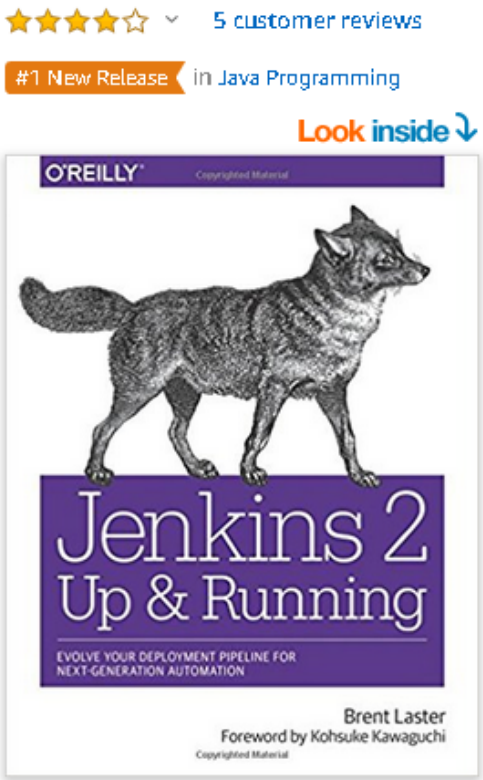
7 customer reviews

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***



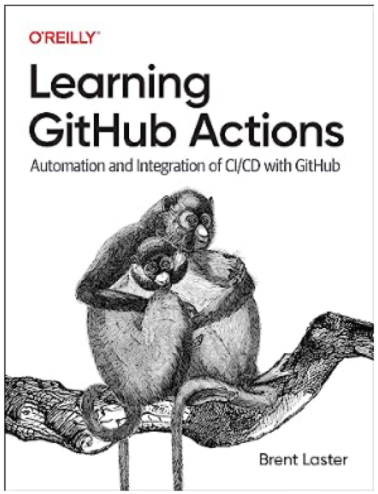
GitHub Actions book

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

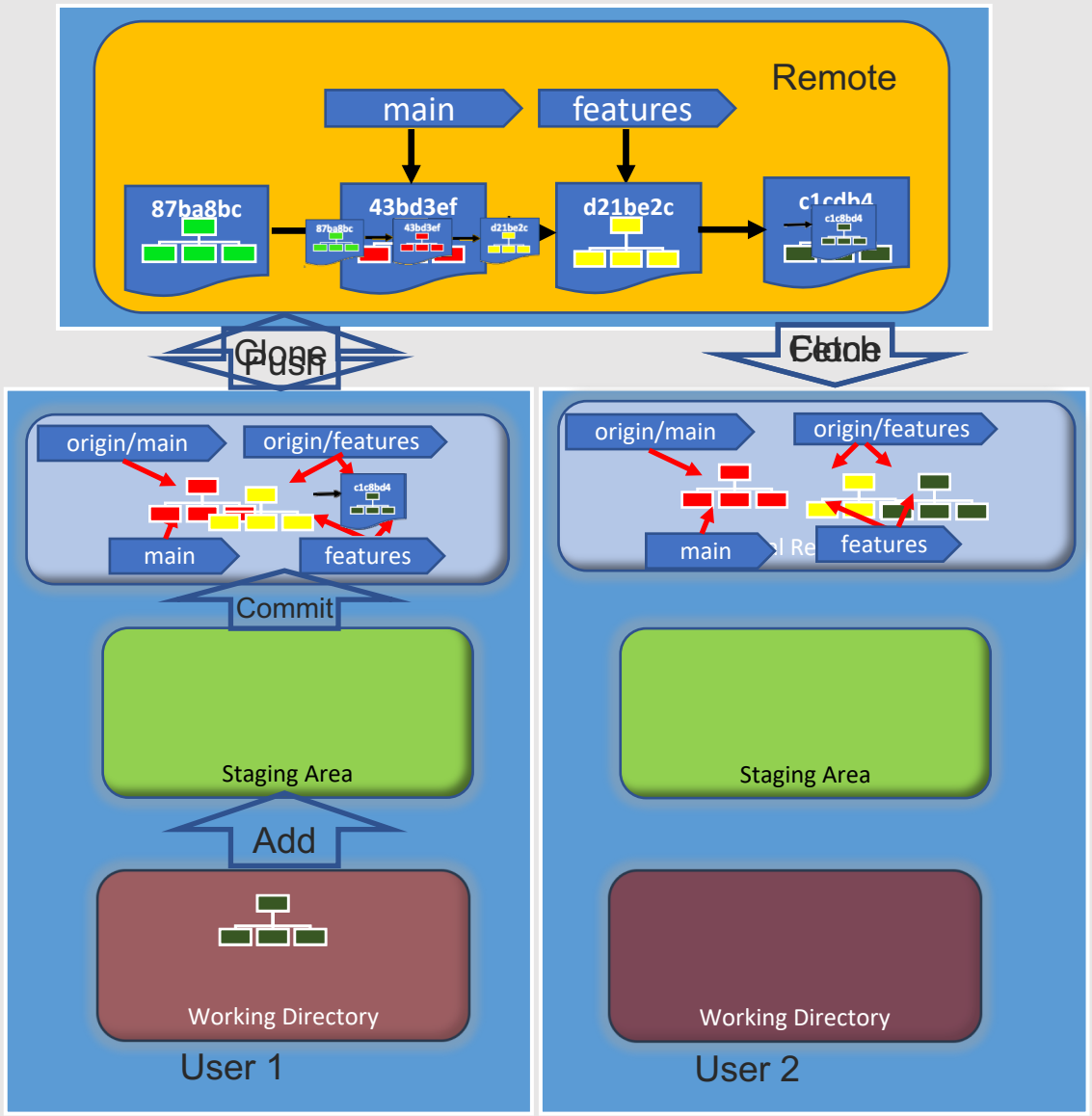
Remote vs Local Branches

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



Switching between Branches

Command: git checkout <branch>

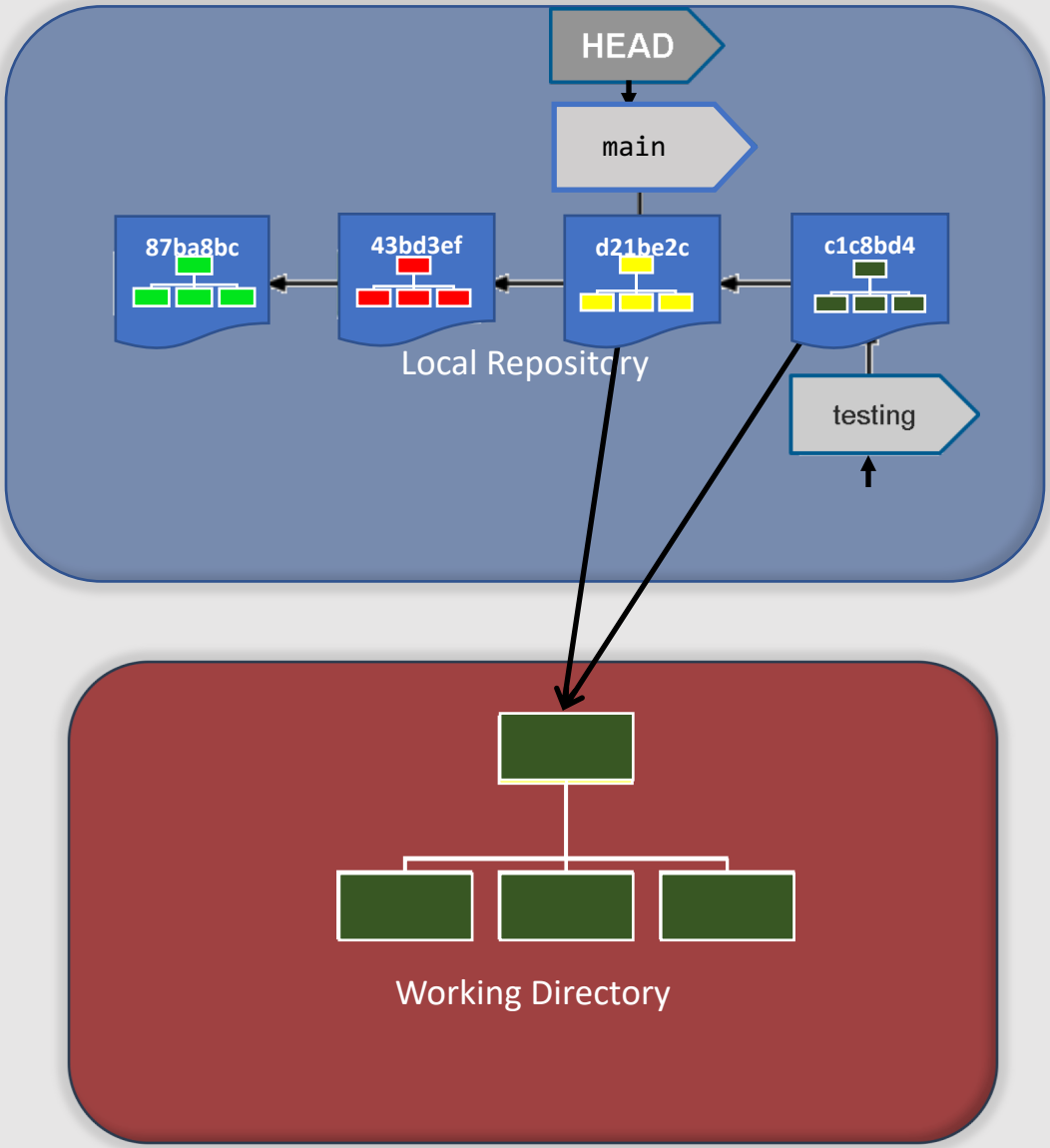
git checkout main

- **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



New Content

Command: Worktrees

- Purpose - Allows multiple, separate Working Areas attached to one Local Repository
- Use case - Simultaneous development in multiple branches
- Syntax

```
git worktree add [-f] [--detach] [--checkout] [--lock] [-b <new-branch>]
<path> [<commit-ish>]
git worktree list [--porcelain]
git worktree lock [--reason <string>] <worktree>
git worktree move <worktree> <new-path>
git worktree prune [-n] [-v] [--expire <expire>]
git worktree remove [-f] <worktree>
git worktree unlock <worktree>
```

- Notes
  - "Traditional" working directory is called the *main working tree;* Any new trees you create with this command are called *linked working trees*
  - Information about working trees is stored in the .git area (assuming .git default GIT_DIR is used)
  - Working tree information is stored in .git/worktrees/<*name of worktree*>.

Worktrees - syntax and usage

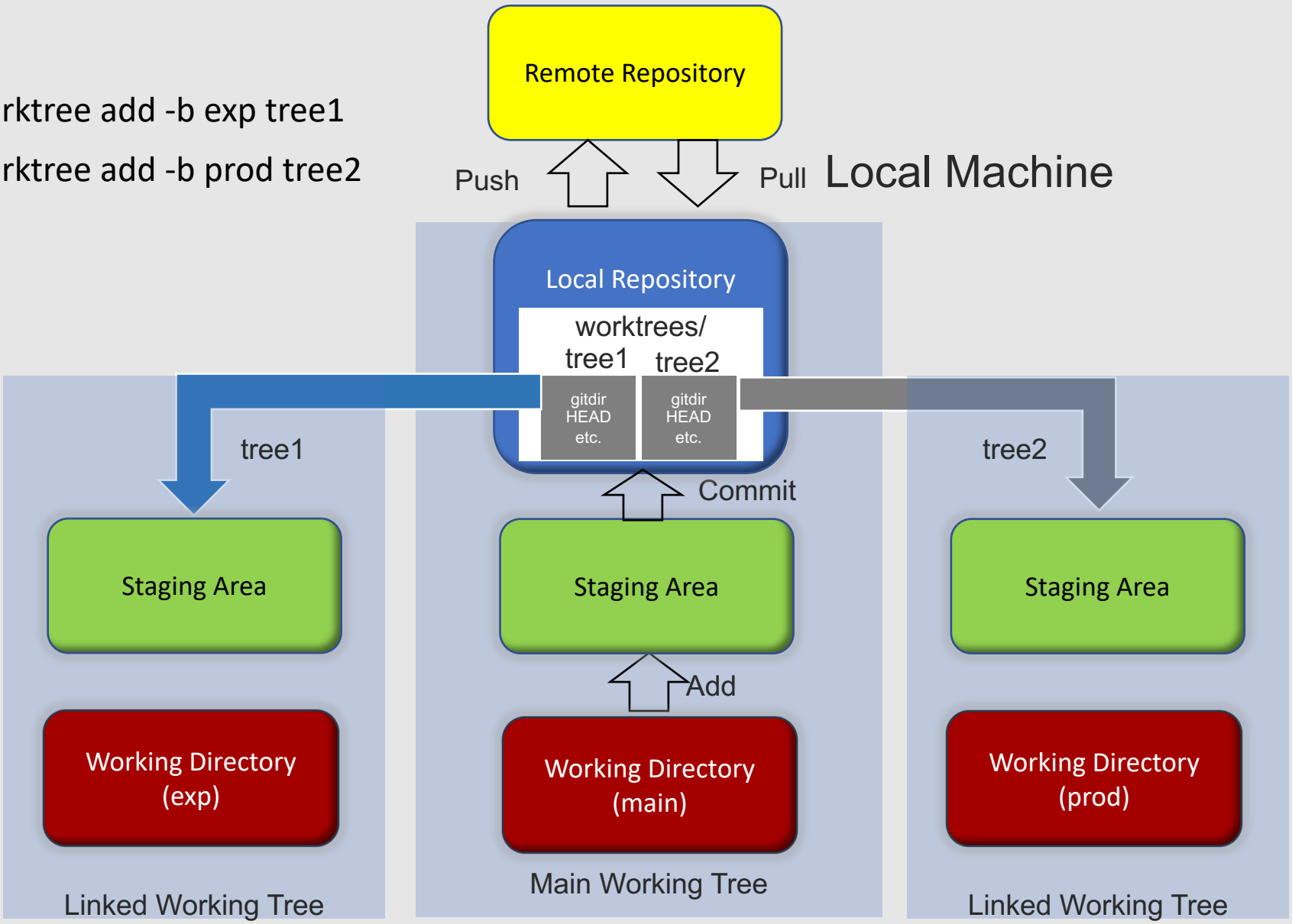
- Syntax
  - *git worktree add* [-f] [--detach] [--checkout] [--lock] [-b <new-branch>] <path> [<branch>]
  - *git worktree list* [--porcelain]
  - *git worktree prune* [-n] [-v] [--expire <expire>]
- Subcommands
  - Add - create a separate working tree for specified branch
    - » has checked out copy of the branch
    - » if no branch specified, uses same name as temp area
  - List - lists out current set of working trees active for this repo
    - » porcelain - consistent and backwards-compatible format
  - Prune - removes worktree information from the .git area
    - » only applies after worktree directory tree has been removed

Worktrees - meta-information

- Information about working trees is stored in the .git area (assuming .git default GIT_DIR is used)
- Working tree information is stored in .git/worktrees/<*name of worktree*>.
- Working trees can be created on removable media
  - To persist after unmounting use “lock” subcommand or option on add
  - Use “--reason” to specify a reason for the lock if using lock subcommand

Worktrees

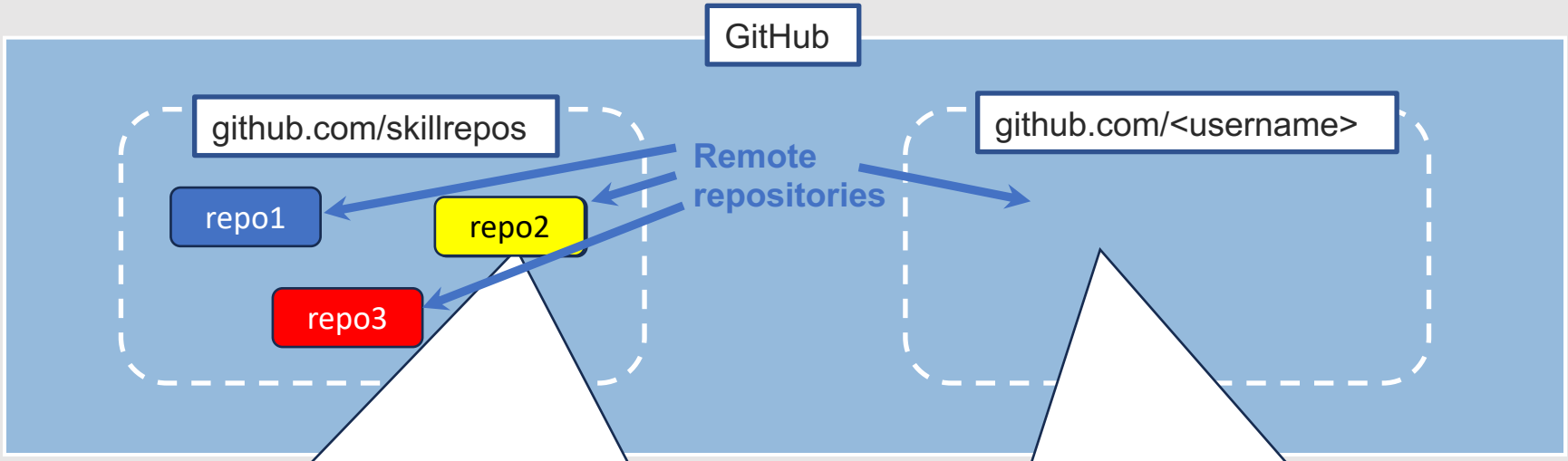
- git worktree add -b exp tree1
- git worktree add -b prod tree2

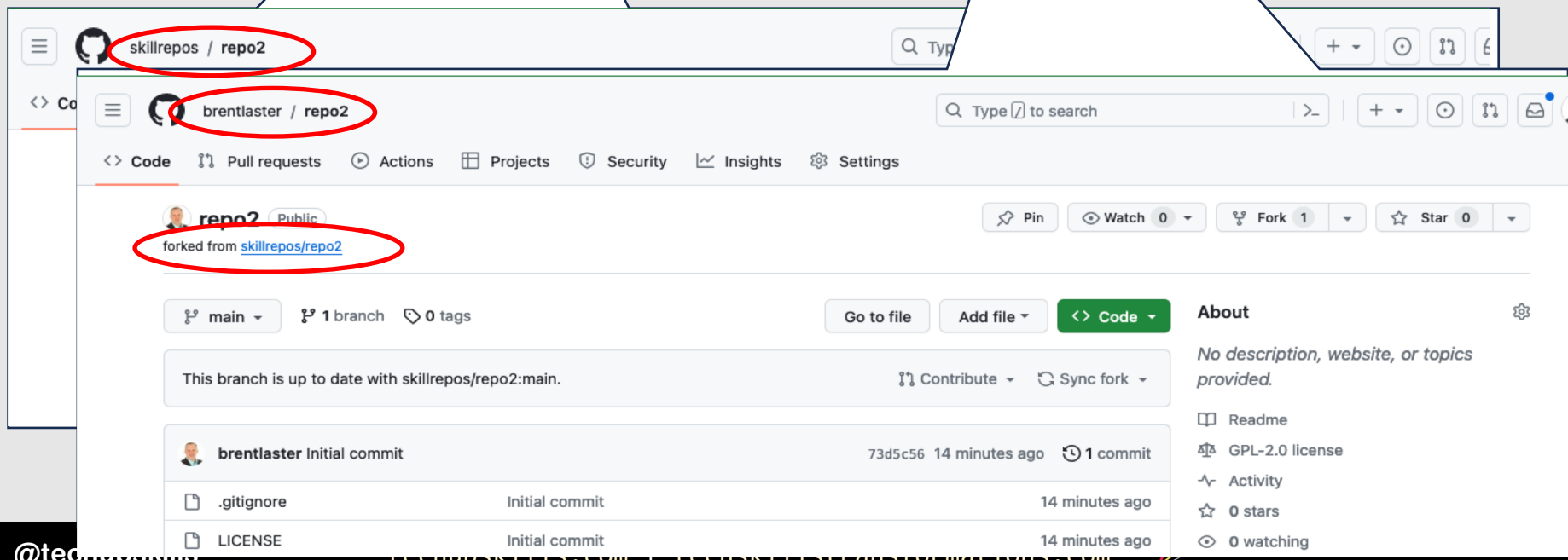


Lab 11 - Working with Worktrees

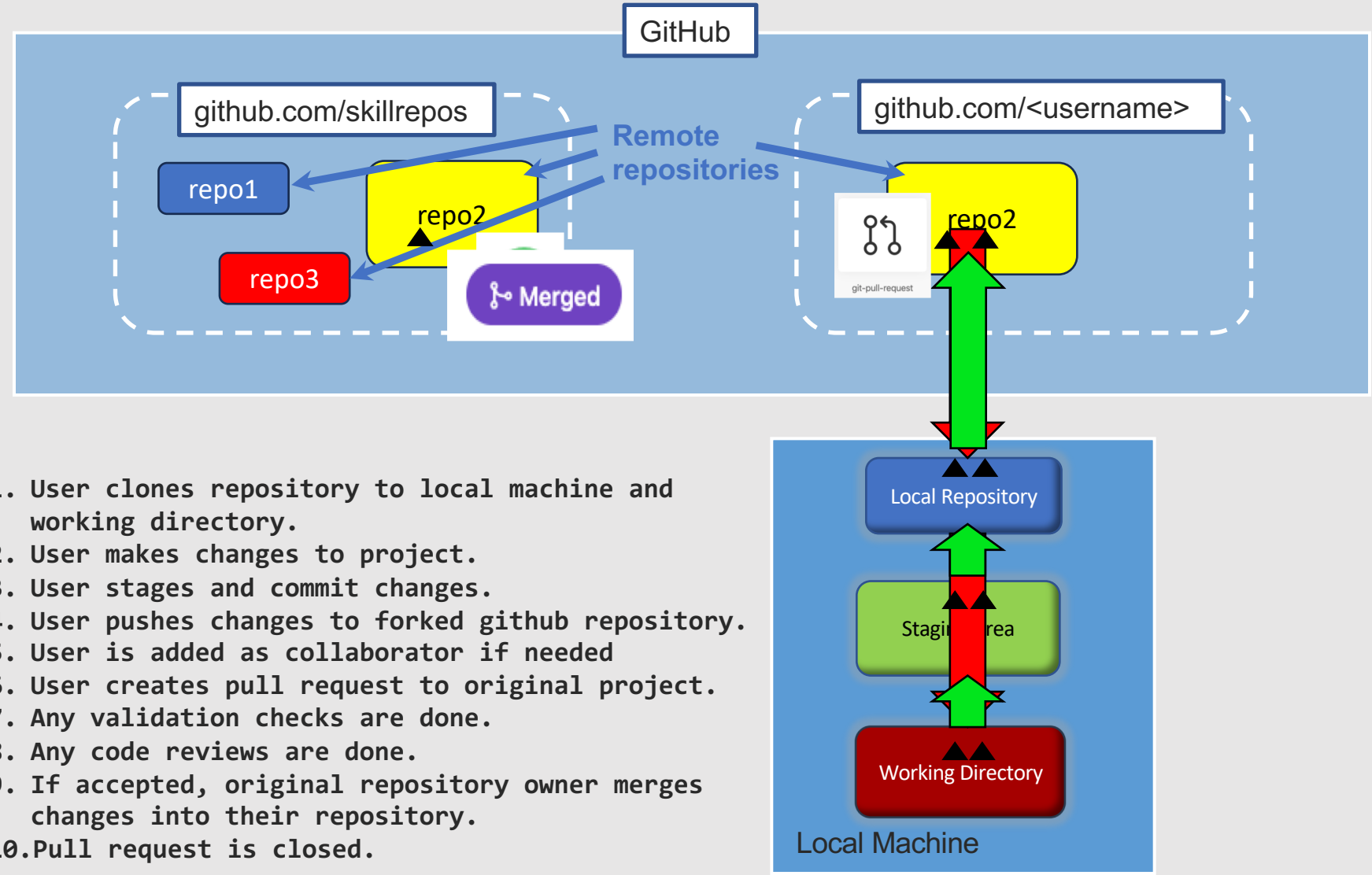
Purpose: In this lab, we'll get some experience with how to work on multiple branches at the same time

Fork model in GitHub



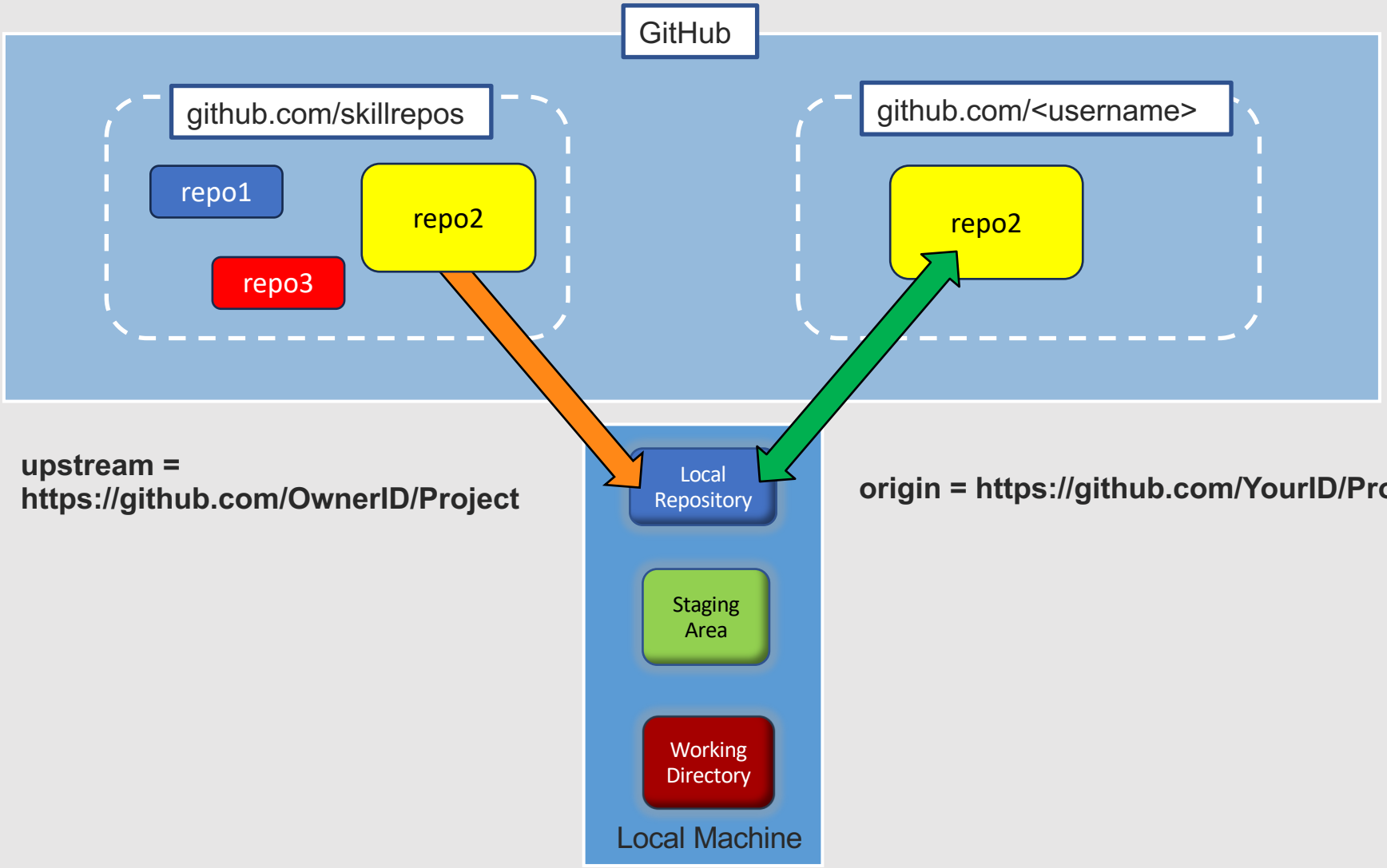


Updating a fork



1. User clones repository to local machine and working directory.
2. User makes changes to project.
3. User stages and commit changes.
4. User pushes changes to forked github repository.
5. User is added as collaborator if needed
6. User creates pull request to original project.
7. Any validation checks are done.
8. Any code reviews are done.
9. If accepted, original repository owner merges changes into their repository.
10. Pull request is closed.

Side note: Multiple Remotes

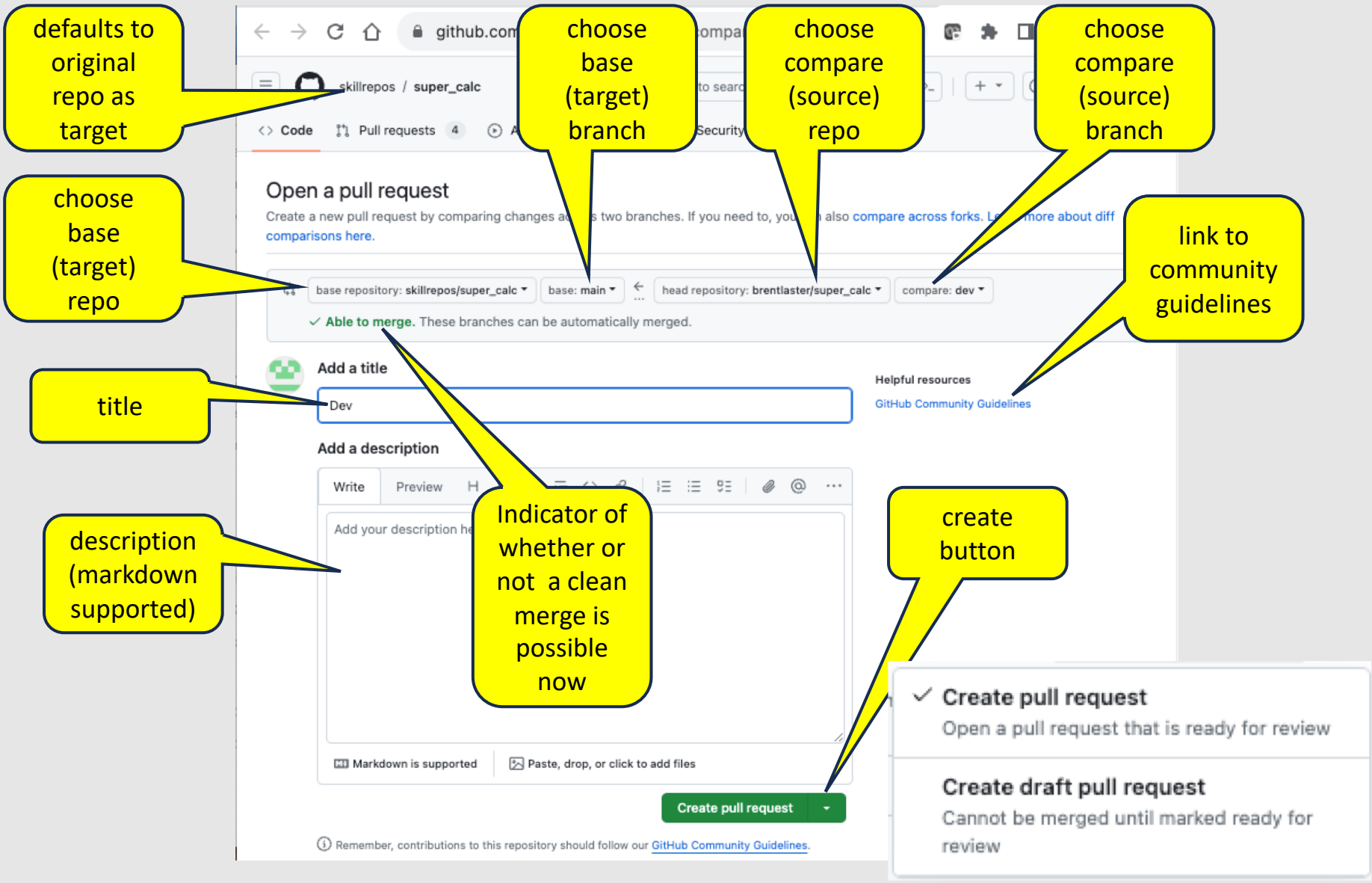


PR Prompt on Command Line

- When push is done to new branch...

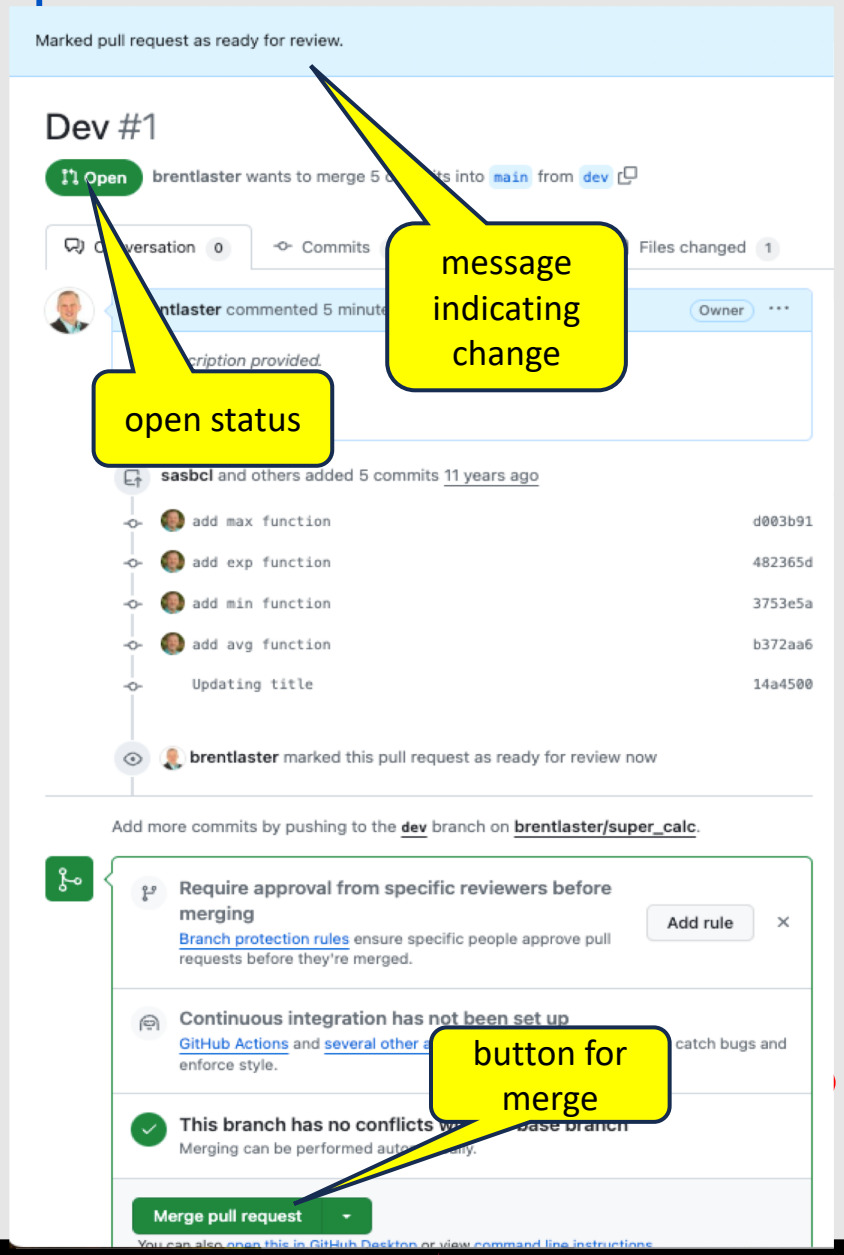
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'test' on GitHub by visiting:
remote: https://github.com/<your github userid>/super_calc/pull/new/test
remote:
To https://github.com/<your github userid>/super_calc.git
* [new branch] test -> test

Opening a pull request



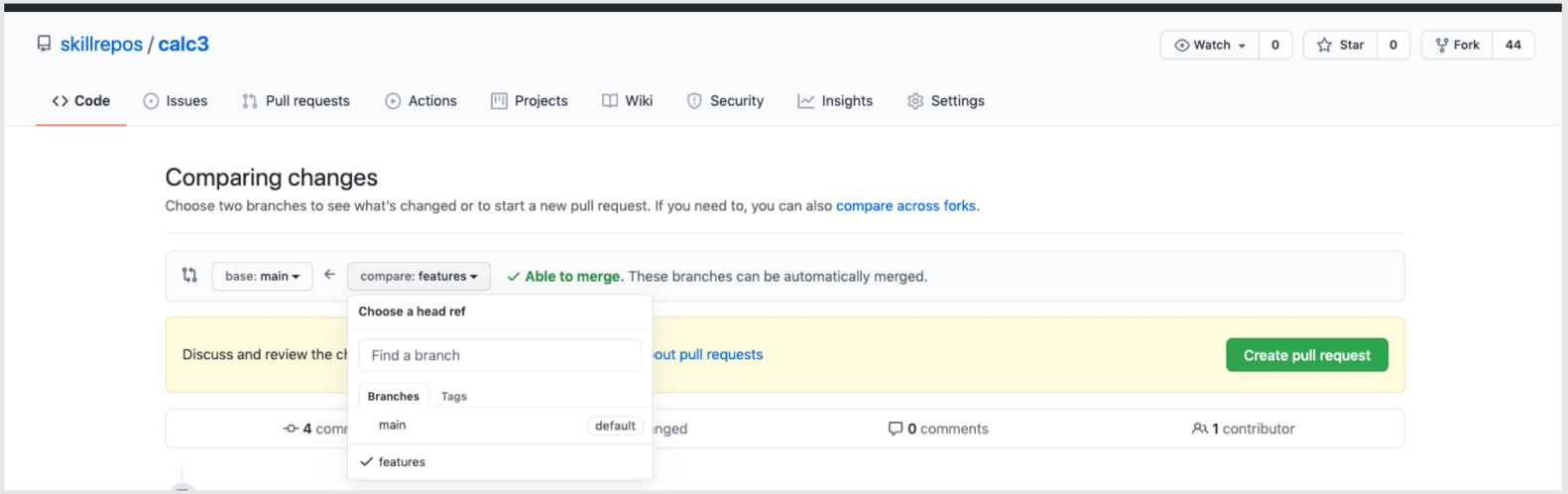
Creating a draft pull request

- Can create as draft vs open
- draft indicates not ready for review/moving forward
- cannot be merged until changed
- members of CODEOWNERS file not signalled
- change from draft to open with button
- status flips from draft to open

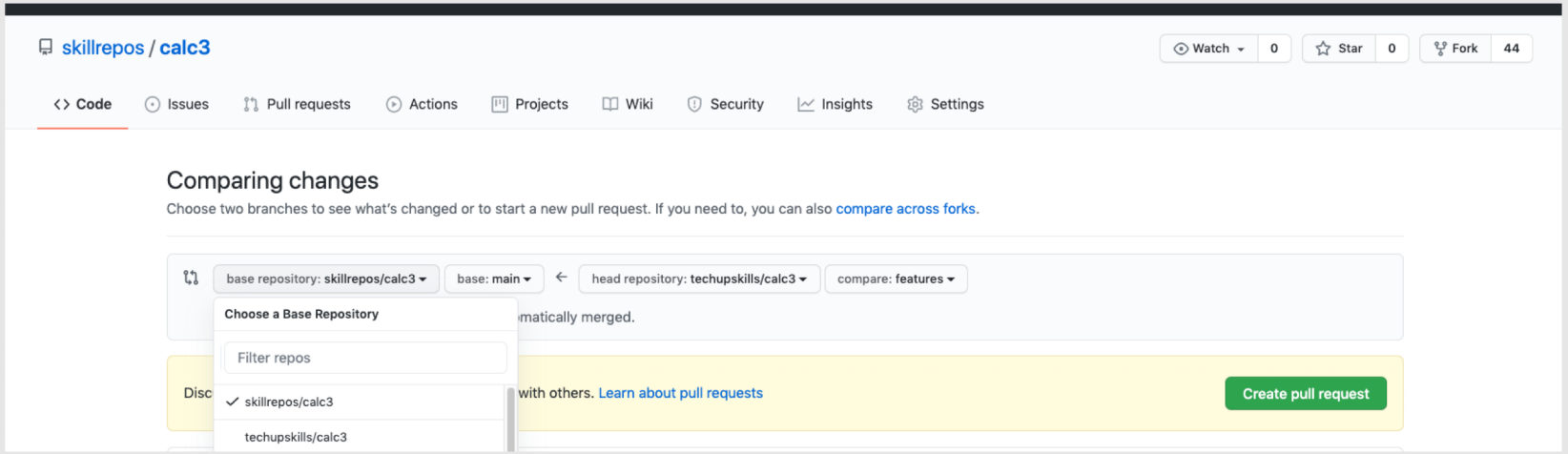


Pull Requests - Choosing what to compare

- Can compare/open PR for branches in same project

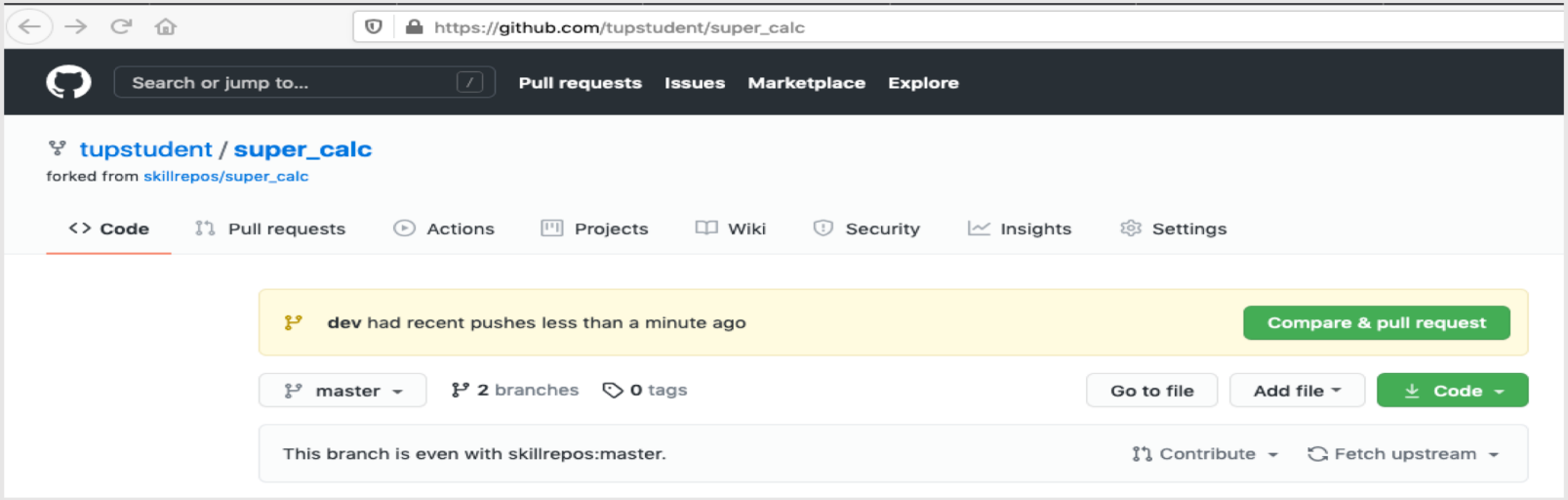


- Can compare/open PR for different projects (across forks)

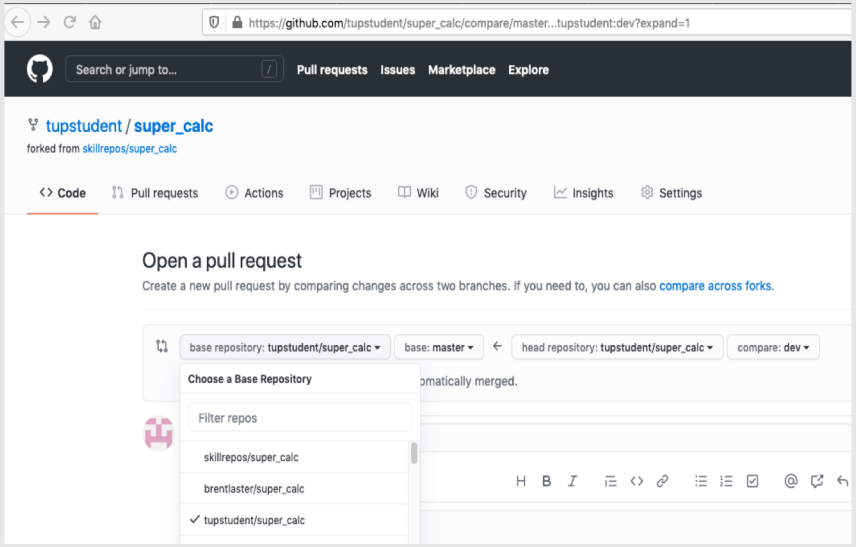


Working with Pull Requests

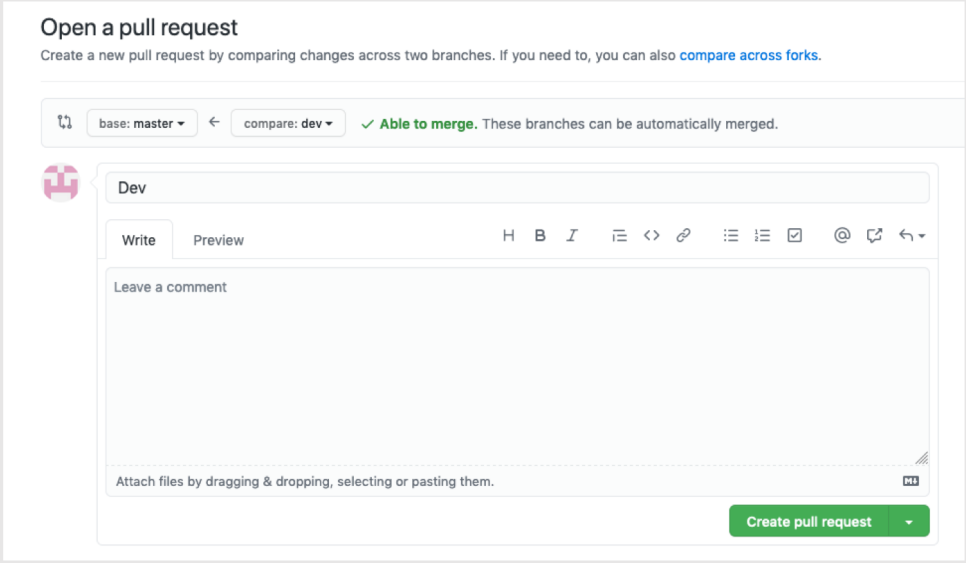
- After push to branch, have opportunity to create PR



- Select two items for compare/merge

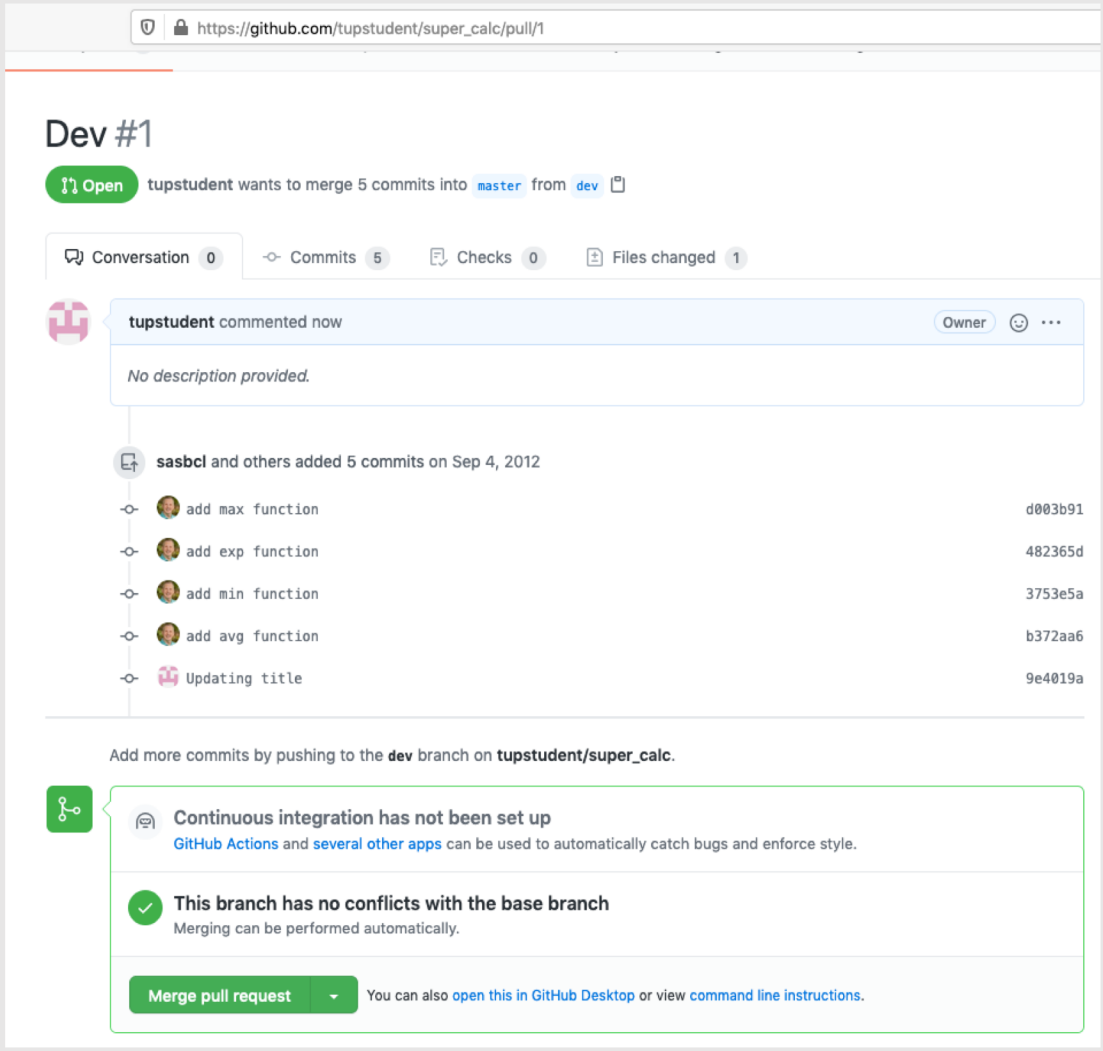


- Create the PR



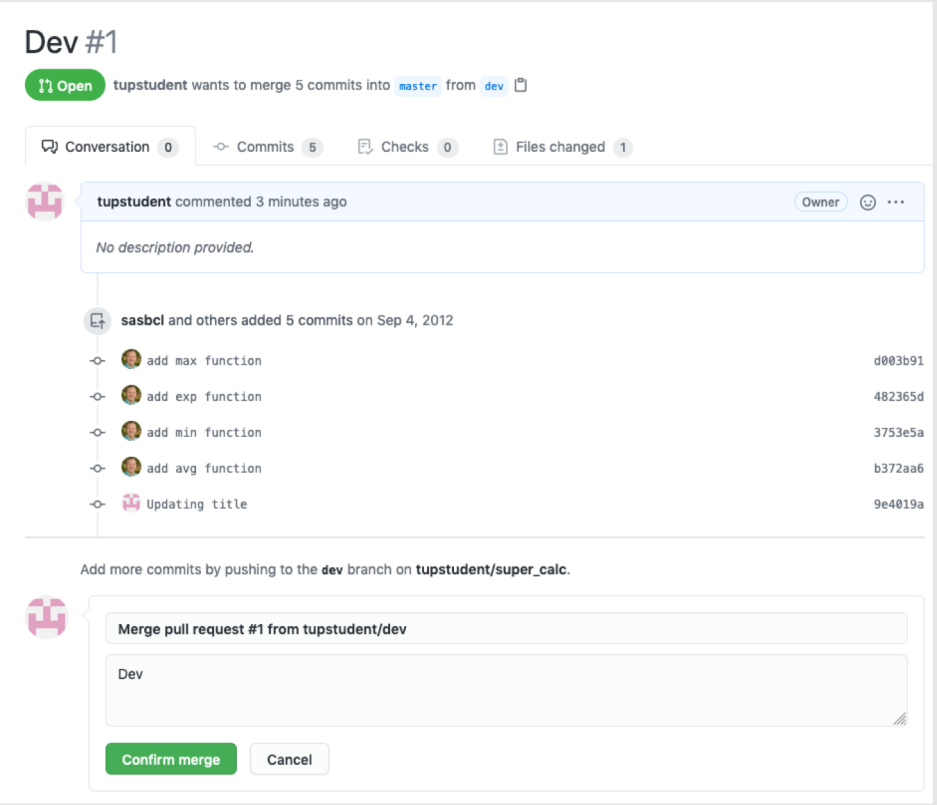
Pull Request in Progress

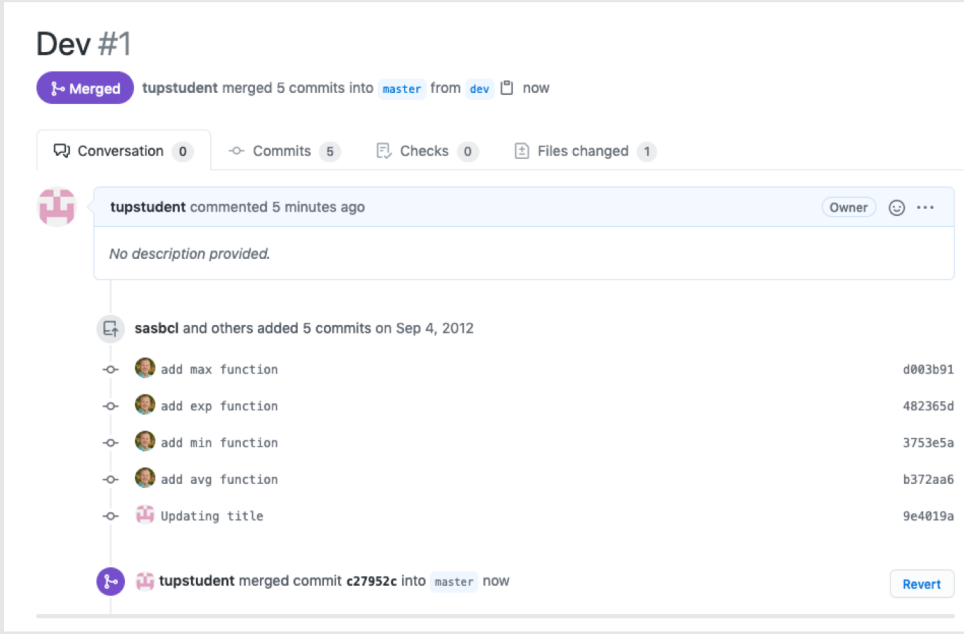
- New PR is opened
- Changes available for preview
- Continuous integration run if setup
- Check done for conflicts
- Once all is good can select to “Merge pull request”



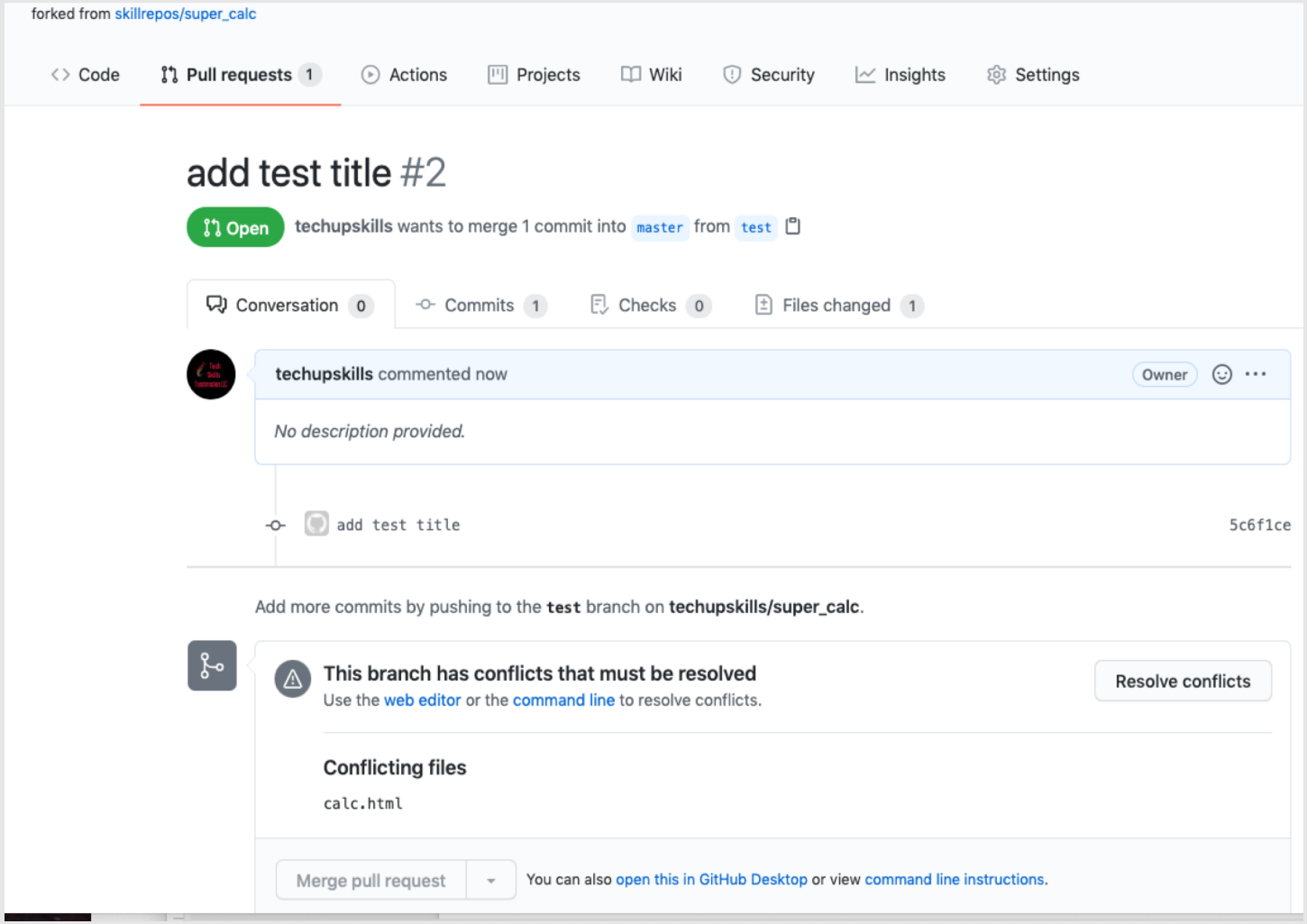
Pull Request Completion

- Confirm merge
- Shows success

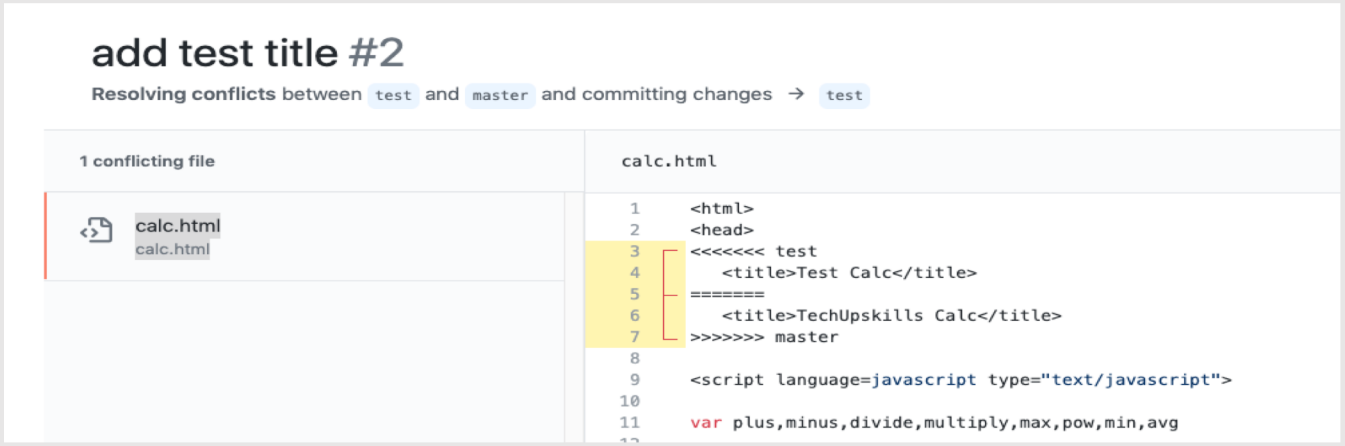


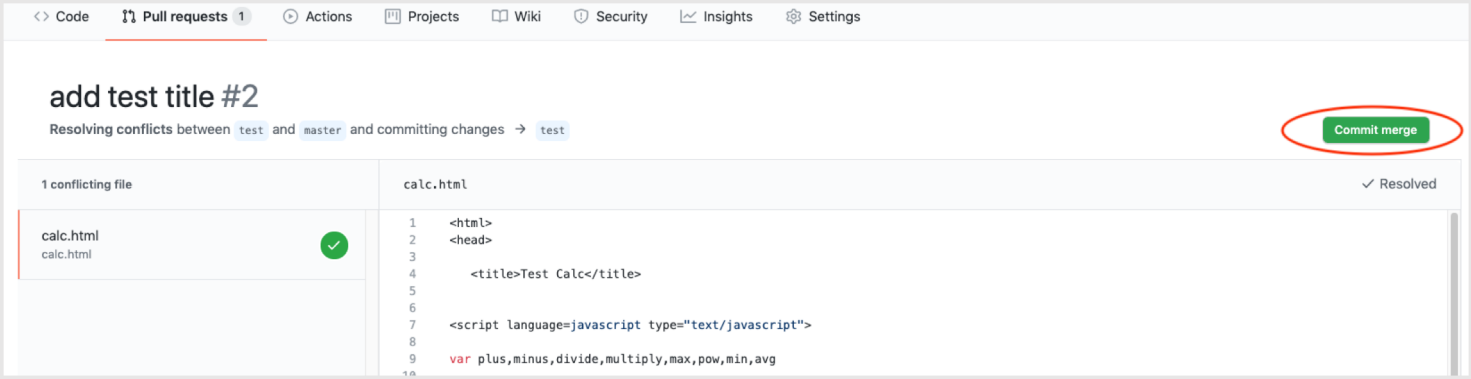


PR with merge conflicts

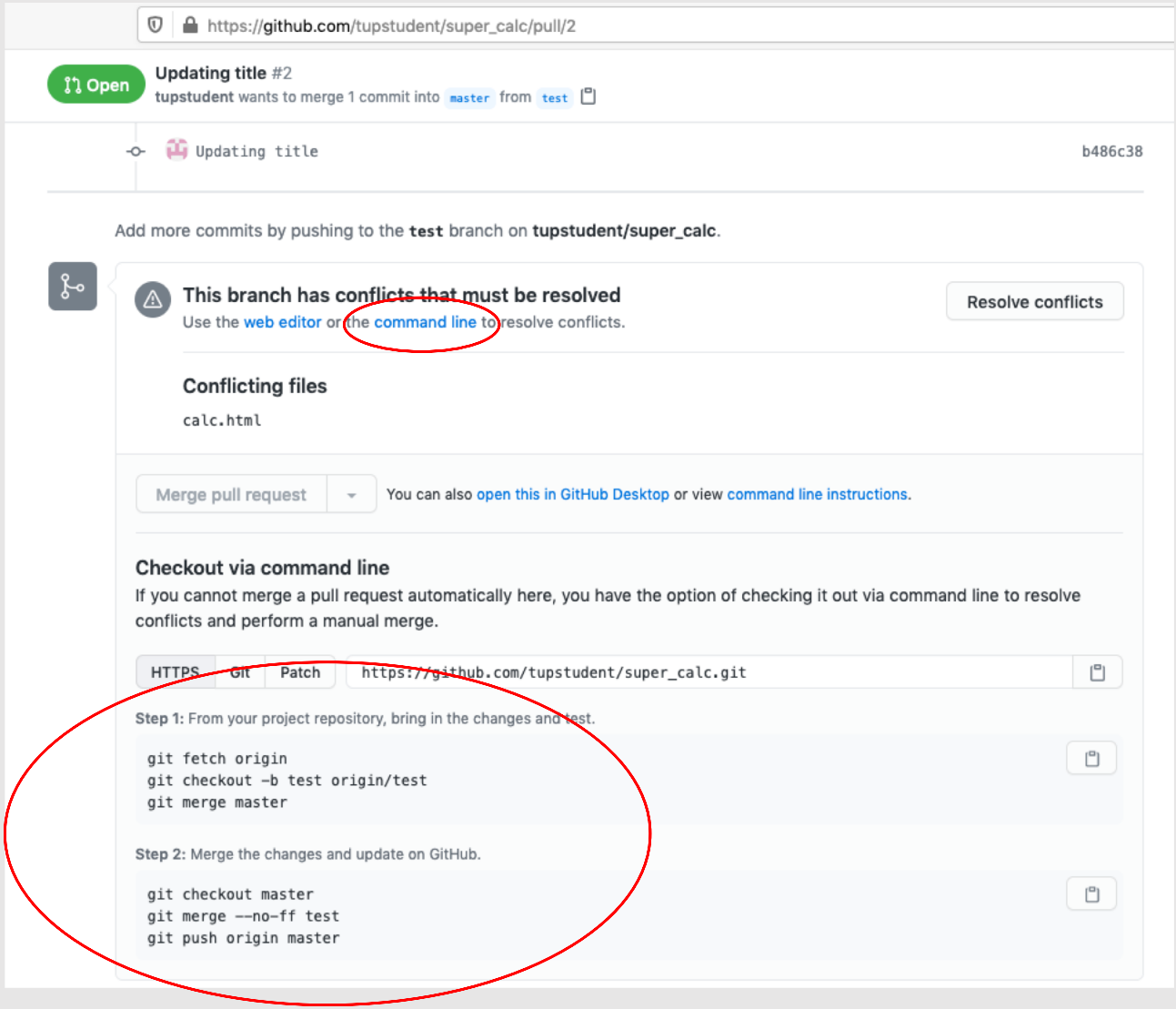


PR with conflicts - resolve in GitHub

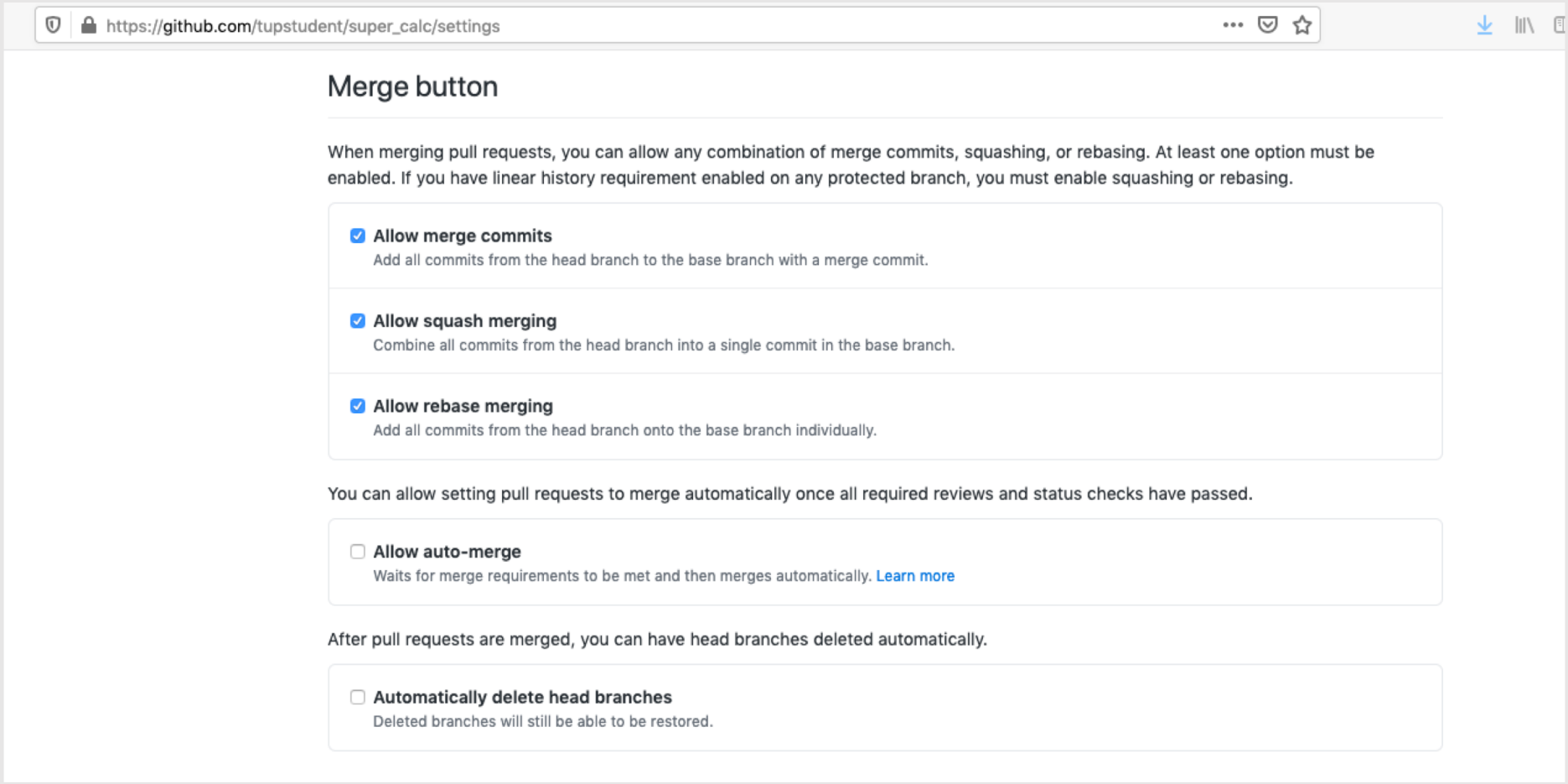




PR with conflicts - resolve with command line



GitHub Project Settings for PRs



Lab 12 - Working with Pull Requests

Purpose: In this lab, we'll see how to create and merge Pull Requests within a GitHub repository

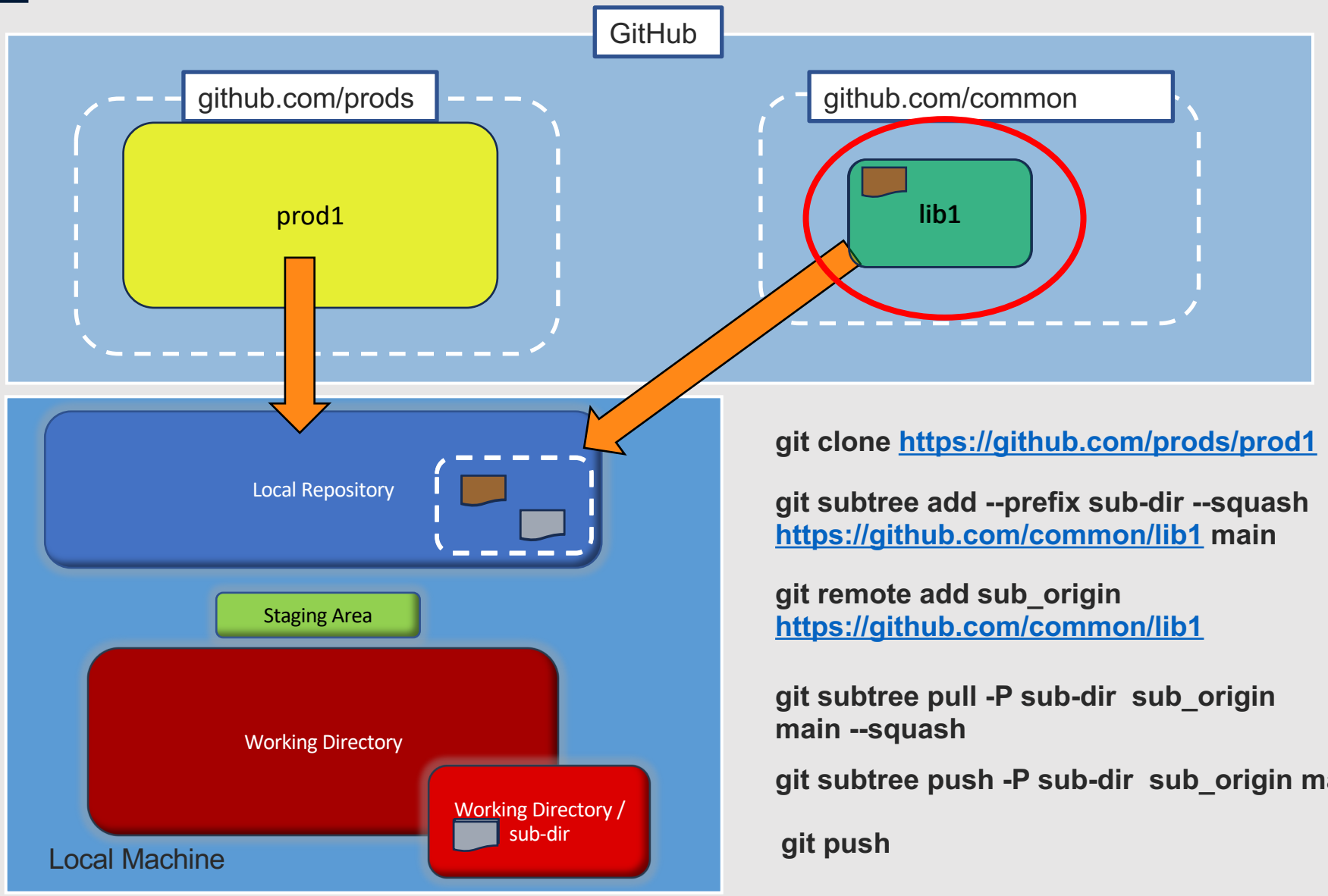
Command: Subtrees

- Purpose - Allows "nesting" a separate repository within your current repository
- Use case - include **a copy** of a Git repository for one or more dependencies along with the original repository for a project
- Syntax

```
git subtree add   -P <prefix> <commit>
git subtree add   -P <prefix> <repository> <ref>
git subtree pull  -P <prefix> <repository> <ref>
git subtree push  -P <prefix> <repository> <ref>
git subtree merge -P <prefix> <commit>
git subtree split -P <prefix> [OPTIONS] [<commit>]
```

- Notes
  - No links like a submodule - just a copy in a subdirectory
  - Advantage - no links to maintain
  - Disadvantage - extra content to carry around with your project

How subtrees work



git clone https://github.com/prods/prod1

git subtree add --prefix sub-dir --squash https://github.com/common/lib1 main

git remote add sub_origin https://github.com/common/lib1

git subtree pull -P sub-dir sub_origin main --squash

git subtree push -P sub-dir sub_origin main

git push

Subtrees - adding a project as a subtree

- Simplest form, specify: prefix, remote path to repository, branch (optional)
- Suppose we clone down a project - myproject
- And, on the remote side, we have another project, subproj.
- Now, we can add subproj as a subtree of myproject (--prefix subproject).
- Subtree now shows subproject in structure and history

```
developer@Bs-MacBook-Pro super_calc % tree
.
└── calc.html

1 directory, 1 file
```

```
$ git subtree add --prefix sub_docs --squash
https://github.com/<github-user>/sub_docs
main
```

```
developer@Bs-MacBook-Pro super_calc % tree
.
├── calc.html
└── sub_docs
    ├── advcalc.docx
    └── simplecalc.docx

2 directories, 3 files
```

```
developer@Bs-MacBook-Pro super_calc % git log --oneline
762a279 (HEAD -> main) Merge commit '3c7953b0a1cf9d7d7027668f85994e670e3416a2'
3c7953b Squashed 'sub_docs/' changes from a83878d..ddc2fc4
6738ef0 Merge commit 'd32a91e5d4d6dde31c674954fc2267c45ccdc5a4' as 'sub_docs'
d32a91e Squashed 'sub_docs/' content from commit a83878d
32d7b92 (origin/main, origin/HEAD) initial version
```

Subtrees - updating and merging

- Update command - similar to add

```
$ git subtree pull --prefix subproject sub_origin master --squash
```

- pulls down the latest content from the remote into the subtree area
- --squash compresses the history again
  - can be omitted, but usually simplifies things
- also a *git subtree merge* command to merge commits up to a desired point into a subproject denoted by the --prefix argument
  - can be used to merge local changes to a subproject, while git subtree pull reaches out to the remote to get changes
- subtree command also supports a push subcommand

```
~/subtrees/local/myproject$ git subtree push --prefix=subproject sub_origin new_branch
git push using:  sub_origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
To /Users/dev/subtrees/remotes/subproj.git
 * [new branch]      906b5234f366bb2a419953a1edfb590aadc32263 -> new_branch
~/subtrees/local/myproject$
```

Subtrees vs. subtree merge strategy

- In Git, there is also a merge strategy named *subtree*.
- git subtree command (actually a script that's been incorporated into Git) is not the same thing as the subtree merge strategy.
- Git chooses the best strategy depending on the situation.
- The subtree merge strategy is designed to be used when two trees are being merged and one is a subtree (subdirectory) of the other.
- subtree merge strategy tries to shift the subtrees to be at the same level in order to merge similar structures.
- For more information, search for *subtree* in the help page for merge.

Lab 13 - Subtrees

Purpose: In this lab, we'll see one way to manage "groups" of repositories in Git

Hooks – basic concepts

- Definition – a program or script that runs when a certain event happens
- Common uses
  - Sending email or other notifications when a change is pushed to a repository
  - Validating that certain conventions have been met before a commit
  - Appending items to commit messages
  - Checking the format or existence of certain elements in a commit message
  - Updating content in the working directory after an operation
  - Enforcing coding standards

Hooks - accessing

- By default, hooks come from Git template area and live in .git/hooks
- As of Git 2.9, can have different path to hooks
  - Set by core.hooksPath config value
  - Provides means to have common path to hooks for multiple users or projects (shared hooks)
- After cloning or init, .git/hooks is populated with sample hooks

applypatch-msg.sample
commit-msg.sample
post-update.sample
pre-applypatch.sample
pre-commit.sample
pre-push.sample
pre-rebase.sample
prepare-commit-msg.sample
update.sample

- Hooks are NOT cloned as part of Git clone
- Can be updated via “git init”

Hooks - attributes

- Domain
  - Local : applypatch-msg, pre-applypatch, post-applypatch, pre-commit, prepare-commit-msg, commit-msg, post-commit, pre-rebase, post-checkout, post-merge, pre-push, post-rewrite, push-to-checkout
  - Remote : pre-receive, update, post-receive, post-update, pre-auto-gc
- Return Code
  - 0 = success; operation should continue
  - Non-0 = not success; operation should abort
- Working Directory Location
  - Non-bare repo – working directory root
  - Bare repo - repository directory
- Environment Variables
  - GIT_DIR, GIT_AUTHOR_DATE, GIT_AUTHOR_EMAIL, GIT_AUTHOR_NAME

Hooks - am

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
am	applypatch-msg	pre-applypatch		post-applypatch	

- am command – takes a patch (may be from email) applies it, and commits the change
- applypatch-msg – operates on commit message for apply part of git am operation (mailing patches)
  - P1 – Name of temporary file with proposed commit message
- pre-applypatch – called after patch is applied, but before committed; allows Git to verify patch results look right
- post-applypatch – called after patch is applied AND committed; useful for notifications

Hooks - commit

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
commit	pre-commit	prepare-commit-msg	commit-msg	post-commit	

- pre-commit - verify that what's about to be committed meets some condition or criteria and is okay to commit
- prepare-commit-msg - Applypatch-msg – operates on commit message for apply part of git am operation (mailing patches)
  - P1: Name of temporary file with proposed commit message
  - P2: Type of operation
  - P3: SHA1 value for certain operations
- commit-msg – called after patch is applied, but before committed; allows Git to verify patch results look right
  - P1: Name of temporary file with proposed commit message
- post-commit – called after patch is applied AND committed; useful for notifications

Hooks - push

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
push	pre-push	pre-receive	update	post-receive	post-update

- pre-push – called prior to a push; can prevent a push
  - P1: Remote reference name
  - P2: Remote URL Extra: STDIN lines of the form <local reference> <local sha1> <remote reference> <remote sha1> LF
- pre-receive – runs once for push before updating references
  - No parameters - Extra: STDIN lines of the form <local reference> <local sha1> <remote reference> <remote sha1> LF
- update – runs once for each reference to be updated
  - P1: Name of reference being updated
  - P2: Old SHA1 value
  - P3: New SHA1 value
- post-receive – runs once after all references are updated; knows old and new
  - No parameters Extra: STDIN lines of the form <old value> <new value> <reference name> LF
- post-update – runs after all references; doesn't know old and new
  - P* (variable number): Name of references being updated

Hooks - rebase

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
rebase	pre-rebase				

- pre-rebase - provides an opportunity to validate that the rebase should go through, issue a warning message, and so on
  - P1: Upstream that the current series of commits come from
  - P2: Branch being rebased (if not the same as P1)
- Example hook – prevents topic branches that have already being merged from being rebased (and so not merged again)

Hooks – push (to non-bare repository)

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
push (to non-bare repo)	push-to-checkout				

- non-bare repository – one that has working directory and staging area and checked out branch
- normally push to bare remote repository only, so receive.denyCurrentBranch setting to prevent pushes to non-bare
- push-to-checkout – allows to override receive.denyCurrentBranch and sync working directory and staging area to make everything consistent
  - P1: Target commit for updating

Hooks – commit --amend or rebase

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
commit – amend or rebase				post-rewrite	

- post-rewrite – runs after operations that change history (rewrite commits)
  - P1: command that called it
  - <stdin> : <old sha1> <new sha1> [<optional extra data>]
- currently no extra data is passed, but might be in future
- runs after commit –amend or rebase (but not filter-branch)
- similar uses to post-checkout and post-merge

Hooks – gc --auto

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
gc --auto	pre-gc-auto				

- gc – run garbage collection (cleaning up objects that aren't used anymore)
  - auto = option to cleanup if there are too many loose objects over a configured threshold
- hook runs first if --auto option is specified
  - No parameters
- can do notification and/or verification before gc

Hooks - checkout

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
checkout				post-checkout	

- post-checkout – runs after a successful checkout (or after clone unless no-checkout option is used)
  - P1: Previous HEAD (before checkout)
  - P2: Current HEAD (after checkout)
  - P3: Checkout type flag: 1= branch, 2 = file
- can be used to cleanout unwanted files

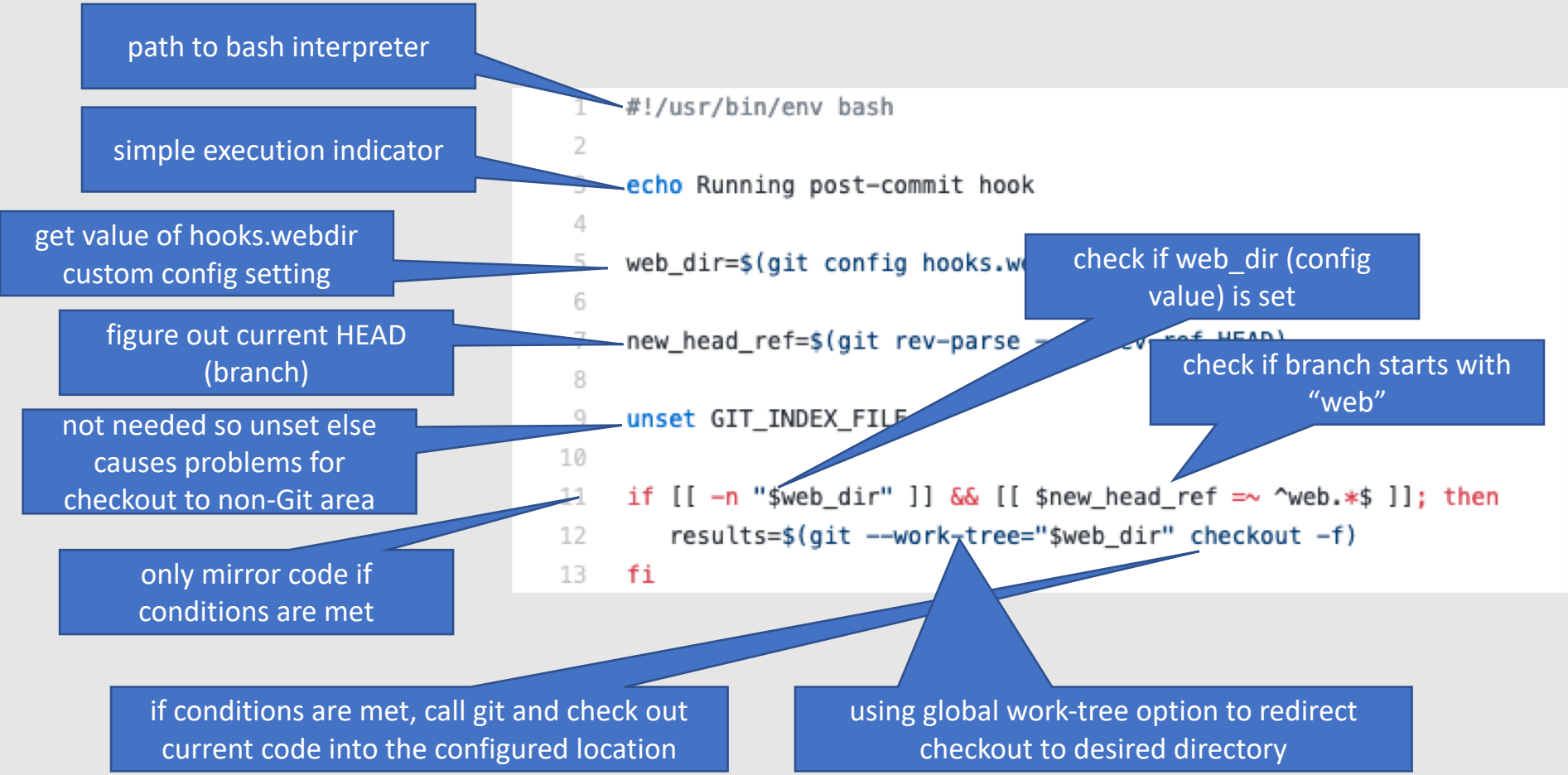
Hooks – merge

Git Operation	Pre-operation Hook 1	Pre-operation Hook 2	During-operation Hook	Post-operation Hook 1	Post-operation Hook 2
merge, pull				post-merge	

- post-merge – runs after a successful pull or merge (or after clone unless no-checkout option is used)
  - P1: flag that indicates squash or merge
- can be used to do things like set permissions or kick off supporting processes (such as testing)

A simple post-commit hook

- Intent: Mirror copies of files when committed to local repo if branch starts with “web”.
- Configure hooks.webdir as location to mirror to

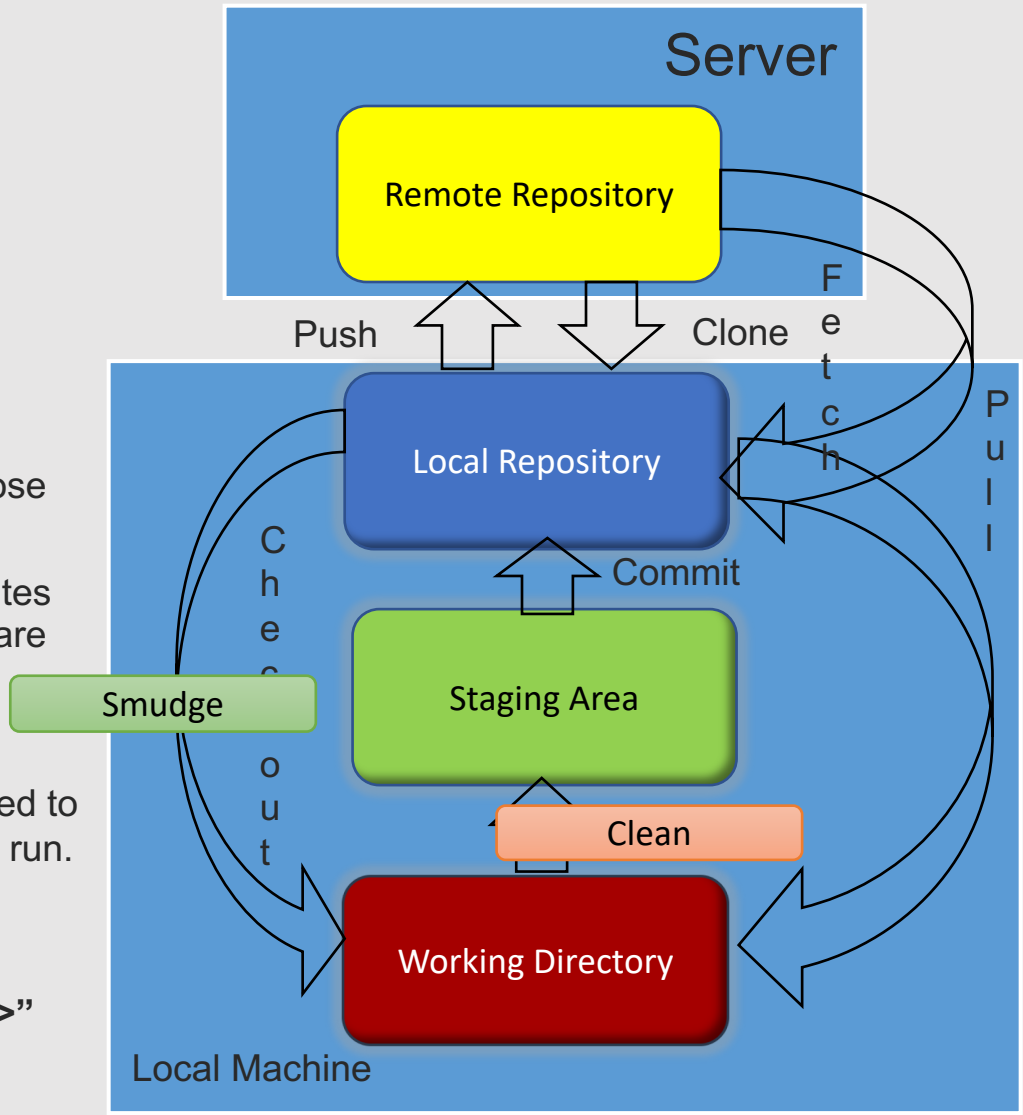


Lab 14 - Creating a post-commit hook

Purpose: In this lab, we'll see how to put a post-commit hook in place to be active in our local Git environment.

Custom Filters

- Defining
  - attribute: filter=name
  - In .gitattributes
    - **“file pattern filter=name”**
- Provides two filtering actions
  - smudge
  - clean
- **smudge** = “For the matching items in the Git attributes file, run a set of code (a filter) when those items are checked out of Git.”
- **clean** = “For the matching items in the Git attributes file, run a set of code (a filter) when those items are committed back into Git.”
- Git assumes that name provided is “filter driver”.
- Filter driver = program or set of commands defined to execute for smudge and clean when their filter is run.
- Mapping of filter name to actual code is done via standard “git config” operations.
  - **git config filter.<name>.smudge “<code>”**
  - **git config filter.<name>.clean “<code>”**



(Optional) Lab 15 - Using a Git Attributes File to Create a Custom Filter

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

That's all - thanks!

techskillstransformations.com
getskillsnow.com