# A Practical Understanding of Git

## Common commands and when to use them

Joseph Bertaux

Purdue University

November 18, 2024

# Contents

1. Overview
2. Commands
    - Getting started
    - Reviewing history, previewing changes
    - Committing changes
    - Merging changes
3. Workflow
4. References

# Contents

1. Overview
2. Commands
   - Getting started
   - Reviewing history, previewing changes
   - Committing changes
   - Merging changes
3. Workflow
4. References

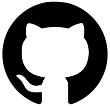
# What is Git?

Figure: Git logo. [1]

- Git is a command-line tool for version control
- It saves a project at various stages using `commits` and `branches`
- It can be used to `push` changes to remote repositories
- And naturally `pull` (or rather `fetch`, `merge` from other repositories)
- Also, generate `diff` files with differences between different files or different versions of the same file

# What Git is not

- Git is not a specific host for `.git` style repositories
- Namely, Git itself is not one of:



Github [2]

Sourceforge [3]

Sourceforge [4]

# Some History

- In 2005, Linus Torvalds created Git over the span of approximately 5 days
  - The same Linus Torvalds that created the Linux kernel
- Linus did this to create a replacement for BitKeeper
  - This was motivated by a licensing dispute
  - BitKeeper was a popular version control software before Git [6]
- The sPHENIX Wiki also provides external links, specifically [5]

# What to expect from this presentation

- This presentation aims to leave you with an idea of what sequence of commands to use in a basic Git workflow
- For all Git commands, please know that you can do
    - `git help <command>`
- To bring up the manual page for that command
    - This allows you to see additional options, which could be useful
    - It gives a proper description of what the command does
- This talk will not discuss advanced Git techniques,
- nor will it discuss Git internals

# Contents

# Getting Started

- You should configure your Git identity before beginning to work with repositories
- Once you have a fork or subdirectory ready to work in, you'll want to configure that subdirectory
- If you've forked a repository, you'll want to configure your remote upstream to point there
- Commands may find yourself using are
  - `git config`
  - `git clone`
  - `git init`
  - `git remote`

# Getting Started

## Configuring Git

- You should configure your Git identity before beginning to work with repositories
- On a filesystem where you have your own user account, you can run
  - `git config --global user.name <your username>`
  - `git config --global user.email <your email>`
  - to update the copy of `.gitconfig` in your home directory
- If instead you're using a user account which is shared with others, then for the local repositories you are maintaining,
  - `git config --local user.name <your username>`
  - `git config --local user.email <your email>`
  - To update the local `.git/config` file at the top level of that repository
  - Note that `--local` can be omitted–this is the default behavior of `git config`
  - (This is the case if you want to use Git on `opc0`)
- This ensures your commits are credited to you
  - For better or worse–but we won't discuss `git blame` here

- You may also want to change the editor Git will use by default for interactive commands
  - `git config --global core.editor <your editor>`
  - The default is nano, but you may wish to change it to vim or emacs, for example
  - You may need to omit `--global` and do this on a per-repository basis for filesystems where you're sharing a user account with other people
- This is useful for `git commit`
  - You can run `git commit` without specifying `-m` to launch your editor to create the commit message
  - You will also see which files will be modified by the commit as commented lines
- and `git rebase -i`
  - You edit a series of files here
  - first selecting which commits to `pick` and `squash`
  - then revising the messages of `pick`'d commits

# Getting Started

## Initializing a Repository

- When beginning work with Git, you need to either obtain an existing repository or create a new one
- The ways to do this are with
    - `git clone <url>`
        - To copy a remote repository here and set the remote origin to `url`
        - Note that this creates a subdirectory for the repository
    - `git init`
        - To initialize a local repository with Git
        - This won't affect existing files, and you can do this with non-empty directories

# Getting Started

## Configuring a Repository

- You can run the aforementioned `git config` commands withouth specifying `--global` to apply them only for this repository
- If you a working with a fork of a repository, you should also run
  - `git remote add upstream <upstream url>`
  - This should be the url of the repository you forked (not the url of your fork)
- You should run `git remote -v` to check the remote references are as they should be
  - `origin` is the url of your fork
  - `upstream` is the url of the original repository you forked from
- If you started with `git clone`, your remote origin should be configured by default

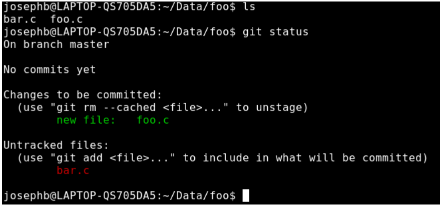
# Reviewing history, previewing changes

- One of the more frequent things to do is preview your changes from the command line
  - This helps you avoid too frequent, premature commits by checking the changes you made will be incorporated the way you intend
  - But performing `git rebase` is the most powerful to keep a clean history–more on this later
- Though some remote `.git` repositories offer ways to view changes,
  - This requires committing your working tree and pushing the changes to a remote host
  - And running a separate application to view the remote changes after they've been pushed (e.g., your browser)
- Commands you may find yourself using often are
  - `git status`
  - `git ls-files`
  - `git log`
  - `git diff`

# Reviewing history, previewing changes

## `git status`, `git ls-files`

- `git status` shows
  - what tracked files have been modified
  - which files are not tracked (and must be `git add`'d before changes can be `git commit`'d



```
josephb@LAPTOP-QS705DA5:~/Data/foo$ ls
bar.c  foo.c
josephb@LAPTOP-QS705DA5:~/Data/foo$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   foo.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        bar.c

josephb@LAPTOP-QS705DA5:~/Data/foo$ 
```

Figure: Example output of `git status` where `foo.c` has been `git add`'d, but not yet `git commit`'d, and bar has not even been `git add`'d

- `git ls-files` shows
  - A list of which files are being tracked by Git
- Notice that these can be given a path, if you want to see the output for a specific subdirectory or even only check one file

# Reviewing history, previewing changes

## `git diff`, `git log`

- `git diff <options> <commit> <paths...>` shows
  - The difference between the state of `<paths...>` as they are on disk and as they are as of `<commit>`
  - By default, `<commit>` is the `HEAD` of the current working tree
  - So running `git diff <paths...>` shows any “unsaved” changes made to `<paths...>` since the last time you ran `git commit <...>` or `git add ...`
- But sometimes you want to see the changes you’ve over the past several commits, and not just unsaved changes
- For this we can use `git log` to see how many commits back we want to check, or which commit we want to check against
- `git log <paths...>` shows
  - The Git commit history of `<paths...>`, with commit messages, dates, and authors

# Reviewing history, previewing changes

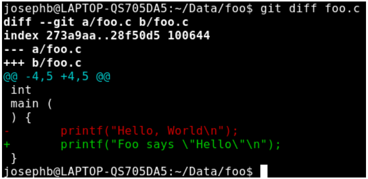
## More on `git diff`

- `git diff`, while knowing what commits we want to compare, is a useful tool
- A particularly useful syntax is `git diff HEAD~<#> <paths...>`, where <#> is the number of previous commits to compare the working tree with
  - For example, compare the working tree to what our code was 3 commits ago:
  - `git diff HEAD~3`
- Or, use `git log` and obtain the hash of a particular commit for comparison
- It can also be used to compare any files
  - `git diff --no-index <file1> <file2>`
  - This is the default behavior of `git diff` if neither file is being tracked by Git

# Reviewing history, previewing changes

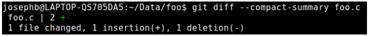
## More on `git diff`

- Try `--compact-summary` if working with many files



```
josephb@LAPTOP-QS705DA5:~/Data/foo$ git diff foo.c
diff --git a/foo.c b/foo.c
index 273a9aa..28f50d5 100644
--- a/foo.c
+++ b/foo.c
@@ -4,5 +4,5 @@
 int
 main (
 ) {
-       printf("Hello, World\n");
+       printf("Foo says \"Hello\"\n");
 }
josephb@LAPTOP-QS705DA5:~/Data/foo$
```

Typical `git diff` output



```
josephb@LAPTOP-QS705DA5:~/Data/foo$ git diff --compact-summary foo.c
 foo.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Output with `--compact-summary`

# Committing changes

## Useful practices

- Before starting new work, you'll want to synchronize your local repositories with the upstream repository
  - You only need to synchronize the Git histories (`git fetch`)
  - But you can also apply the latest changes if you want to check the state of the code (`git merge`, `git pull`)
- When making changes, it's good create `branches` for each feature
  - This allows you to work on multiple features at a time while maintaining only one fork
  - It allows independent changes to be tracked and merged (or discarded) independently
- Commands you may find yourself using often are

- `git checkout`
- `git fetch`, `git merge`, `git pull`
- `git push`

- `git add`, `git commit`
- `git rm`, `git restore`
- `git rebase`

# Committing changes

## Checking out a new branch

- To check out a new branch, you can run either
  - `git checkout -b <branch name>`
  - `git branch <branch name>`
- These give you a new branch that is synchronized with your local `master` branch
  - It may be instead called `main`
  - This is the default name for more recent Git repositories
- You'll then want to obtain remote changes and apply them to your working branch,
  - `git fetch upstream/master; git merge upstream/master`, or
  - `git pull upstream/master`
- Note that it may be `upstream/main` instead
- `git fetch` synchronizes the Git history
- `git pull` and `git merge` synchronize the history and state of tracked files

# Committing changes

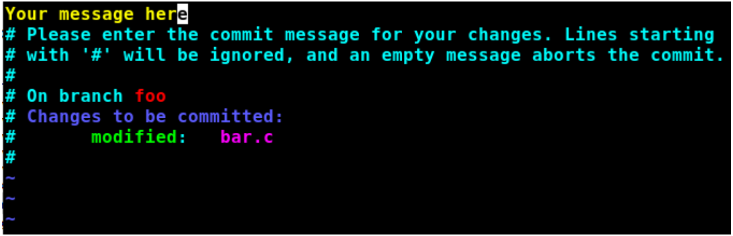
## Making changes

- At this point
  - (you are on your working branch and have synchronized it with the remote)
- You can edit files locally into the state they should be
  - Making changes using your preferred workflow and text editor
  - Add additional files or entire subdirectories using `git add <path...>`
  - Removing files that have become superfluous using `git rm <path...>` (this removes them on disk also)
- You can do this over multiple sessions
  - But you will need to `commit` you changes before switching branches
  - Creating superfluous commits is fine with `rebase`, so this is the way I'd advise
  - You can also `stash` your changes and then `apply` them later
  - But this makes them easy to loose and there is only one `stash` at any time

# Committing changes

## Committing changes

- Commit all changes `git commit -a -m <commit message>`
- You can also run `git commit -a` and then edit the message interactively
  - You will use your Git editor for this as described earlier, which is why it is important to set it



```
Your message here
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch foo
# Changes to be committed:
#       modified:   bar.c
#
~
~
~
```

Figure: Example of an interactive commit. Notice the information about what will be committed in the commented lines. Note that my Git editor has been changed to vim.

# Committing changes

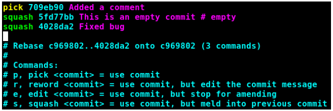
## Rebasing changes

- Once you've committed changes, you can **OPTIONALLY** rebase your commit messages
- You can run `git log` to check which commit you should rebase from
    - This will usually be the commit when you `merge`'d the latest state of upstream you one of your local branches
- Once you've identified the commit, you can run
    - `git rebase -i <commit>`, or
    - `git rebase -i HEAD∼<#>`
- This will launch an interactive (`-i`) rebasing session where you can `squash` superfluous commits
    - You will use your Git editor for this as described earlier, which is why it is important to set it
- **ONLY REBASE YOUR OWN COMMITS**
    - Intermediate commits are lost
    - Don't push changes if you have modify the git history of others' work

# Committing changes

## Rebasing changes

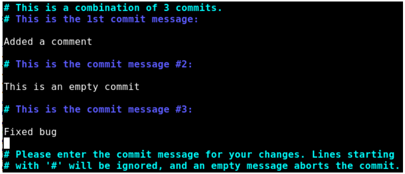
- What you will see will depend on your editor and commit history
- Note that newer commits are listed lower in the file
    - (Opposite to `git log`, where newer commits are shown toward the top)



```
pick 709eb90 Added a comment
squash 5fd77bb This is an empty commit # empty
squash 4028da2 Fixed bug

# Rebase c969802..4028da2 onto c969802 (3 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
```

Figure: Selecting which commits to pick and squash, saving and closing this file in your editor takes you to the next step



```
# This is a combination of 3 commits.
# This is the 1st commit message:

Added a comment

# This is the commit message #2:

This is an empty commit

# This is the commit message #3:

Fixed bug

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

Figure: Editing the messages of the commits you picked in the previous step, saving and closing this file in your editor finishes the rebase

# Committing changes

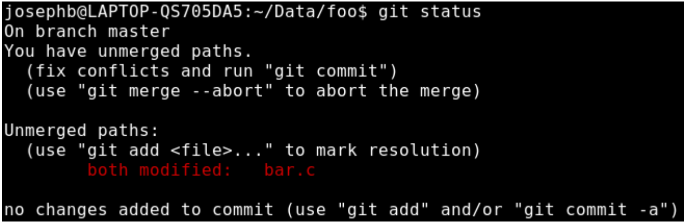
## Merging changes

- You should verify the changes you've made can be merged
  - (The upstream branch may have changed while you were editing your feature branch)
- A way to do this is
  - Checkout your local master branch (`git checkout master`)
  - Merge your local feature branch (`git merge <feature branch>`)
  - Merge the upstream master branch (`git merge upstream/master`)
- Note that
  - This is not the only way to achieve this
  - The merges can be performed in either order
  - The master branch may not be called `master`, but something else (e.g., `main`)
- You might encounter merge conflicts when trying to `git merge` or `git pull` branches involving incompatible changes to the same file

# Committing changes

## Resolving merge conflics

- These occur when multiple branches are `merge`'d, but
  - Branches specify different changes to the same file(s)
  - Git cannot resolve how to change the file (the selected `diff` algorithm failed)
- You can check the status of a merge by running `git status`



```
josephb@LAPTOP-QS705DA5:~/Data/foo$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   bar.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Figure: Output of `git status` when a merge conflict exists. Note that is says `both modified` for conflicting files, and give instructions on how to proceed

# Committing changes

## Resolving merge conflics

- Conflicting sections of conflicting files are modified with sections like

```
<<<<<<< HEAD
 Section of code as it exists in the branch you are merging into
 (The target branch that will be changed by the merge)
=======
 Section of code as it exists in the branch you are merging into
 (The source branch that will be unchanged by the merge)
>>>>>>> <name of source branch>
```

- You can traverse files by searching for the literals <<<<<<<, =======, or >>>>>>> which Git inserts into the file
- The simplest way to resolve the merge is to keep the sections from one branch
- But you will need to incorporate features added by other contributors with the features you are trying to add

# Committing changes

## Pushing changes

- Once you're satisfied with your changes, you can run
  - `git push`
- To push your the commits of your local feature branch to your `origin`
- If you have not done this for the first time with this branch, you may need to run
  - `git push --set-upstream origin <feature>`
  - This will create a new branch at your `origin` to receive changes from your local feature branch
- If you have done this already, you may need to run
  - `git push -f`
  - `git push --force`
  - This will force push the commits you've made
  - And will be necessary if you've `rebase`'d your changes since your last `push`

# Contents

1 Overview

2 Commands
- Getting started
- Reviewing history, previewing changes
- Committing changes
- Merging changes

3 Workflow

4 References

# Worflow

## Starting out

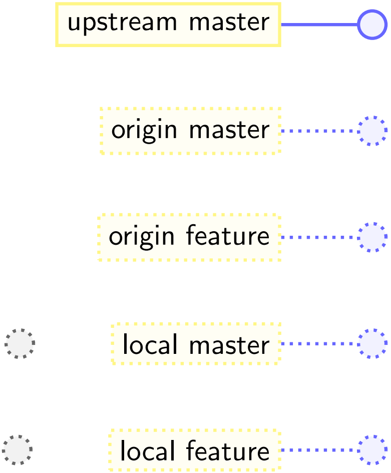


Figure: There is a `remote upstream` ("upstream master") repository that you want to track

# Worflow

## Forking

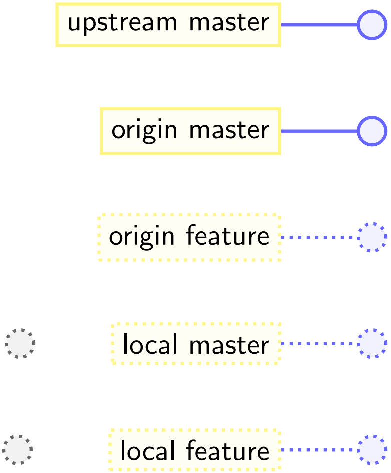


Figure: You fork this respository (e.g., on your Github), so there is a `remote origin` ("origin master") to track

# Worflow

## Forking

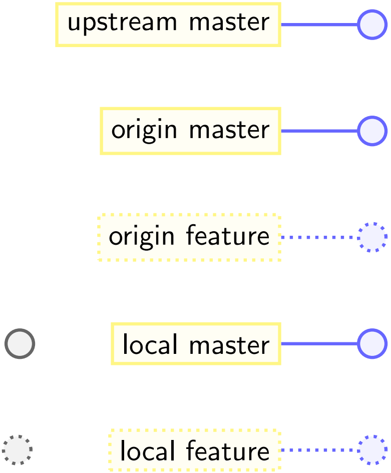


Figure: You clone your fork of the repository with `git clone <url>`

# Worflow

## Forking

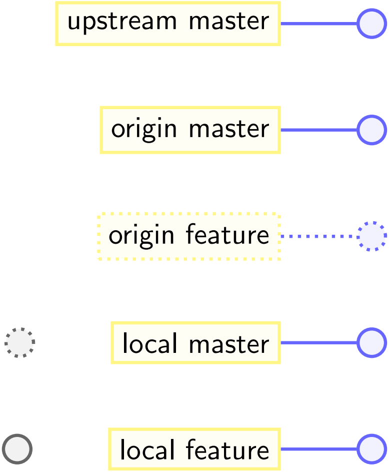


Figure: You make a branch to implement a feature with `git checkout -b` or `git branch`

# Worflow

## Editing and committing

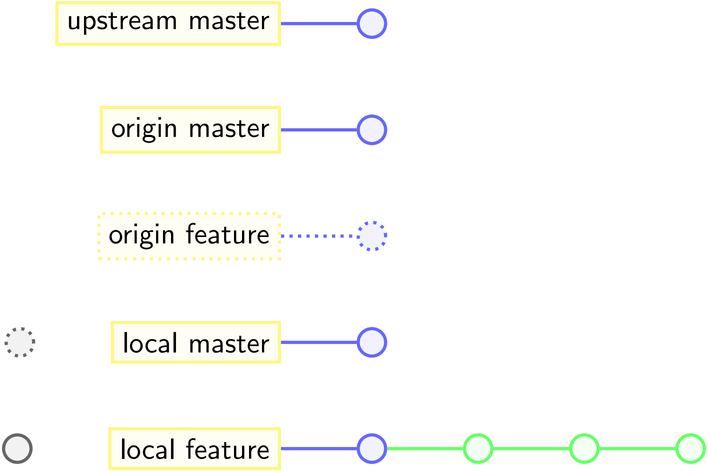


Figure: You make edits to the working tree and `commit` them as you go

# Worflow

## Rebasing

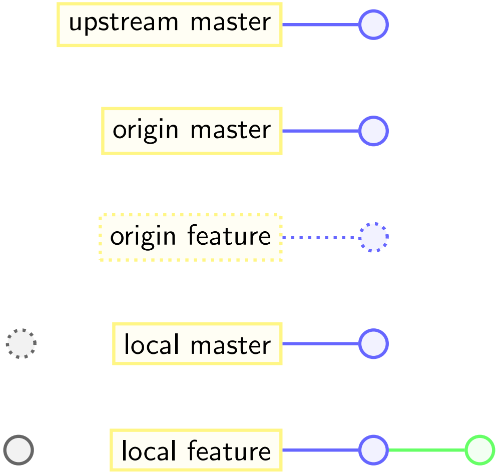


Figure: You (optionally) `rebase` your commits into a single commit

# Worflow

## Fetching from upstream

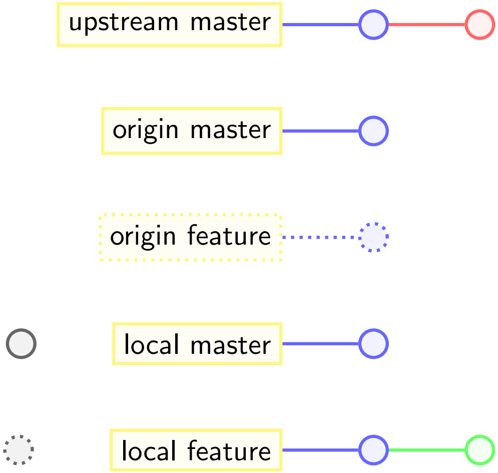


Figure: You `checkout` your local master branch to fetch upstream changes

# Worflow

## Fetching from upstream

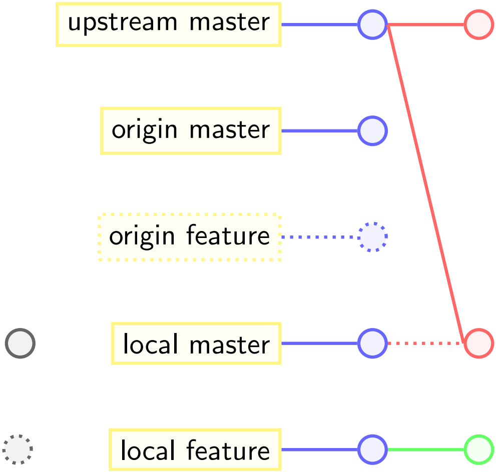


Figure: You (`fetch` and `fetch`) or `pull` from your upstream master into your local master

# Worflow

## Merging feature changes

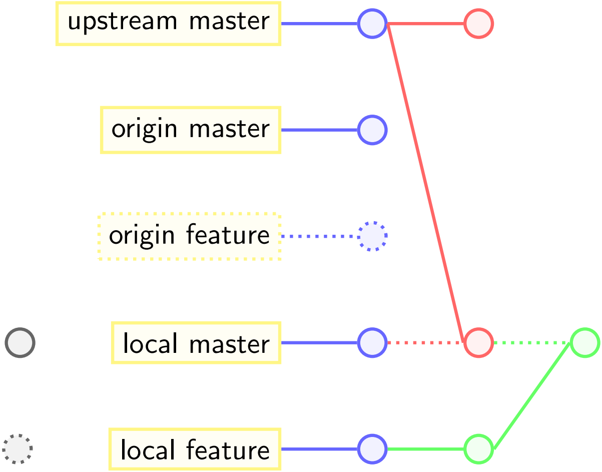


Figure: You `merge` your feature branch to your local master

# Worflow

## Pushing feature changes

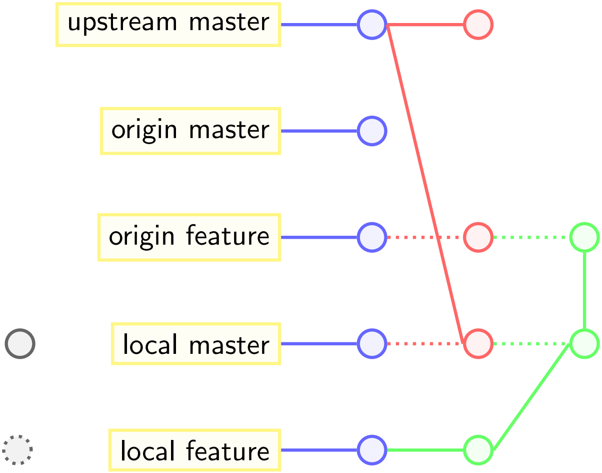


Figure: You `push` your feature branch to your origin as a feature branch

# Worflow

## Creating a pull request

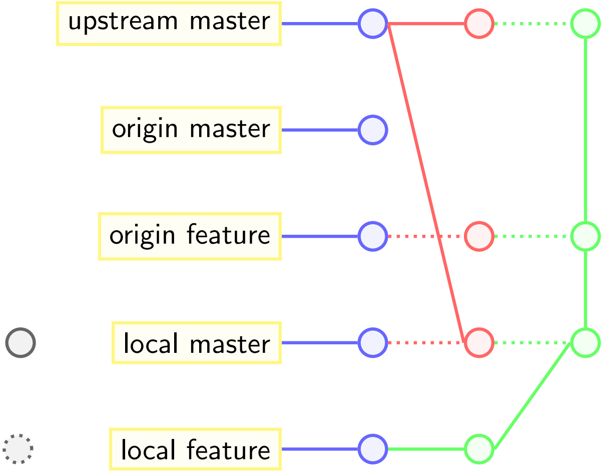


Figure: You create a pull request on your feature branch to the upstream branch

# Contents

1. Overview
2. Commands
   - Getting started
   - Reviewing history, previewing changes
   - Committing changes
   - Merging changes
3. Workflow
4. References

# References I

[1] https://commons.wikimedia.org/wiki/File:Git-logo.svg.

[2] https://en.m.wikipedia.org/wiki/GitHub#/media/File%3AGitHub_Invertocat_Logo.svg.

[3] https://en.m.wikipedia.org/wiki/File:SourceForge_logo_transparent.svg.

[4] https://en.m.wikipedia.org/wiki/File:GitLab_logo.svg.

[5] Chris Belyea.
A git workflow using rebase.
https://medium.com/singlestone/a-git-workflow-using-rebase-1b1210de83e5, April 2018.

# References II

[6] Kenneth DuMez.
Understanding git: The history and internals.
https://graphite.dev/blog/understanding-git, November 2023.