Advanced Git

trainologic

Overview

Git Basics (reminder)

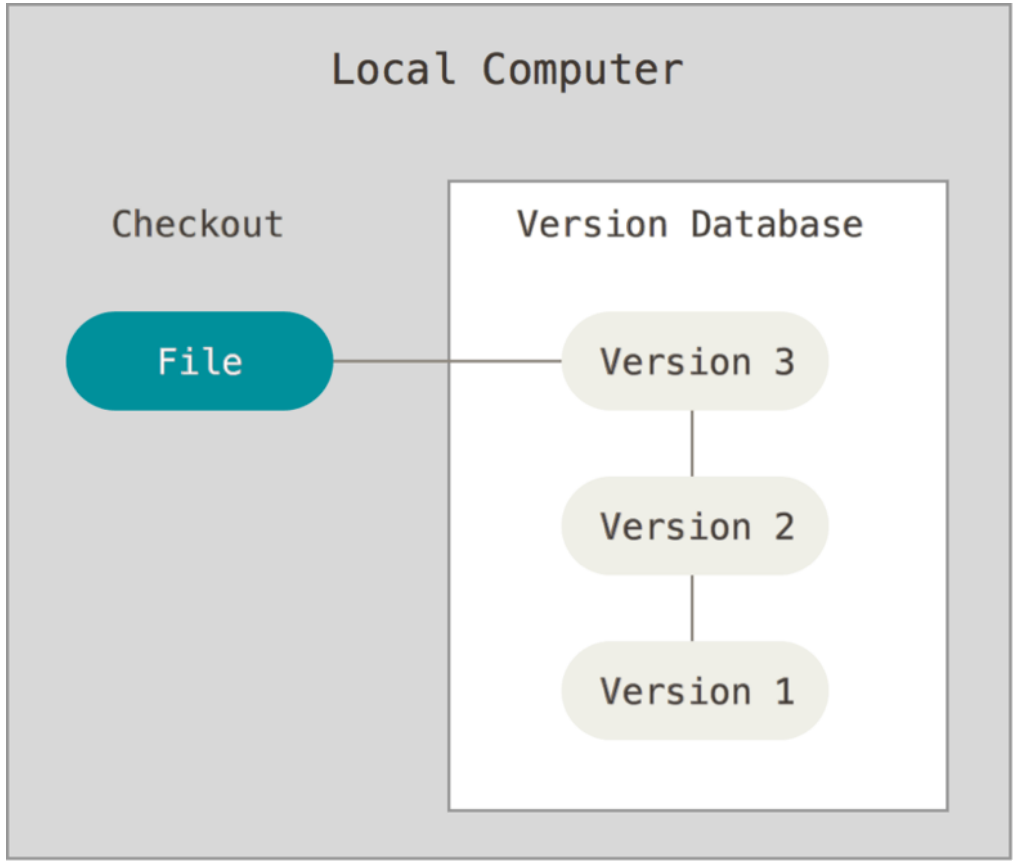
trainologic

What Is Version Control?

- Tracking changes in a set of files over time
- Ability to "recall" specific past versions in the future
- Compare changes over time
- Track who introduced what change and when (and why)
- Undo changes
- Revert files to specific past versions
- Generally – safety net for introducing changes

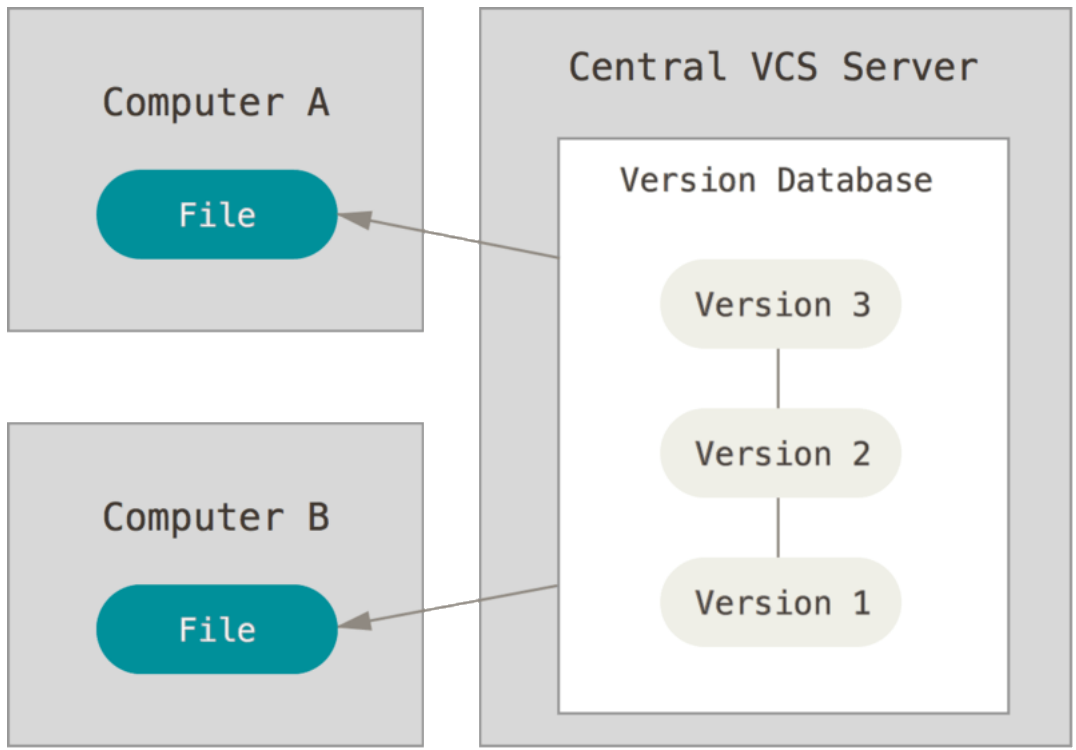
Version Control

- Local Version Control (VCS)



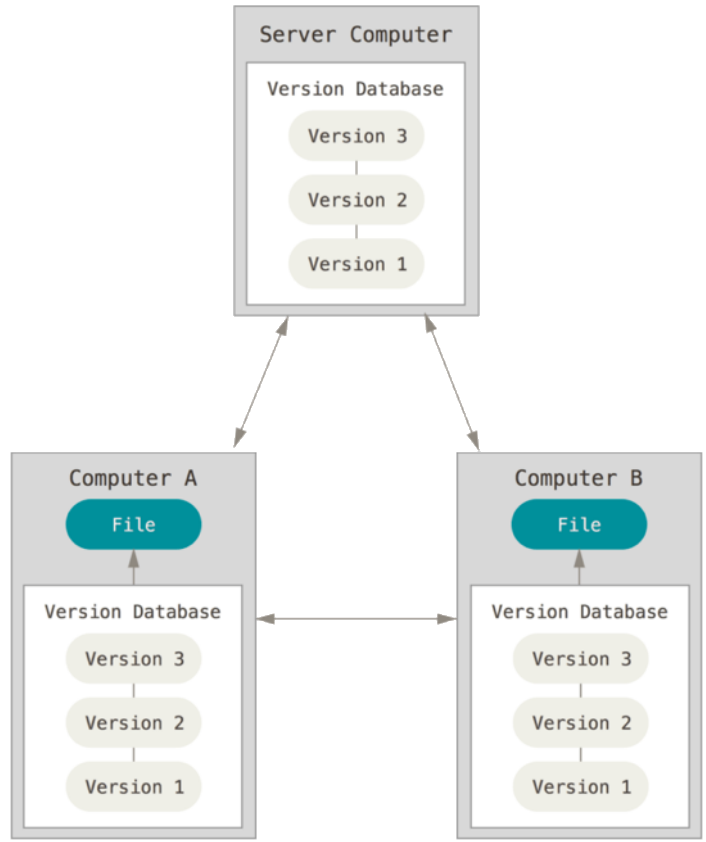
Version Control

- Centralized Version Control (CVCS)



Version Control

- Distributed Version Control (DVCS)

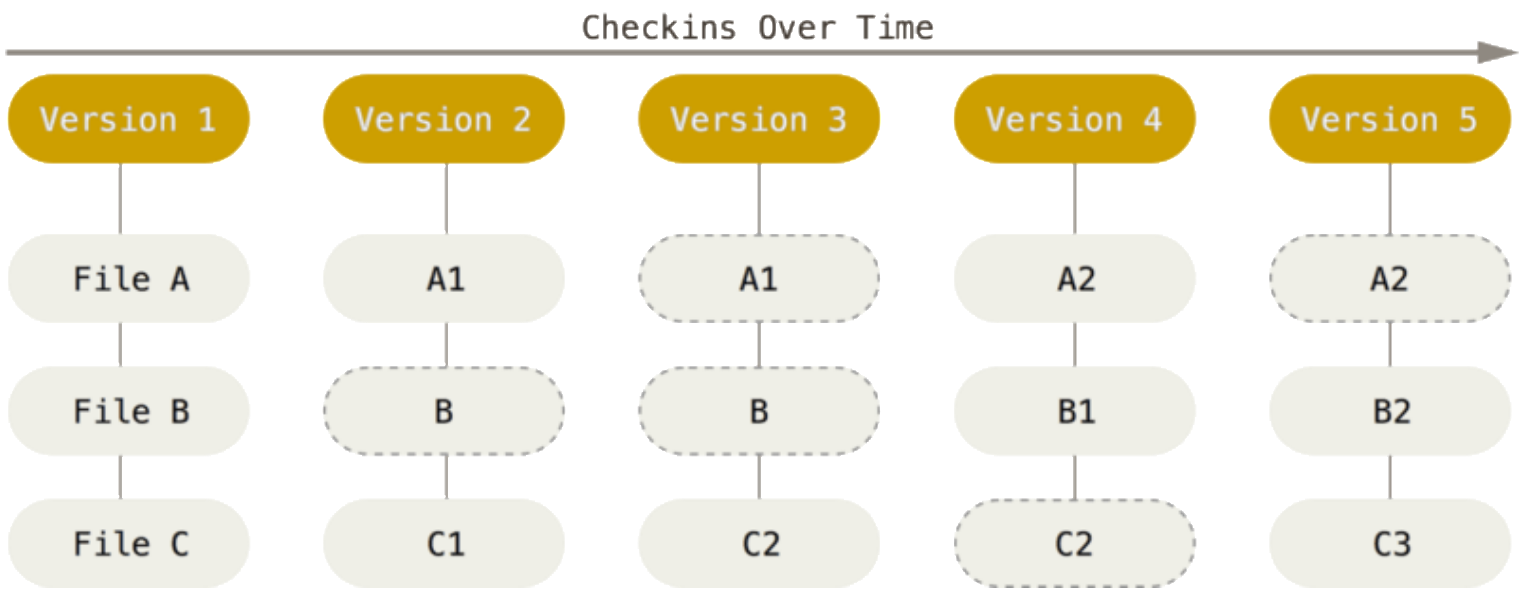


Git

- Git is a DVCS
- Created by Linus Torvalds (the creator of Linux) in 2005
- Original goal – serve the development of Linux Kernel
- Emphasizes speed, non-linear development, fully distributed, ability to handle large projects efficiently

Git

- Git thinks about its data like a “stream of snapshots” (as opposed to “series of deltas”)

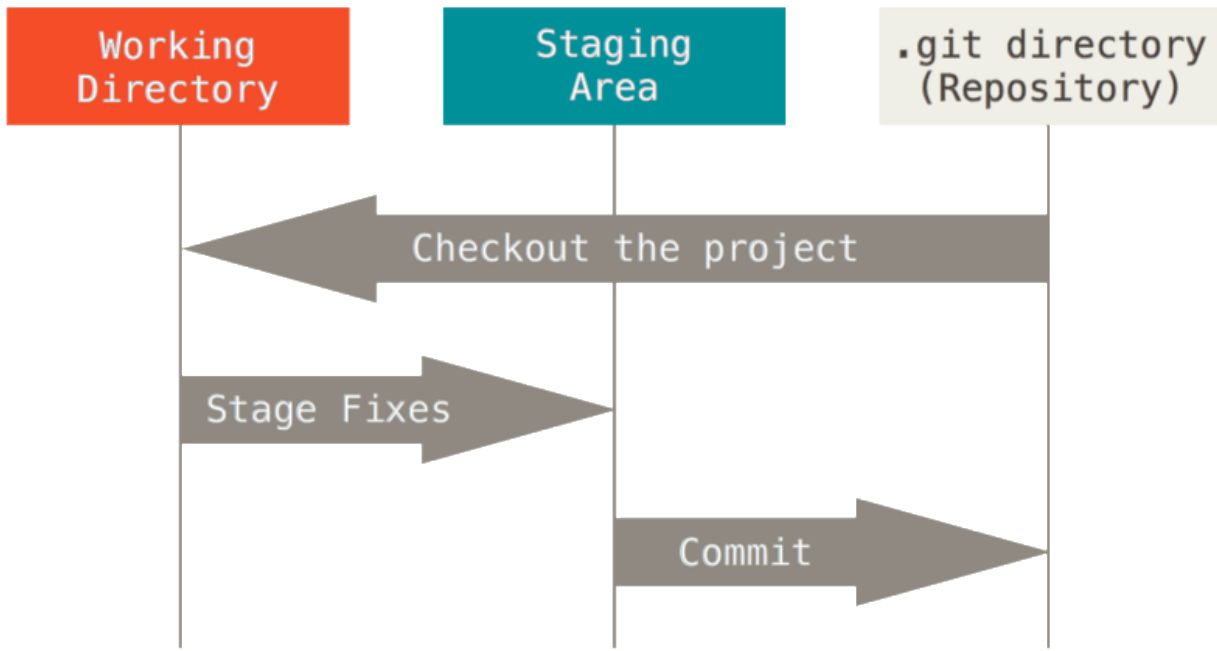


Git

- Most operations in Git are local, making it blazing fast
- Git has built-in integrity, based on SHA-1 hashes
- Git makes it hard to lose data that was committed

Git

- Git "thinks" about files in three states



Working With Git

- Git configuration controls how a git installation behaves
  - Per-repository config has highest priority (".git")
  - Global config takes next priority ("$HOME/.gitconfig")
  - System config has lowest priority ("/etc/gitconfig")
- Manage Git config from the command line
  - `git config [--global|--system] prop.name "Prop Value"`
  - `git config --list [--global|--system]`
  - `git config [--global|--system] prop.name`

Working With Git

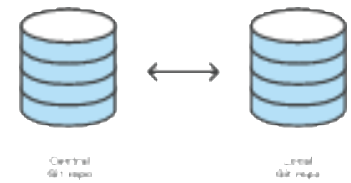
- Important config properties
  - Identity: user.name, user.email (used in every commit)
  - core.editor
  - core.eol (native, lf, crlf)
  - core.autocrlf (input, true)
- Many many more…
  - See `git help config`
  - Or https://git-scm.com/docs/git-config

Working With Git

- Getting help
  - Very detailed help (man pages)
    - `git help <verb>`
    - `man git-<verb>`
  - Quick & concise help
    - `git <verb> -h / --help`

Working With Git

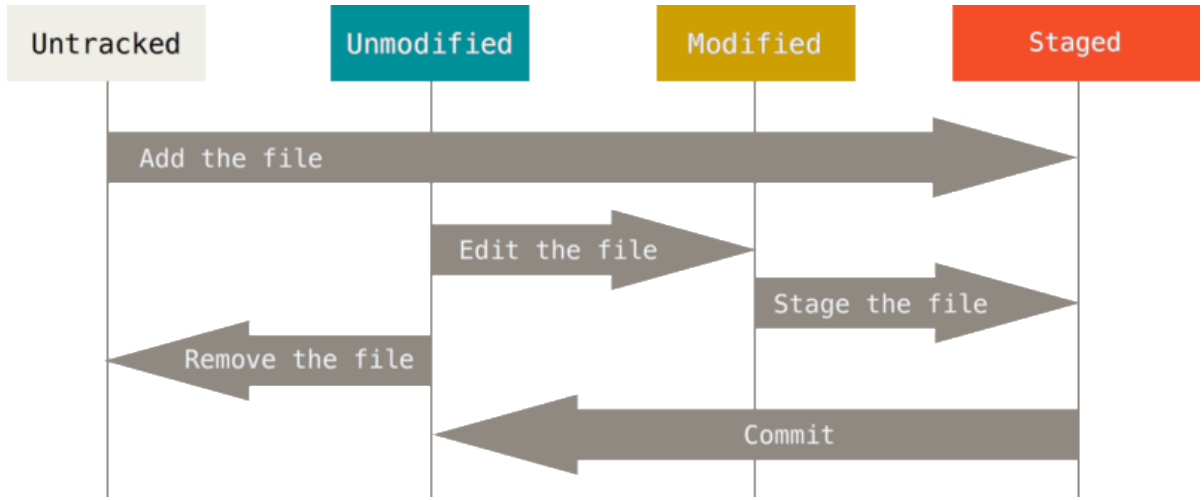
- Initialize a local Git repository in an existing directory
  - `git init`
- Download (clone) a remote repository
  - `git clone <remote-url> <local-dir>`



- Review workspace status & state
  - `git status`
- Show changes (working space, commits, branches)
  - `git diff [--cached]`
- Show history
  - `git log`

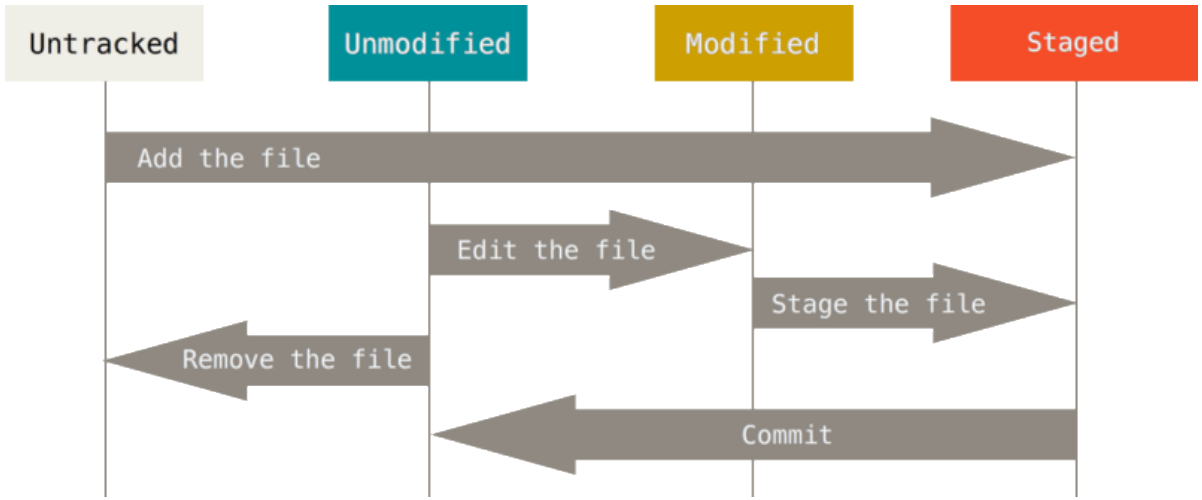
Working With Git

- Tracking new files, staging modified files
  - `git add <file> <file...>`



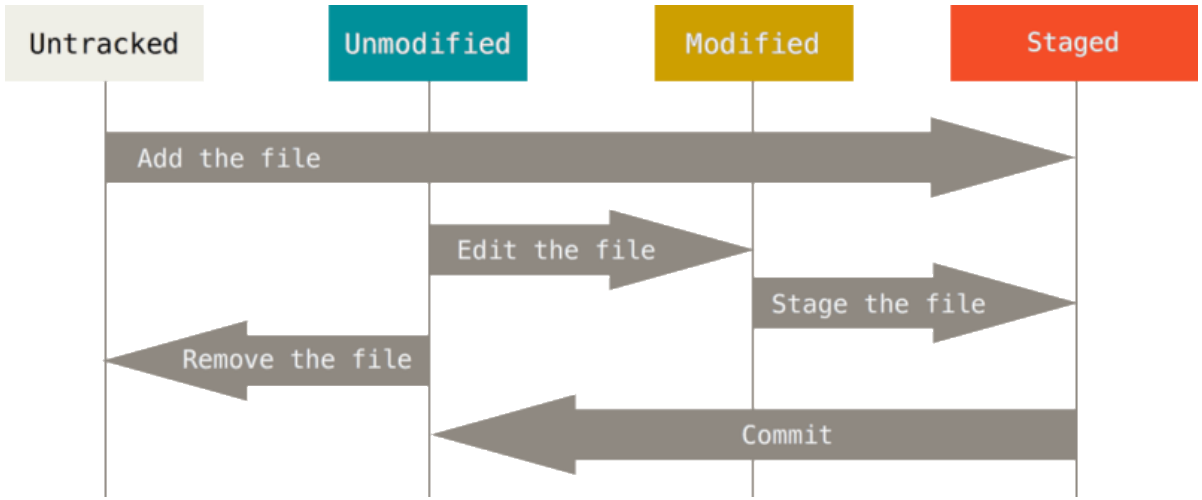
Working With Git

- Adding in Interactive Mode
  - `git add [-p|-i] <file> <file...>`



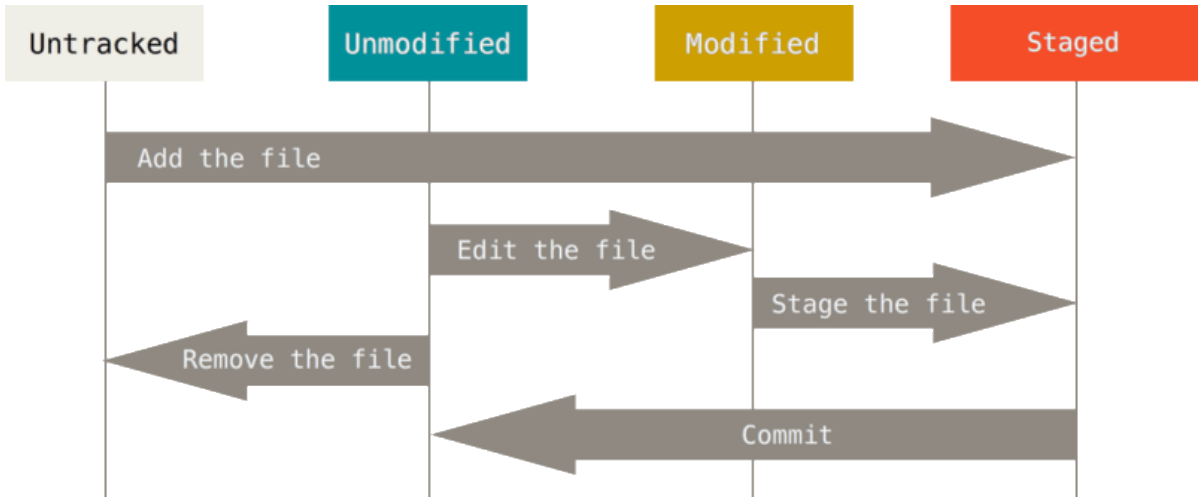
Working With Git

- Removing tracked files
  - `git rm [--cached] <file> <file...>`



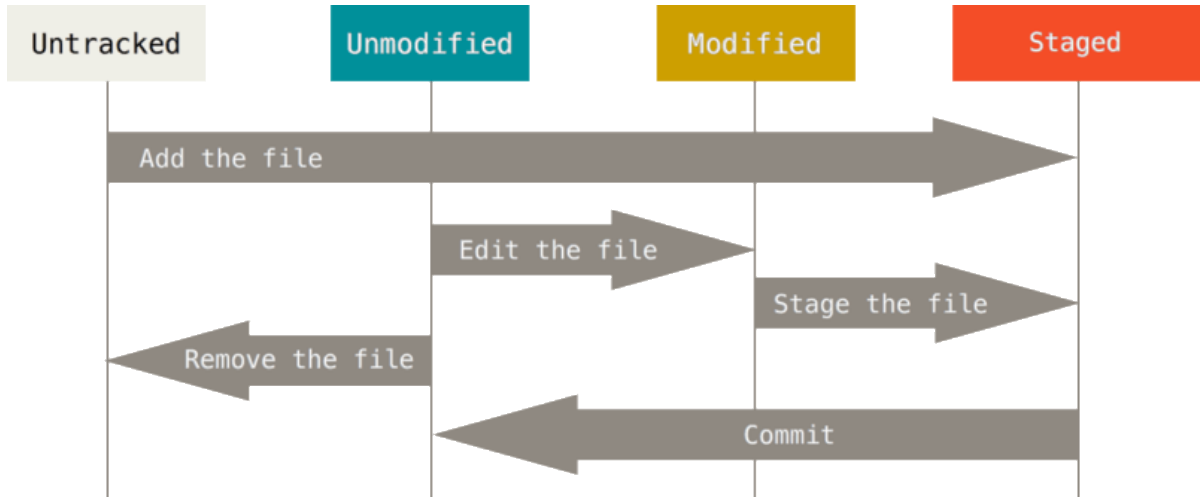
Working With Git

- Moving tracked files
  - `git mv <file> <file...>`



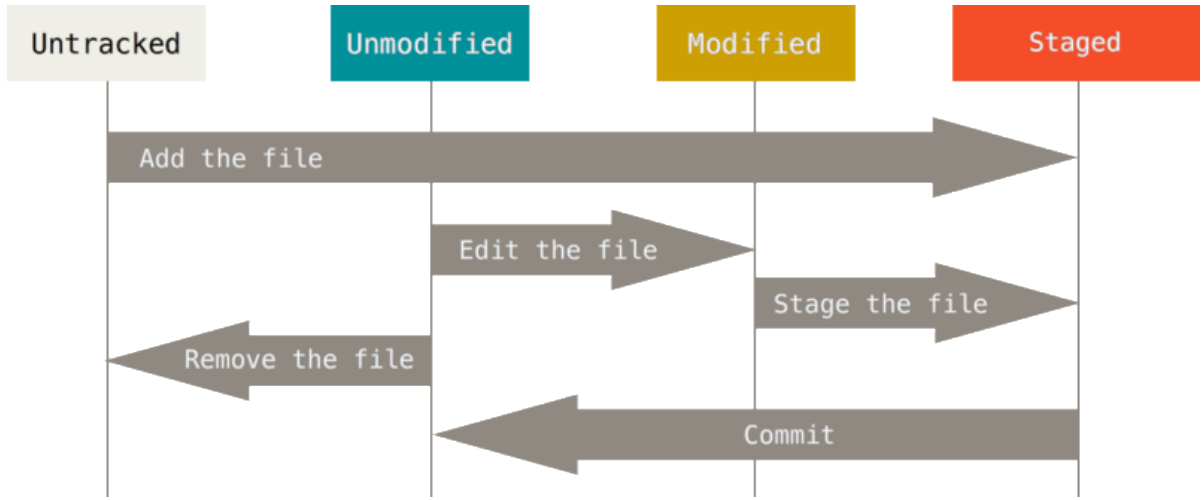
Working With Git

- Committing changes
  - `git commit [-a] [-m 'commit message']`



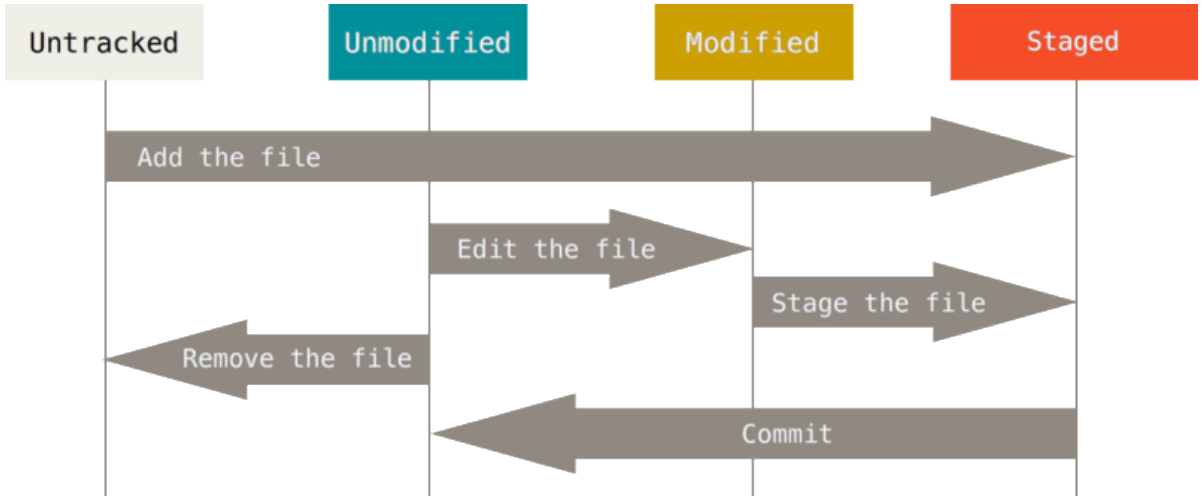
Working With Git

- Ignoring files
  - `echo "<pattern>" >> .gitignore`



Working With Git

- Stashing changes (maybe in Interactive Mode)
  - `git stash [-p]`
  - `git stash <pop|list|show>`

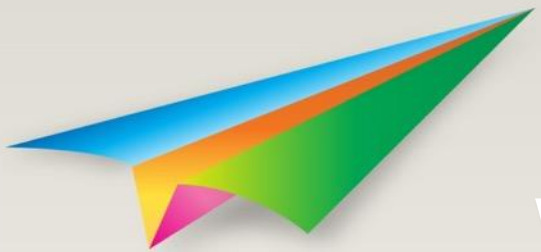


Working With Git

- Remote repositories can be accessed over SSH or HTTPS
  - Depends on the server set up
  - Using SSH requires SSH client and keys (possibly different sets of keys for different servers)
  - Using HTTPS may require entering credentials (username & password) when access is restricted
  - Many servers support both SSH & HTTPS

Working With Git

- Managing remotes
  - Cloning a remote repository automatically adds it as a remote named "origin" to the cloned repository
  - Showing remotes: `git remote [-v]`
  - Add remote: `git remote add <shortname> <url>`
- Syncing with remote
  - `git fetch <remote>`
  - `git pull <remote> <branch>`
  - `git push <remote> <branch>`

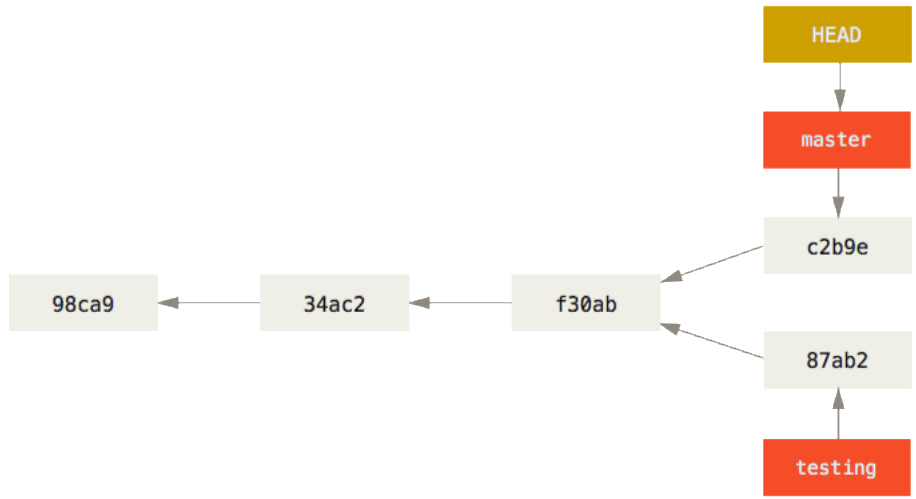


Branches
Workflows
Bitbucket & Gerrit

trainologic

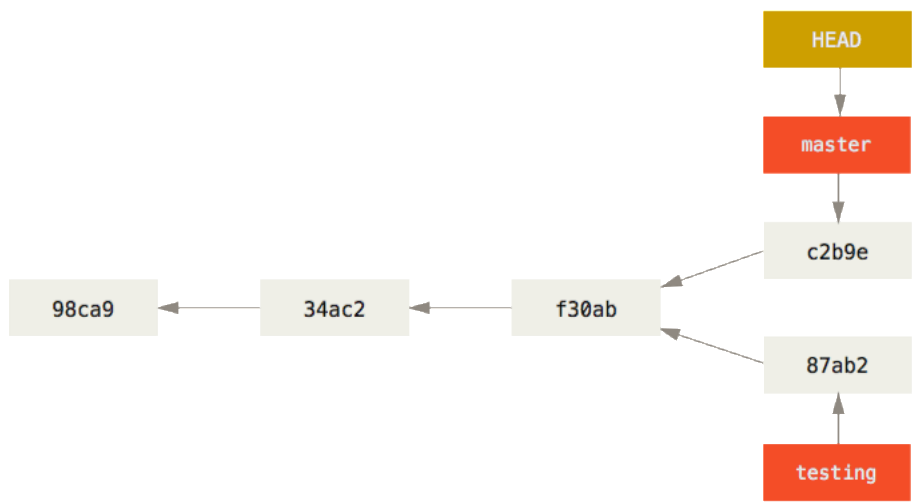
Branches

- Conceptually, in many VCS, ”branching” refers to the ability to diverge from the main line of development and merge back into it in a non-linear fashion



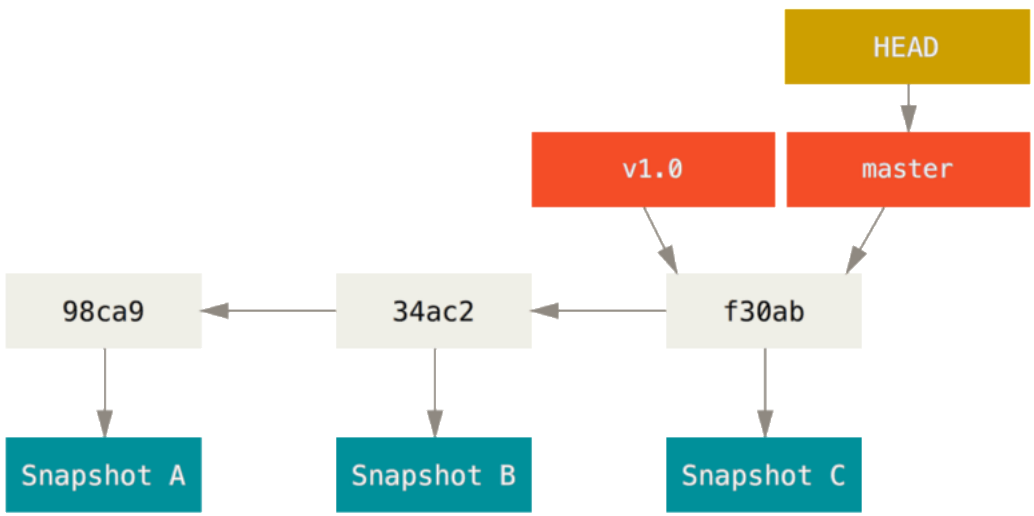
Branches

- The Git branching model is a killer feature
  - Lightweight
  - Fast
  - Local
  - Easy



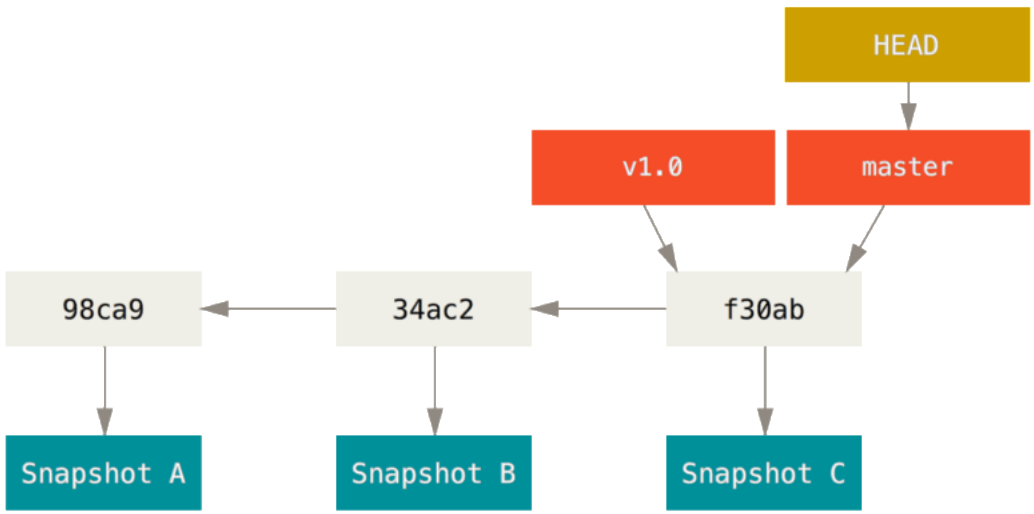
Branches

- A Git branch is simply a lightweight movable pointer to a commit object



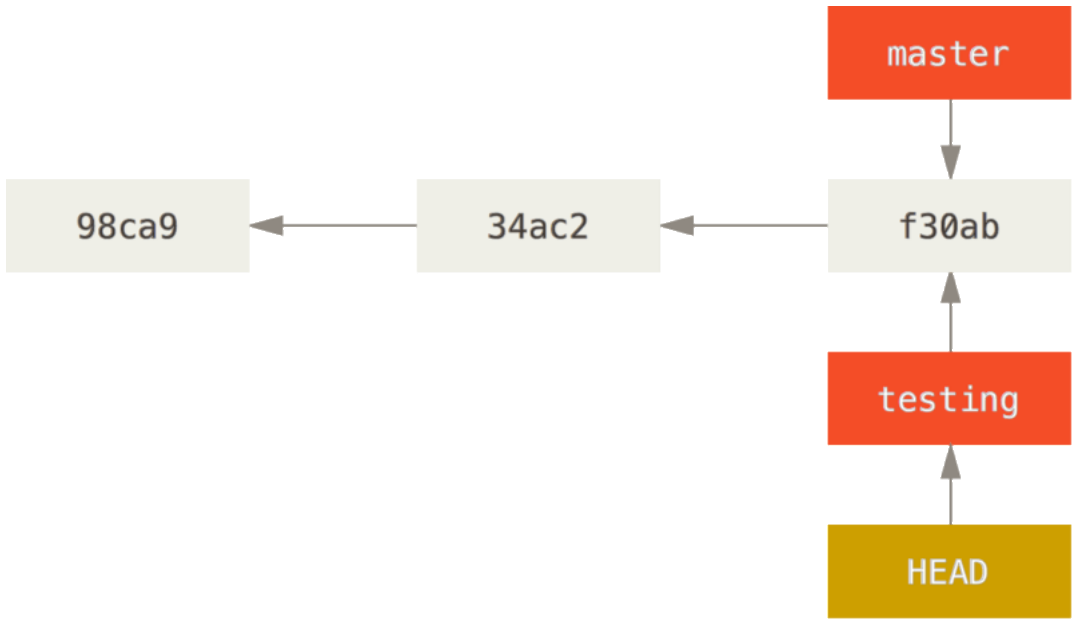
Branches

- Create a new branch based on the current commit:
  - `git branch <name>`



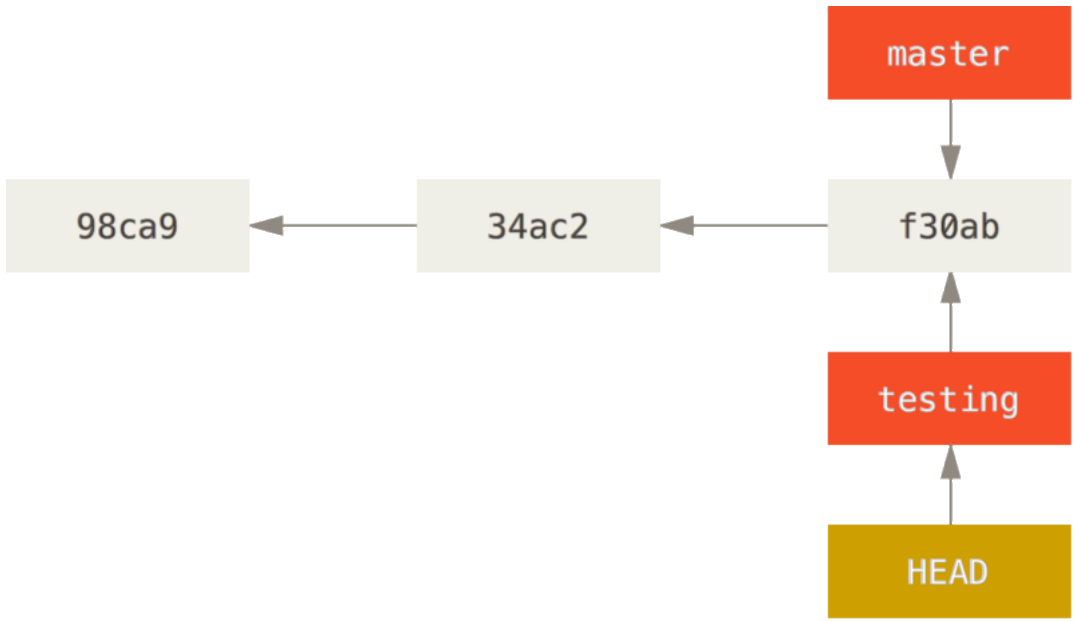
Branches

- Switch to an existing branch
  - `git checkout <name>`



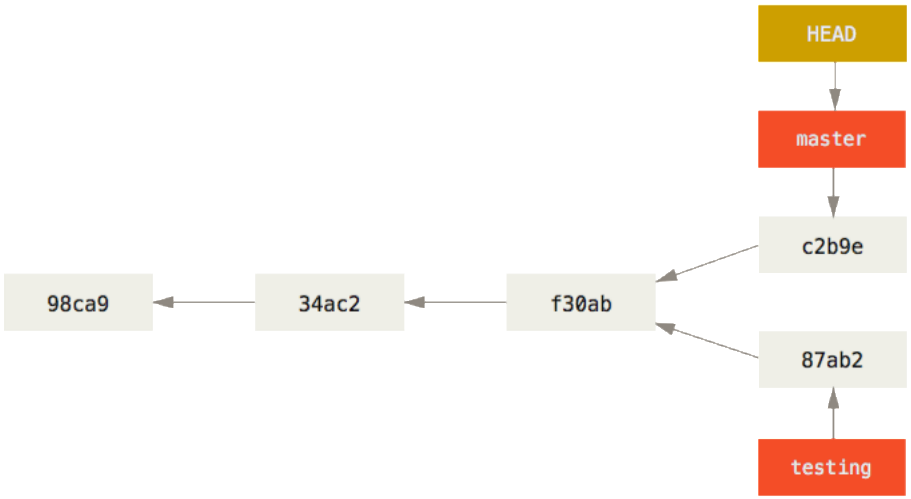
Branches

- Create branch and switch to it immediately
  - `git checkout -b <name>`



Branches

- Non-linear development with branches



Branches

- Branching tips & tricks
  - Create branch based on *another* branch / commit
    - `git branch <name> <source>`
  - Create branch based on *another* branch and switch to it
    - `git checkout -b <name> <source>`
  - List branches
    - `git branch [-v] [--no-merged|--merged]`
  - Delete branch (local)
    - `git branch <-d|-D> <name>`

Branches

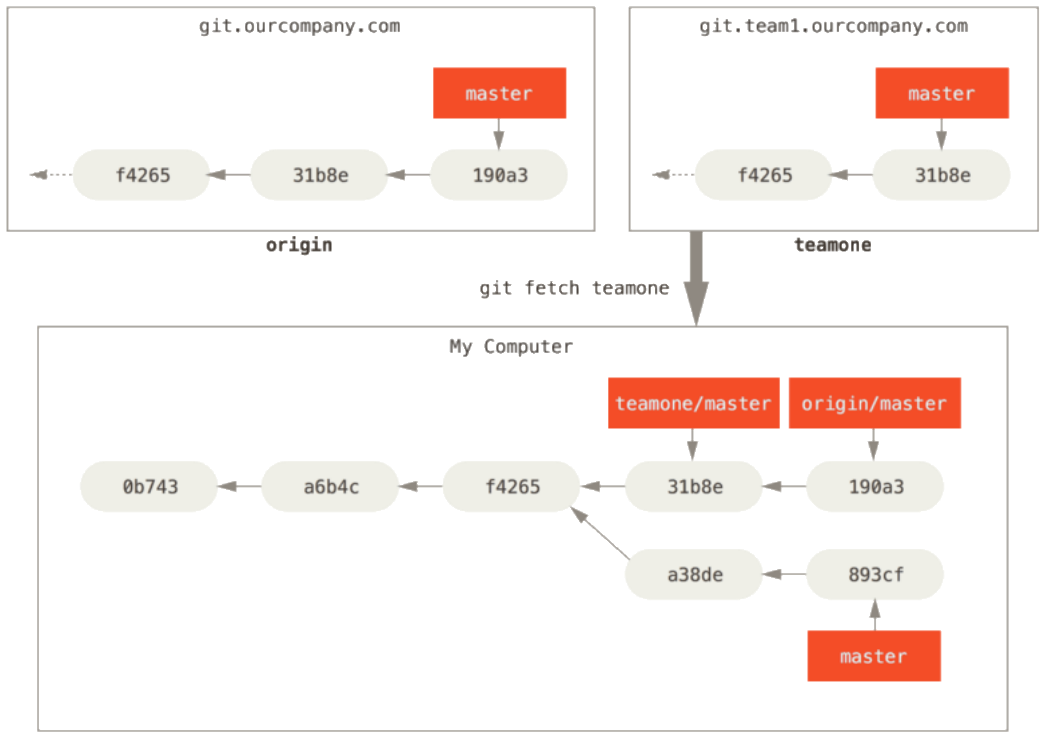
- Comparing branches
  - Compare current branch with “other” branch
    - `git diff <other>`
  - Compare branch “foo” with branch “bar”
    - `git diff <foo> <bar>`

Branches

- Pulling out a specific file from another branch
  - `git checkout <name> -- <file> <file...>`

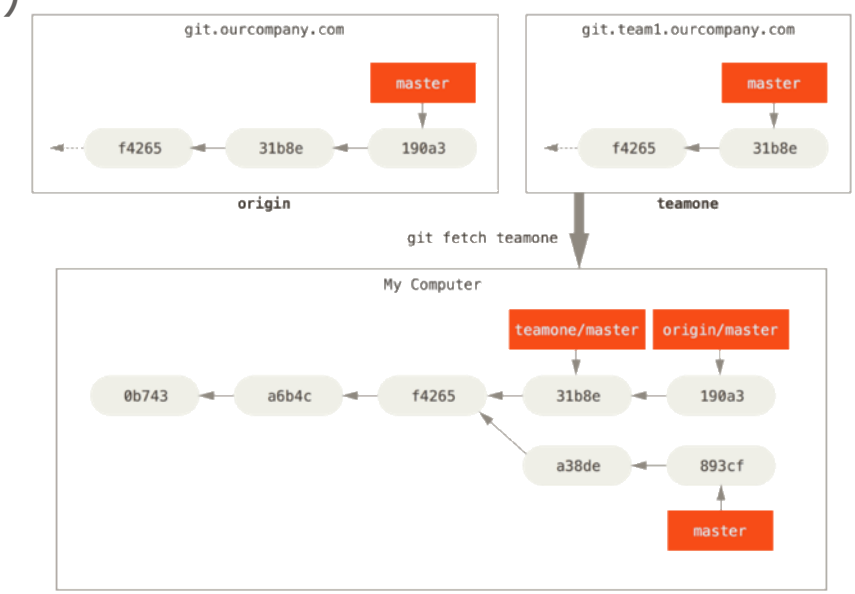
Branches

- Collaborating on branches



Branches

- Collaborating on branches
  - Remote-tracking branches are local references to the state of a remote branch - <remote>/<branch>
  - Git manages them on its own whenever you do network operations (fetch / pull / push)
  - They're like bookmarks



Branches

- Collaborating on branches
  - Publish a local branch to a remote
    - `git push <remote> <remote-branch-name>`
    - `git push <remote> <branch-name>:<remote-branch>`
  - Get a remote branch locally
    - `git fetch <remote> && git checkout <remote-branch>`
  - Sync new remote changes into locally checked out branch
    - `git pull <remote> <remote-branch-name>`
  - Deleting remote branches
    - `git push <remote> --delete <remote-branch-name>`

Git Workflows

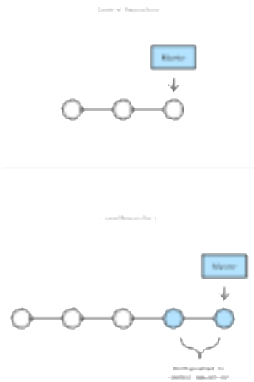
- What's a workflow?
  - There are many ways to use the branching capabilities
  - A workflow is a specific way to apply branching to a project
  - Teams choose the workflow that serves them best (and can change workflows if something else fits better)
  - Generally, Git workflows rely on the fact that branching is easy and leightweight, and simple 3-way merges between branches is usually easy to do
  - We want the workflow to enhance to effectiveness of the team and not be a burden that limits productivity

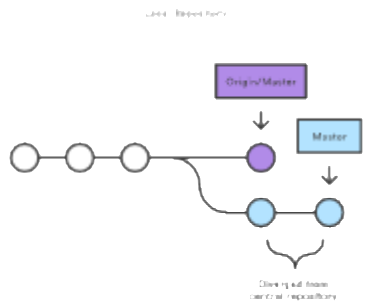
Git Workflows

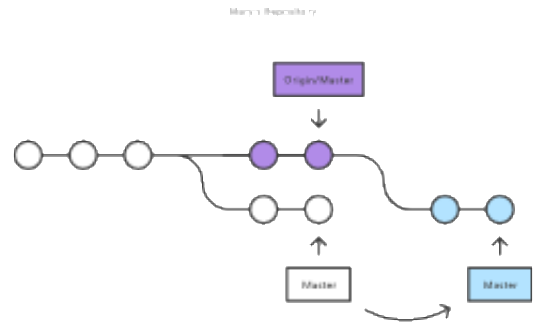
- Considerations for choosing a workflow
  - Does it scale with team size?
  - Is it easy to undo mistakes and errors that may occur?
  - Does it impose any new unnecessary cognitive overhead?
  - Is it complex and difficult to understand?
  - Short-lived branches promote cleaner merges & deploys
  - Minimize and simplify the need for reverts
  - Match a release schedule

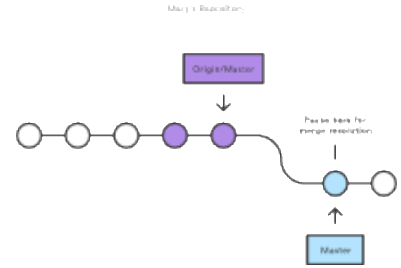
Git Workflows

- The Centralized Workflow
  - Simple workflow that doesn't require any branch besides master
  - Usually pull's are rebased: `git pull --rebase origin master`



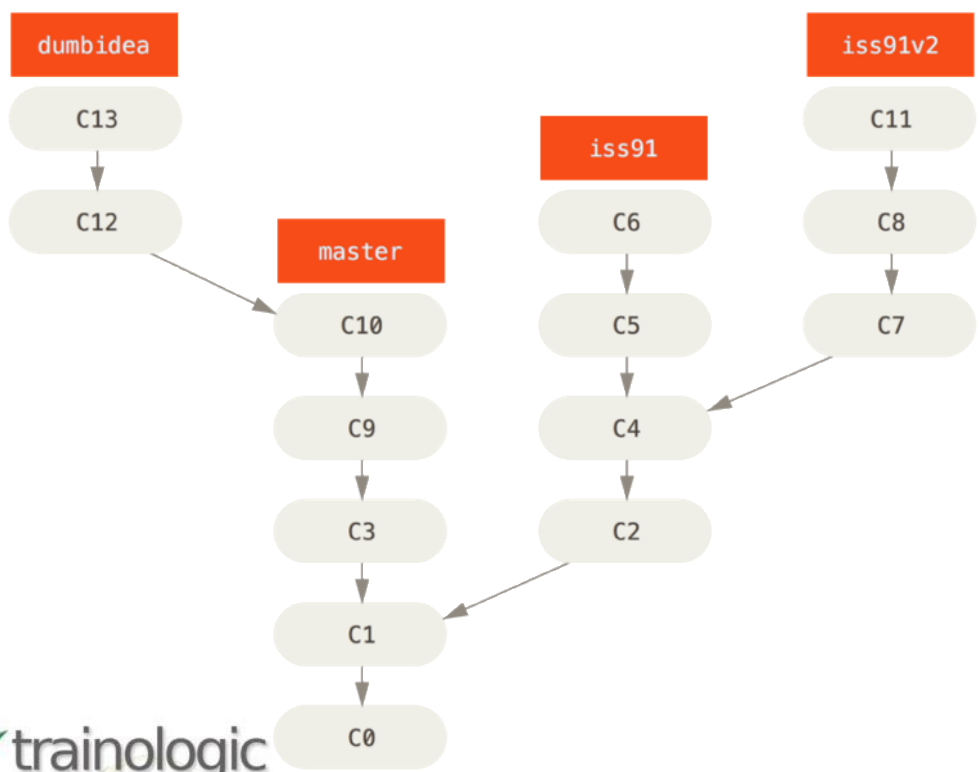


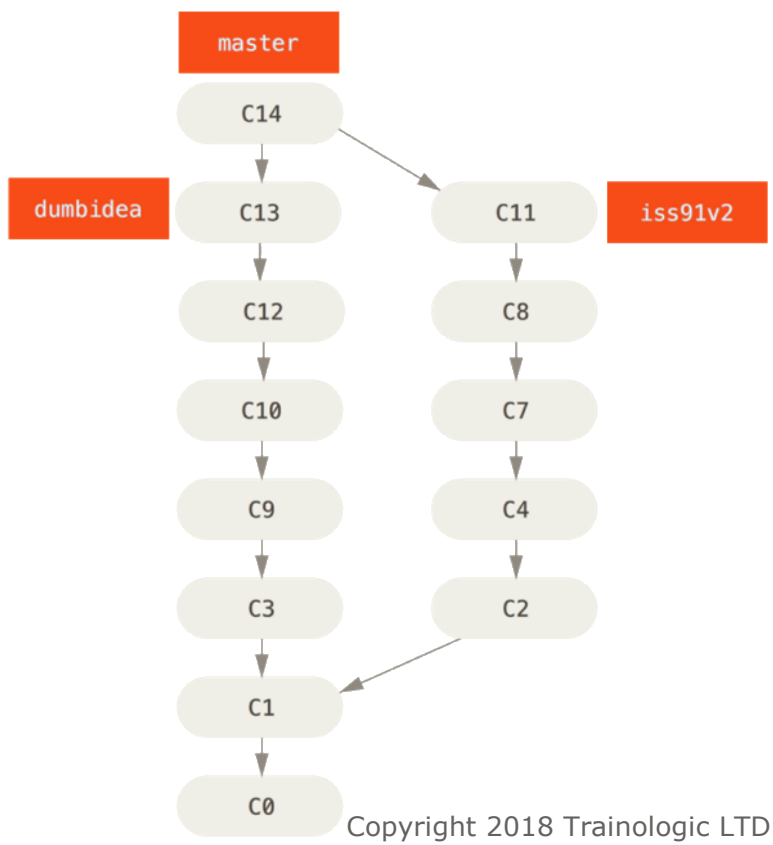




Git Workflows

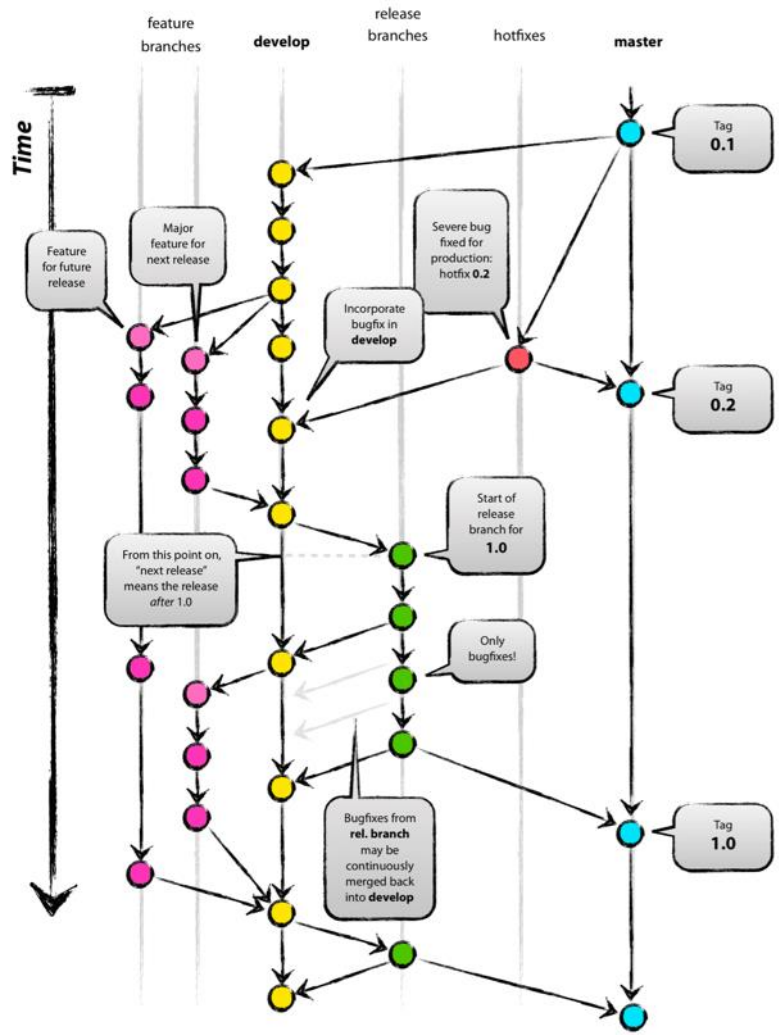
- Topic Branches / Feature branches
  - Short-lived branch created for a single feature or related work





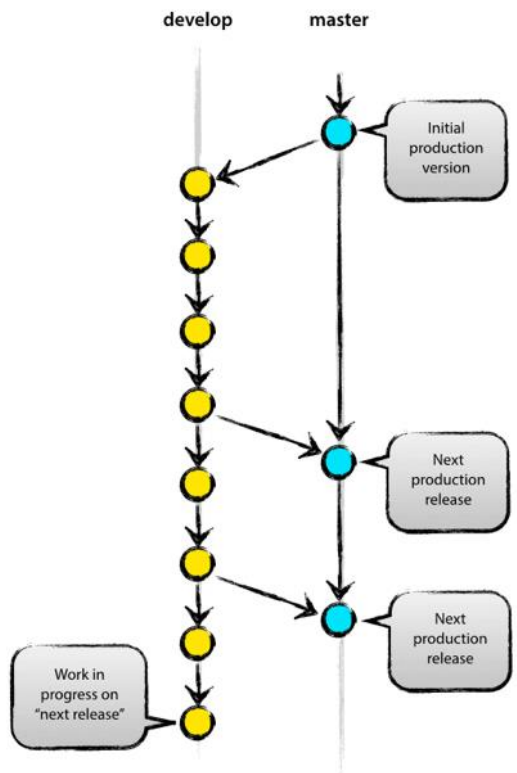
Git Workflows

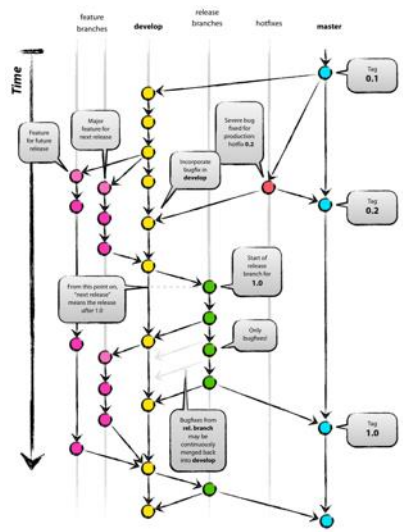
- The Gitflow Workflow



Git Workflows

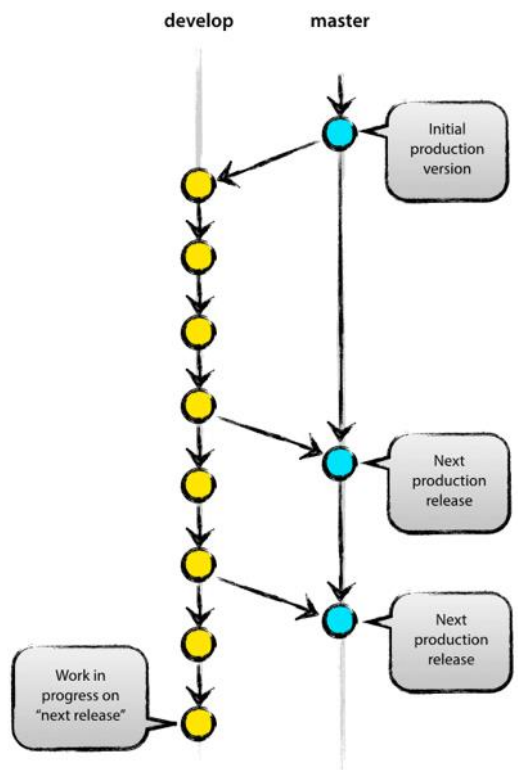
- The Gitflow Workflow
  - Strict branching model designed around project releases
  - master always reflects a production-ready state

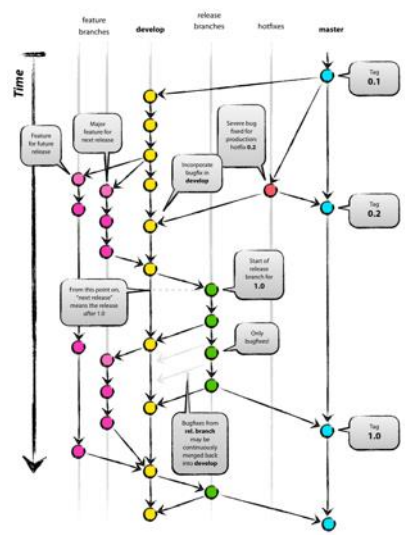




Git Workflows

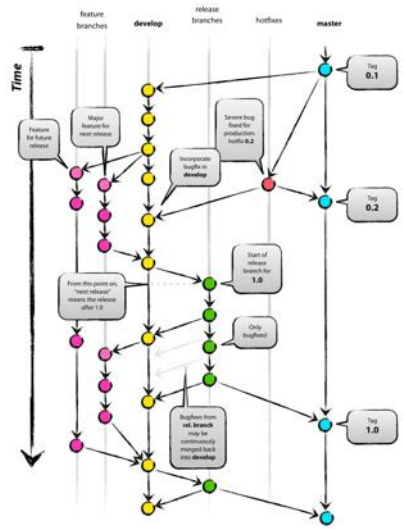
- The Gitflow Workflow
  - Strict branching model designed around project releases
  - develop is the main branch that always reflects the latest state for the next release





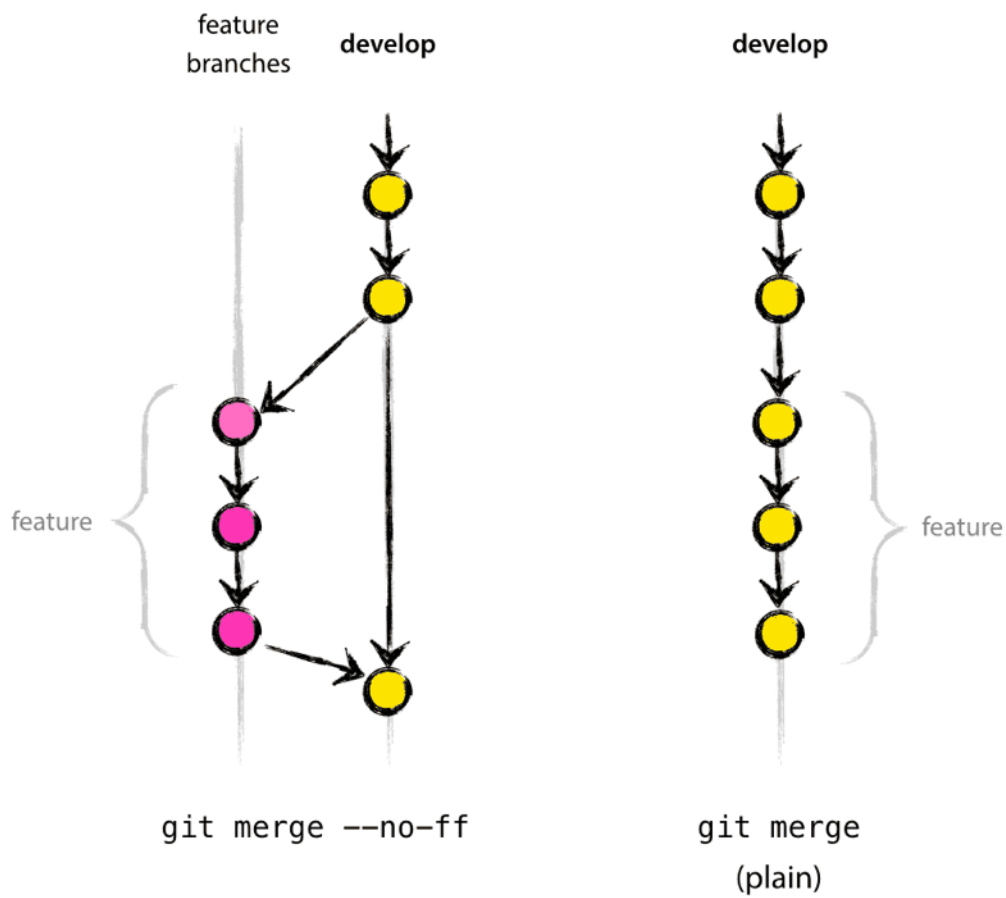
Git Workflows

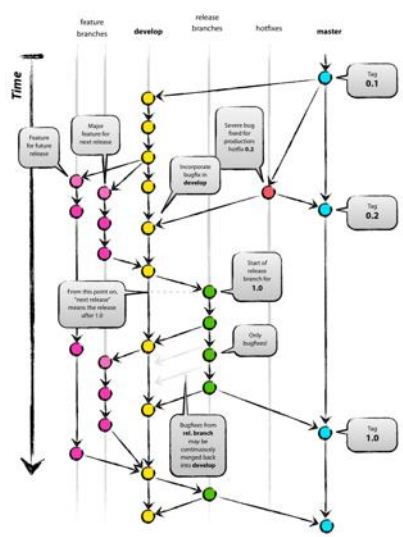
- The Gitflow Workflow
  - Strict branching model designed around project releases
  - master and develop live infinitely
  - Support branches are short-lived and may be used for
    - Feature branches
    - Release branches
    - Hotfix branches



Git Workflows

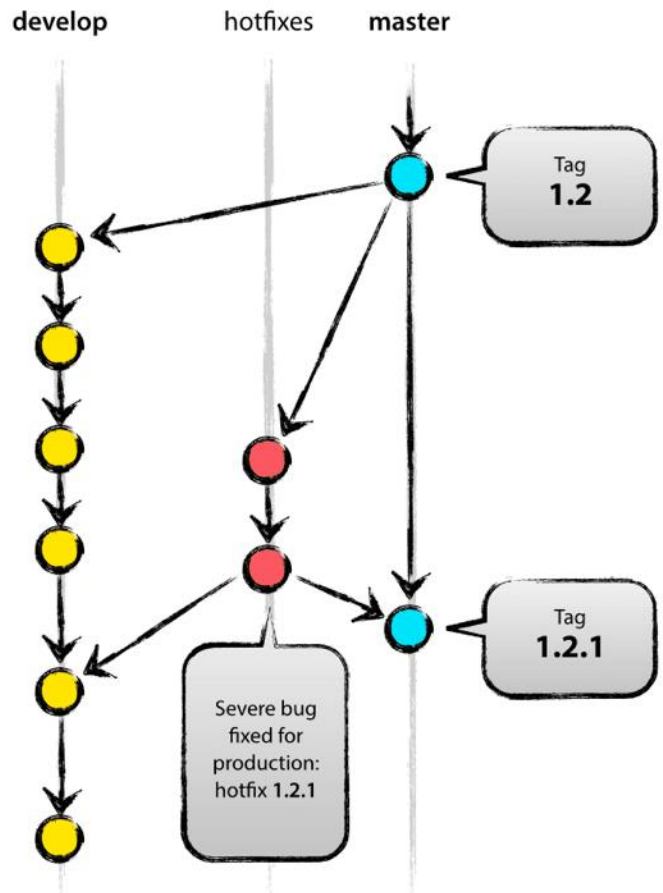
- The Gitflow Workflow
  - Strict branching model designed around project releases

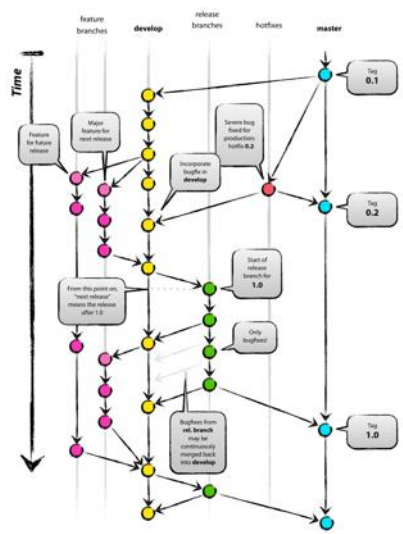




Git Workflows

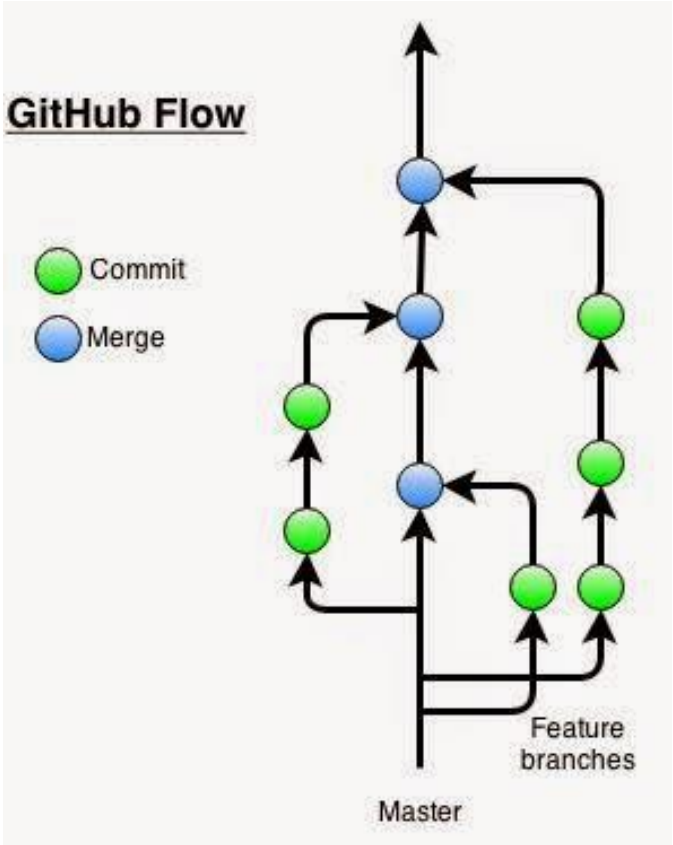
- The Gitflow Workflow
  - Strict branching model designed around project releases





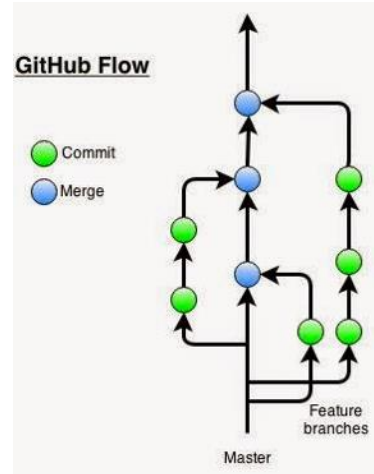
Git Workflows

- The GitHub Flow



Git Workflows

- The GitHub Flow
  - master is always deployable
  - No "releases" (as in Gitflow) – anything that is ready is merged to master and released ASAP (usually immediately)
  - Work is done on short-lived feature branches and integrated back into master through a Pull Request process

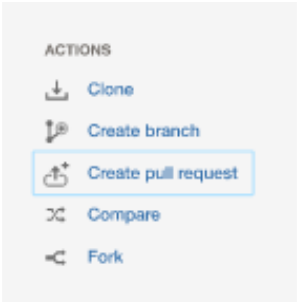


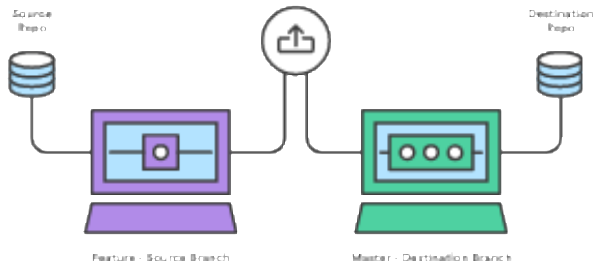
Bitbucket / GitHub

- Bitbucket and GitHub are two similar collaboration platforms that offer Git hosting as well as a Git server product for teams that want to host on their own
- Both platforms allow teams to choose any Git workflow they want by providing a centralized “remote” for everyone to sync with, and a web interface to interact with the centralized remote

Bitbucket / GitHub

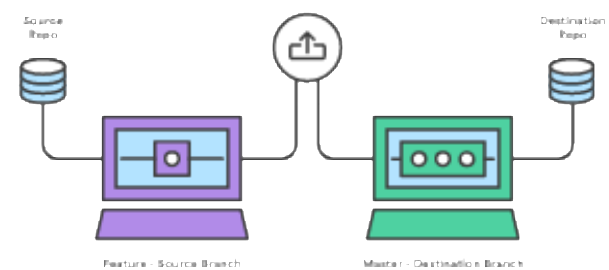
- The basic process these platforms offer to support merges between branches is the Pull Request
  - Defined by the the source repo & branch, and the destination repo & branch





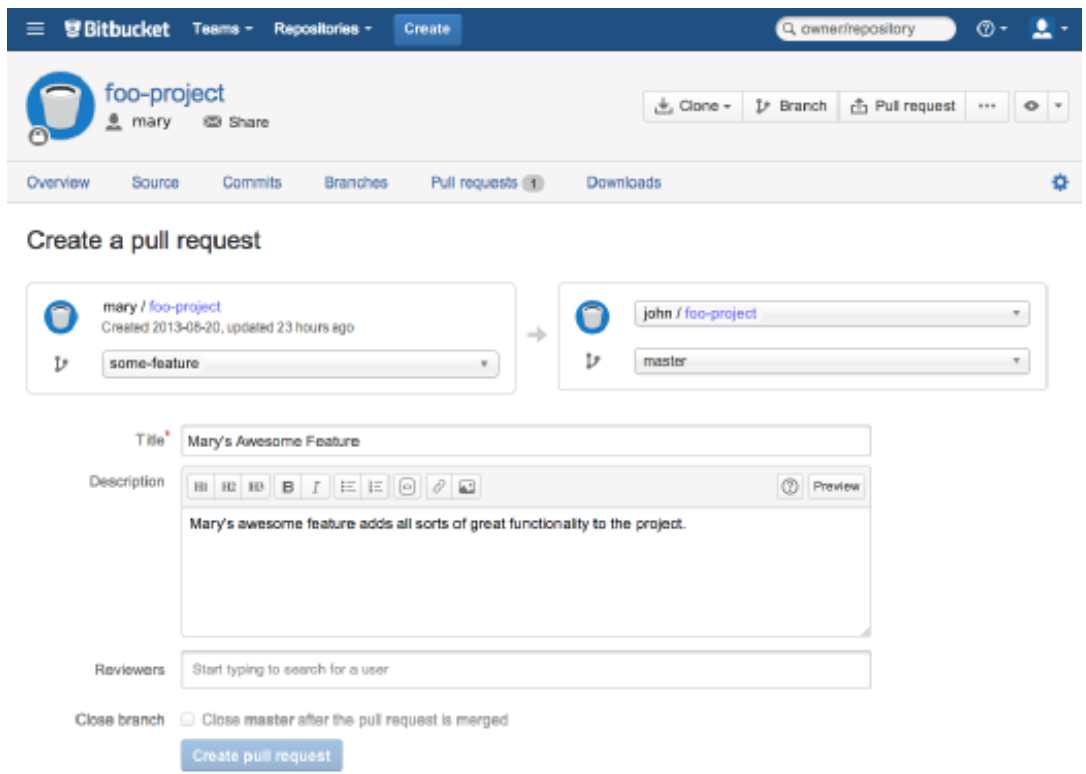
Bitbucket / GitHub

- General process around Pull Requests (PR's):
  - A developer creates a feature branch in their local repo
  - The developer pushes the branch to a public Bitbucket repo
  - The developer creates a PR via Bitbucket web interface
  - The team reviews the code changes
  - The project maintainer merges the feature into the destination repo/branch and closes the PR



Bitbucket / GitHub

- Pull Requests

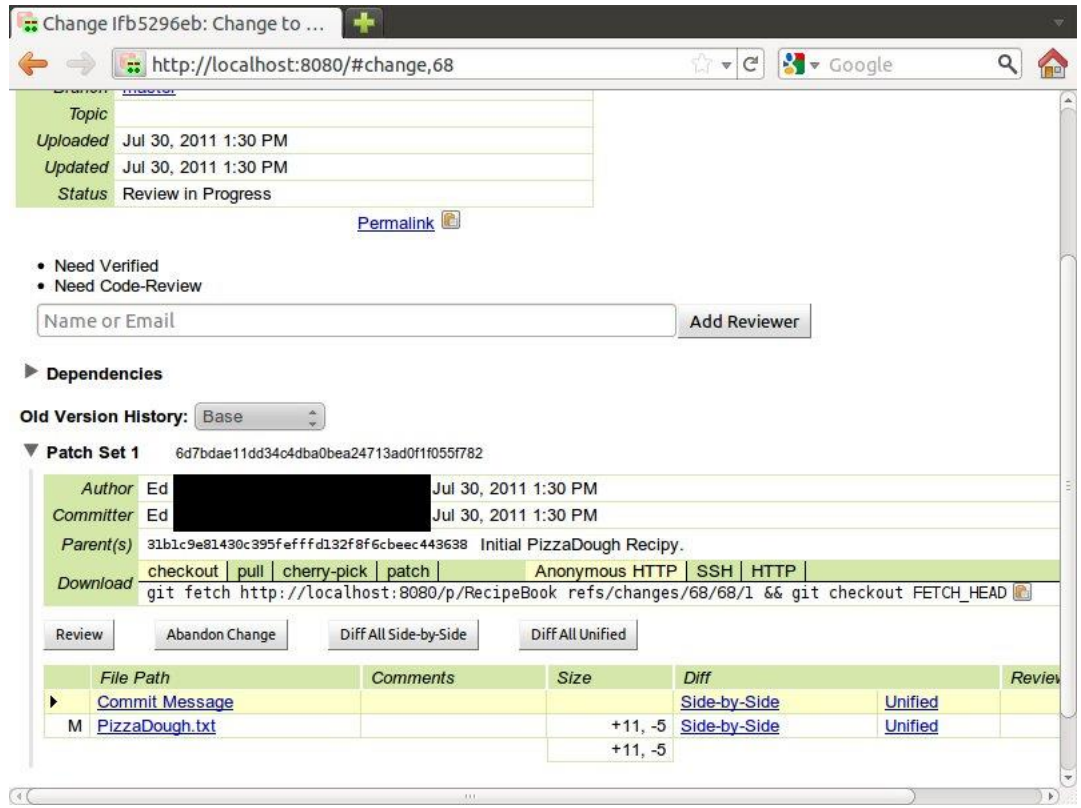


Gerrit

- Gerrit is another collaboration platforms that offer a Git server product (no hosting service)
- Gerrit is focused on being a web-based code review tool built on top of Git
- It makes code reviews easy by providing a lightweight framework for reviewing commits before they are accepted into the codebase
- It supports fine-grained roles and permissions

Gerrit

- The core concept in Gerrit is a Review

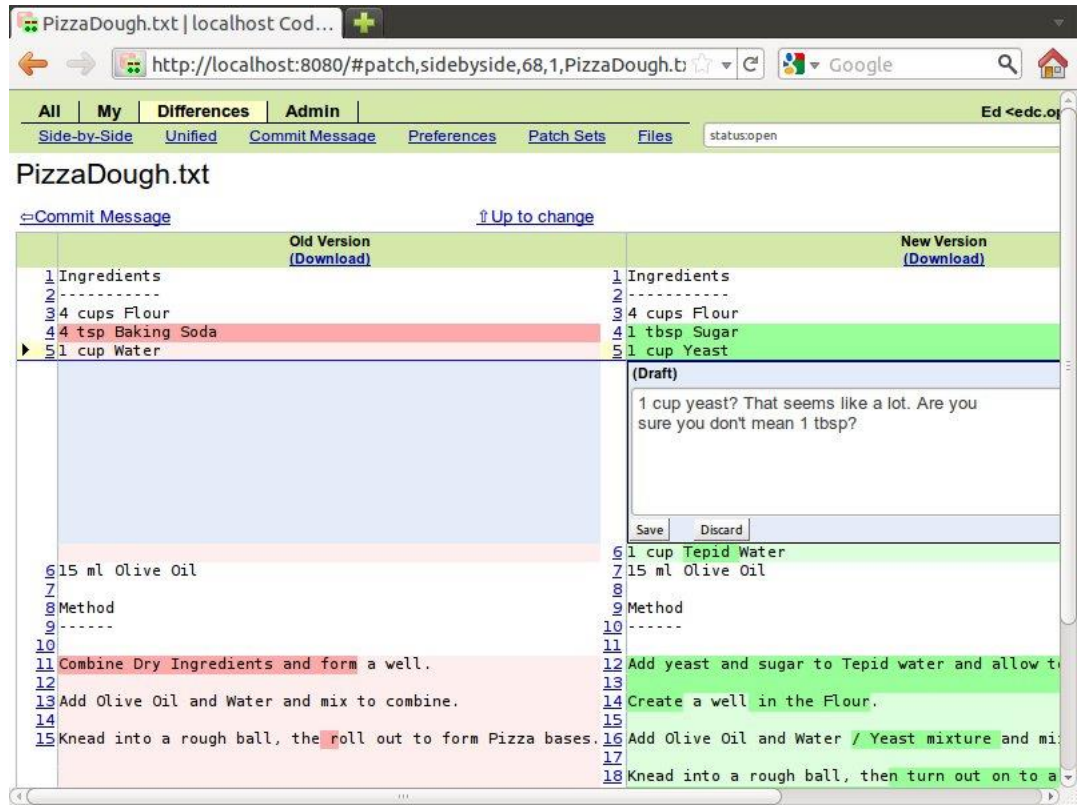


Gerrit

- Gerrit review process
  - A developer makes changes and commits locally
  - The developer pushes to a special remote branch that creates a review
    - `git push origin HEAD:refs/for/master`
  - The team reviews the commit in the Gerrit web interface

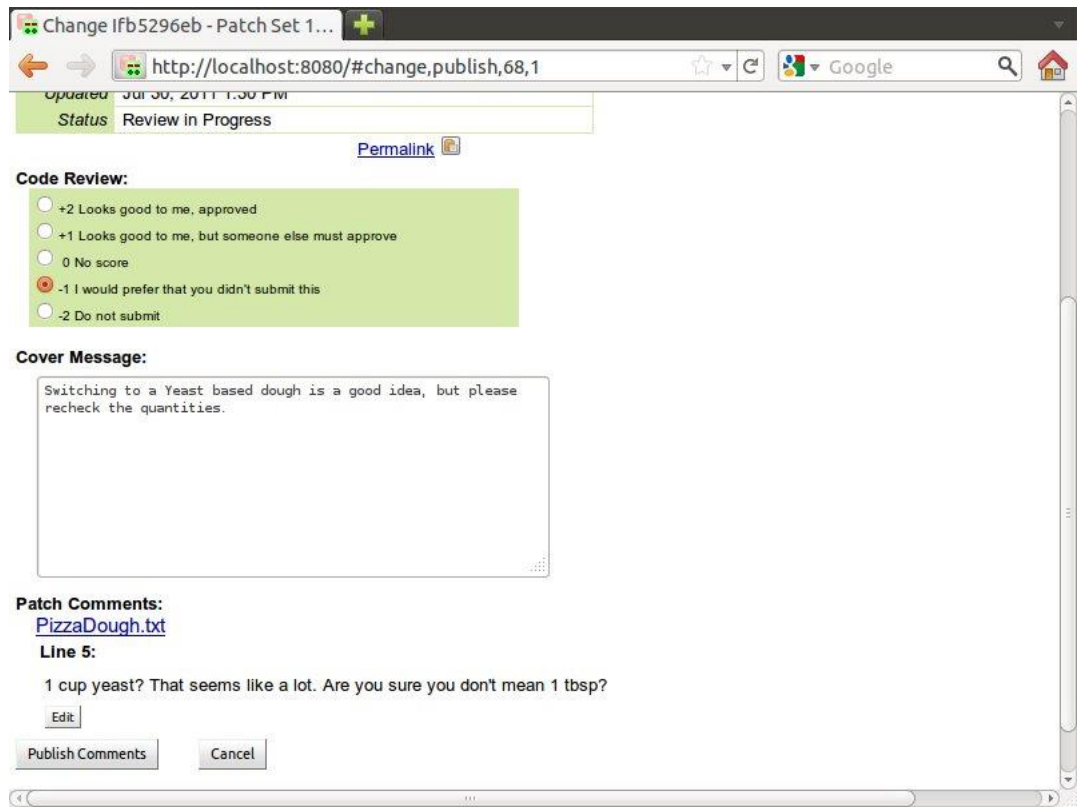
Gerrit

- Gerrit review example



Gerrit

- Gerrit review example

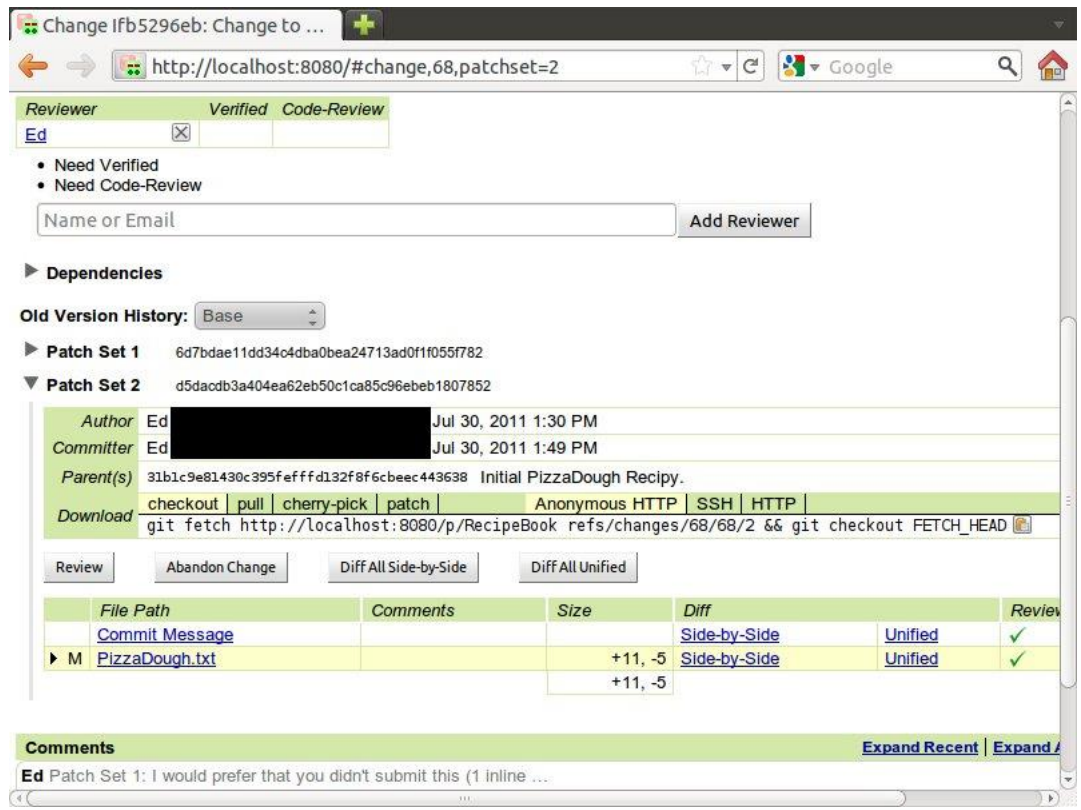


Gerrit

- Gerrit review process
  - The developer reworks the commit to incorporate feedback
  - The developer amends the commit and pushes it again to update the review
    - `git commit --amend`
    - `git push origin HEAD:refs/for/master`
  - The commit is published & submitted
  - Note: Gerrit relies on a ChangeId in the commit message to match new commits to existing reviews
    - A commit-msg hook is provided to assist with that

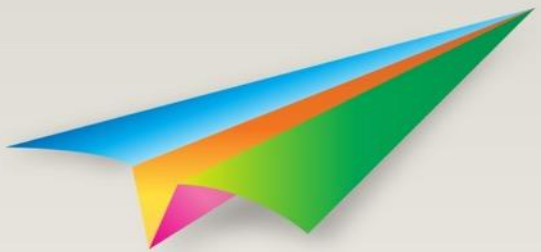
Gerrit

- Gerrit review example



Exercise #1

trainologic



Merging
Rebasing
Cherry-picks
Patches

trainologic

Merging & Rebasing

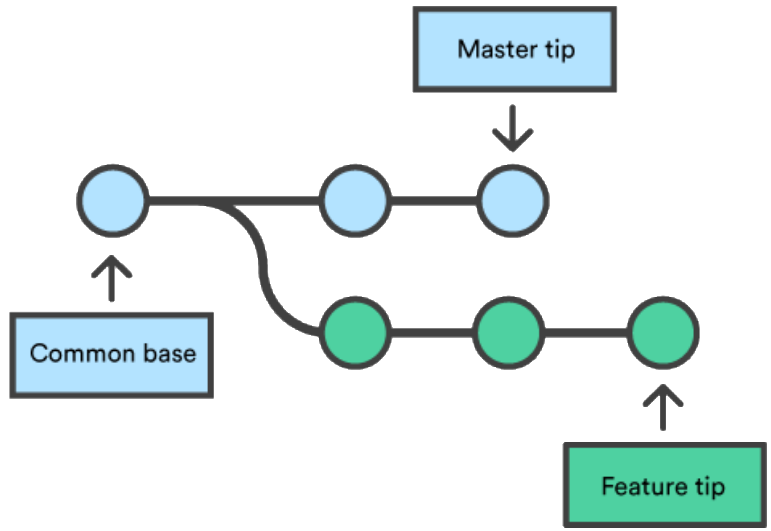
- What is "git merge"?
  - Merging is git's way of putting divergent histories back toegether
  - It is used to take independent lines of development and integrate them into a single branch
    - Usually – into the current checked out branch
    - The target branch is unaffected by a merge
    - Often, after merging, the target branch is deleted

Merging & Rebasing

- What is “git merge”?
  - Algorithmically, when asked to merge branch “foo” into the current branch “bar”, git searches for the common base commit between ”foo” and “bar”
  - Once the base commit is found, git can combine the change from the two lines that connect between base-foo and base-bar in branch ”bar”

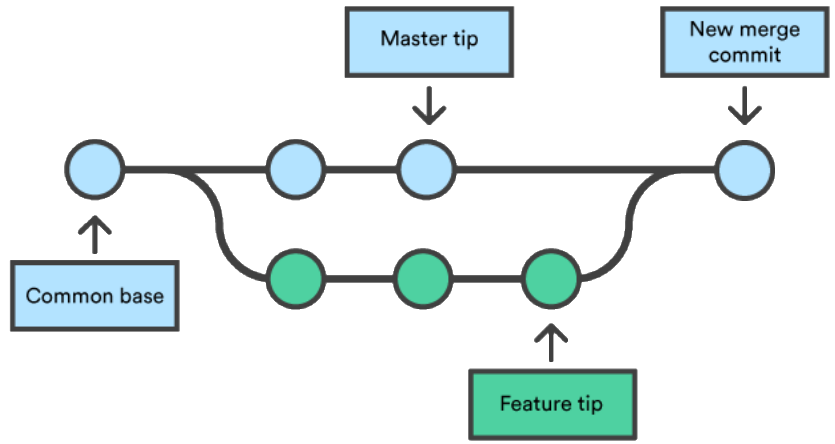
Merging & Rebasing

- What is “git merge”?
  - A common case is that the source and target branches have diverged in different directions from the base commit
    - Called “3-way merge”



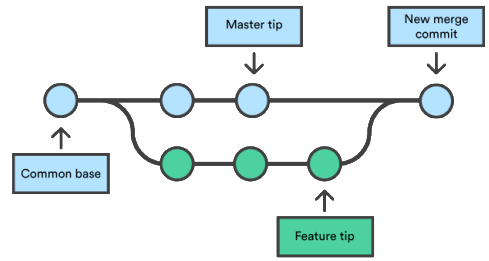
Merging & Rebasing

- What is "git merge"?
  - A common case is that the source and target branches have diverged in different directions from the base commit
  - Running `git merge feature` when master is checked out will create a new "merge commit" on the master branch



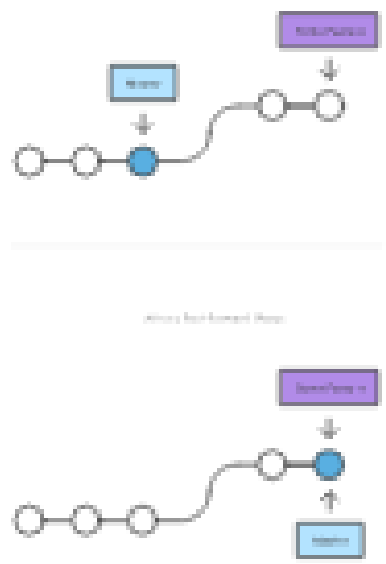
Merging & Rebasing

- What is “git merge”?
  - Merged commits are special in the sense that they have two parent commits
  - Git attempts to automatically combine the two divergent histories in the merge commit
  - Git fails to do that if some data is changed in different ways in both histories
    - A **conflict**
    - Requires user intervention to resolve



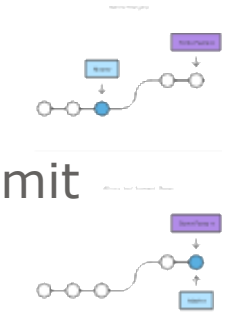
Merging & Rebasing

- What is “git merge”?
  - Another case is when there is a linear path from the current branch tip to the target branch (e.g. the histories are not divergent)



Merging & Rebasing

- What is "git merge"?
  - Another case is when there is a linear path from the current branch tip to the target branch (e.g. the histories are not divergent)
  - In this case, merging is easy – git just needs to move the current branch tip up to the target branch tip, effectively combining the histories
  - This is called a "**fast forward merge**"
  - This is not possible in a 3-way merge scenario
  - The fast forward scenario does not result a merge commit



Merging & Rebasing

- Recommended merge flow
  - Run `git status` to verify clean working tree
  - Checkout the receiving branch (e.g. master)
  - Fetch latest commits from the remote (`git fetch && git pull`)
  - Merge the target branch: `git merge <target-branch>`
  - Delete the target branch: `git branch -d <target-branch>`

Merging & Rebasing

- Forcing a merge-commit
  - Sometimes a team wants to always have a merge commit, even if a fast-forward merge is possible
    - Useful for audit purposes
  - This can be done using the --no-ff flag:
    - `git merge --no-ff <target-branch>`

Merging & Rebasing

- Avoiding a merge-commit
  - Sometimes a team wants to never have a merge commit (e.g. always use fast-forward merging)
    - Makes the history on the receiving branch cleaner
  - As we saw, fast-forward merges are not always possible
  - This is where rebasing is useful, as we'll see later

Merging & Rebasing

- Dealing with conflicts
  - If the two branches both changes the same part of the same file in different ways, git won't be able to figure out which version to use, and will need help from the user to decide what to do
  - In this situation, git prepares a merge commit, but stops before making the commit so the user can help resolve the conflict(s)
  - Git uses the familiar edit-stage-commit workflow to resolve merge conflicts
  - Running `git status` shows which files have unresolved conflicts

Merging & Rebasing

- Dealing with conflicts
  - When git encounters a conflict, it edits the files with visual conflict markers that indicate the scope of the conflict:
    - <<<<<<<, =======, and >>>>>>>
  - It is useful to search for these markers to quickly get to the conflict area
  - Generally, the content above the ======= is from the receiving branch, and the content below it is from the target branch

Merging & Rebasing

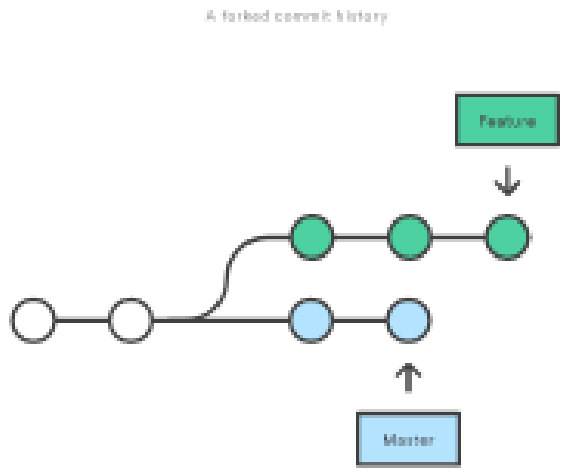
- Dealing with conflicts
  - You'll need to fix the conflict manually (choose either block and delete the other one, or rewrite the block to reflect the logical intent in both), and remove the conflict markers
    - `git checkout <--ours|--theirs> path/to/file` can help
  - Once all conflicts in a file are resolved, the file can be staged using `git add <file>`
  - Once all conflicts are resolved and all files are staged, just resume the merge by running `git commit` to generate the merge commit
  - Naturally, fast-forward merges can't have conflicts

Merging & Rebasing

- What is “git rebase”?
  - Rebase is intended to solve the same problem that merge does – to integrate changes from one branch into another branch
  - Rebase achieves this in a very different way compared to merge

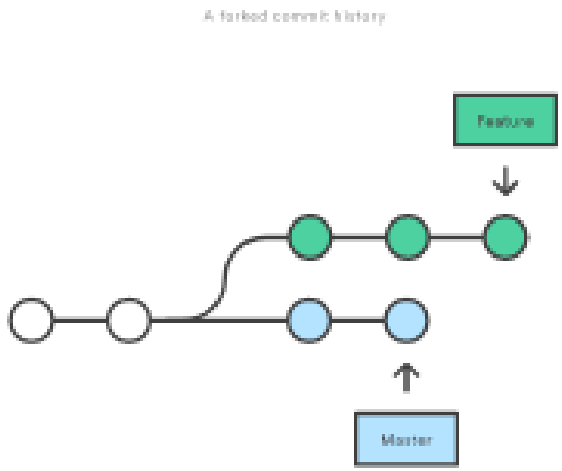
Merging & Rebasing

- What is “git rebase”?
  - Consider the common scenario, where you work on “Feature” in a feature branch that is based on master, and someone else pushes new commit to master
  - This results a divergent, or forked, history



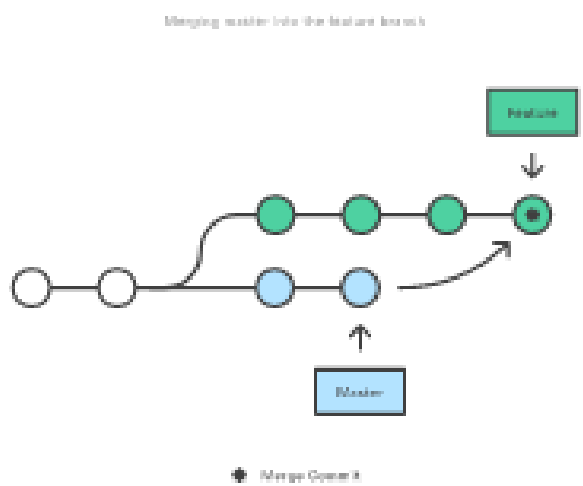
Merging & Rebasing

- What is “git rebase”?
  - If you want to integrate the new commits from master into Feature, you have two options: merging or rebasing



Merging & Rebasing

- What is “git rebase”?
  - If you choose to merge (`git merge master` when feature is checked out) then git will create a merge commit on the feature branch that brings in the divergent commits from master



Merging & Rebasing

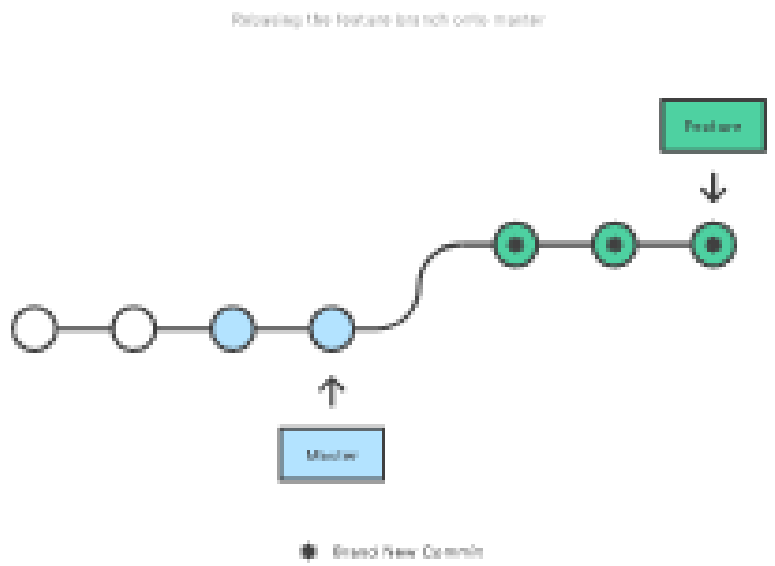
- What is "git rebase"?
  - If you choose to merge (`git merge master` when feature is checked out) then git will create a merge commit on the feature branch that brings in the divergent commits from master
    - This is nice because it's "non-destructive" and simple
    - This is also a little messy, because it adds redundant commits on the feature branch that have nothing to do with the feature
      - If you branched out from master a bit later, then you wouldn't need this merge commit…
      - Can pollute the branch if master is very active

Merging & Rebasing

- What is "git rebase"?
  - Rebase helps with this
    - `git rebase master` (when feature is checked out)
  - This command "rebases" the feature branch on top of master
    - This means that the entire "feature" branch is ***moved*** to begin on the tip of the master branch
    - It ***rewrites*** the history of the "feature" branch, as if all the commits that happened on the branch in fact happened after the tip of master
    - It "replays" all the commits on top of master

Merging & Rebasing

- What is “git rebase”?
  - After a rebase, all the commits on the “feature” branch are different from those that were there originally!



Merging & Rebasing

- What is “git rebase”?
  - This allows for a much cleaner project history
    - It eliminates the redundant merge commits
    - It results a perfectly linear project history, making it possible to perform fast-forward merges back into master, even if originally it wasn’t possible
  - These benefits are not free… You pay with:
    - Safety
    - Traceability

Merging & Rebasing

- Costs of rebasing
  - If you rewrite the history of a branch that you collaborate on with others, they will have hard time sync'ing
    - Potentially it can have catastrophic effects
  - Also, by avoiding merge commits, you lose the context that merge commits provide

Merging & Rebasing

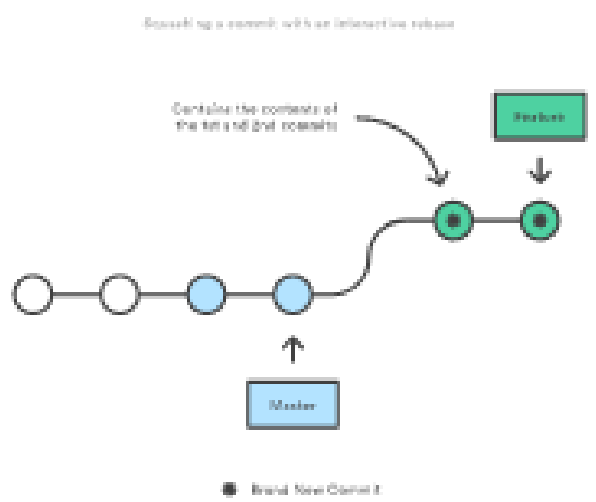
- Interactive Rebasing
  - Interactive rebasing gives you the opportunity to alter & mutate commits as they are rewritten
  - It's even more powerful than a plain rebase, that just replays the commits as they were, giving you total control over the branch history
  - Typically, this is used to clean up a branch history before merging a feature into master
  - To initiate an interactive rebase on top of master:
    - `git rebase -i master`

Merging & Rebasing

- Interactive Rebasing
  - `git rebase -i master`
  - This opens a text editor with all the commits that are about to be rewritten:
    - ```
      pick 33d5b7a Message for commit #1
      pick 9480b3d Message for commit #2
      pick 5c67e61 Message for commit #3
      ```
  - The list defines the “rebase script” to execute
  - You can change the ”pick” commands and/or reorder & delete lines to shape and manipulate the new branch history
      ```

Merging & Rebasing

- Interactive Rebasing
  - `git rebase -i master`
  - For example, by changing “pick” to “squash” or “fixup” you collapse the commit into the previous commit – making them a single commit

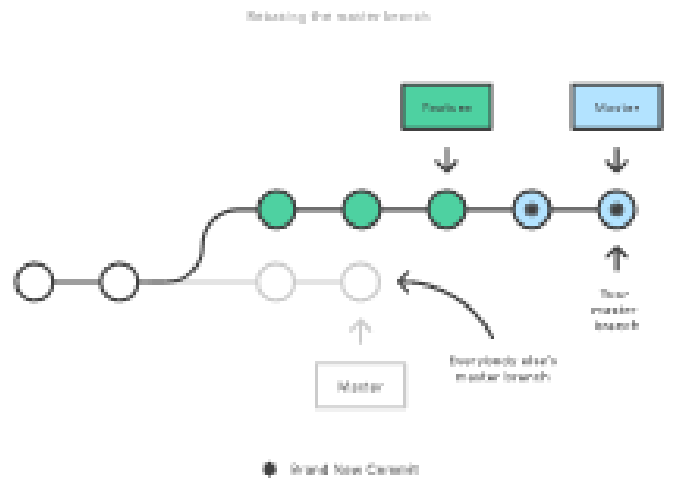


Merging & Rebasing

- Interactive Rebasing
  - `git rebase -i master`
  - For example, by changing “pick” to “squash” or “fixup” you collapse the commit into the previous commit – making them a single commit
    - ”squash” keeps the commit message of both commits, joining them into a single commit message (that you will be able to edit)
    - “fixup” discards the commit message, keeping only the message from the “picked” commit

Merging & Rebasing

- The Golden Rule of Rebasing
  - Once you understand rebasing, the most important thing to learn is when *not* to do it
  - The golden rule is to never use it on *public* branches
  - What would happen if you rebased master on top of your feature branch?



Merging & Rebasing

- The Golden Rule of Rebasing
  - Once you understand rebasing, the most important thing to learn is when *not* to do it
  - Before running rebase, ask yourself, is anyone else looking at the branch I'm about to rewrite?
  - If the answer is yes – don't do it!

Merging & Rebasing

- Force-pushing rebased branches
  - If you try to push a branch with rewritten history to a remote that already has to original branch, the push will fail
  - To push the new branch anyway, you must use the –force
    - `git push --force` (or `-f`)
  - This overwrites the remote branch to match the rebased branch
  - Again, be careful with this command, and use it only when you know exactly what you're doing…

Merging & Rebasing

- Last rebase warning
  - Once again, remember –
    DO NOT MESS WITH THE HISTORY OF PUBLIC BRANCHES

Cherry-picking

- Git cherry-pick
  - Cherry-picking is the git tool to re-apply (or "replay") changes that were already introduced by some existing commit to the current working tree
    - `git cherry-pick <commit> <commit..>`
  - It's a useful way to "copy" fixes between branches
    - For example, a hotfix on a release branch can be applied to the mainline by cherry-picking it
  - The working tree must be clean (no local changes)

Git Patch

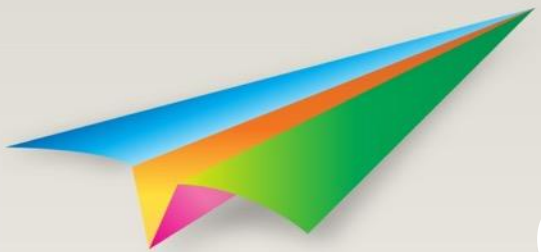
- What is a Git patch?
  - Patching is one way to share out a branch or a commit with others, without pushing commits to a remote
  - Useful when working on someone else's project with no write permissions and no fork permissions
    - Common in e-mail based workflows

Git Patch

- Creating a patch
  - From current branch to master:
    - `git format-patch master --stdout > myfix.patch`
  - The patch will include all commits in the branch which are not in master

Git Patch

- Apply a patch
  - To apply a patch from a file to the current branch:
    - `git am < myfix.patch`



Resetting

Checking out

Reverting

trainologic

Resetting, Checking Out & Reverting

- Three very useful git commands
  - git checkout
  - git reset
  - git revert
- They can be used for similar things, which can be confusing, making it harder to decide which command should be used in any given situation
- It helps to think about the commands in terms of their effect on the "three tree states" of the Git model – the working tree, the staging area (or index), and the commit history

Resetting, Checking Out & Reverting

- Git checkout
  - A checkout moves the HEAD ref pointer to a specified commit
  - It updates the working tree to match a specific commit in the commit history
  - It can be used in a file level scope – a file level checkout changes the file's content to those of the specific commit

Resetting, Checking Out & Reverting

- Git revert
  - A revert is an operation that takes a specified commit and creates a new commit which inverses the specified commit
  - It only works on a commit level scope – there's no file level functionality

Resetting, Checking Out & Reverting

- Git reset
  - A reset takes a specified commit and resets the “three trees” to match the state of the repository at that specified commit
  - A reset can be invoked in 3 different modes

Resetting, Checking Out & Reverting

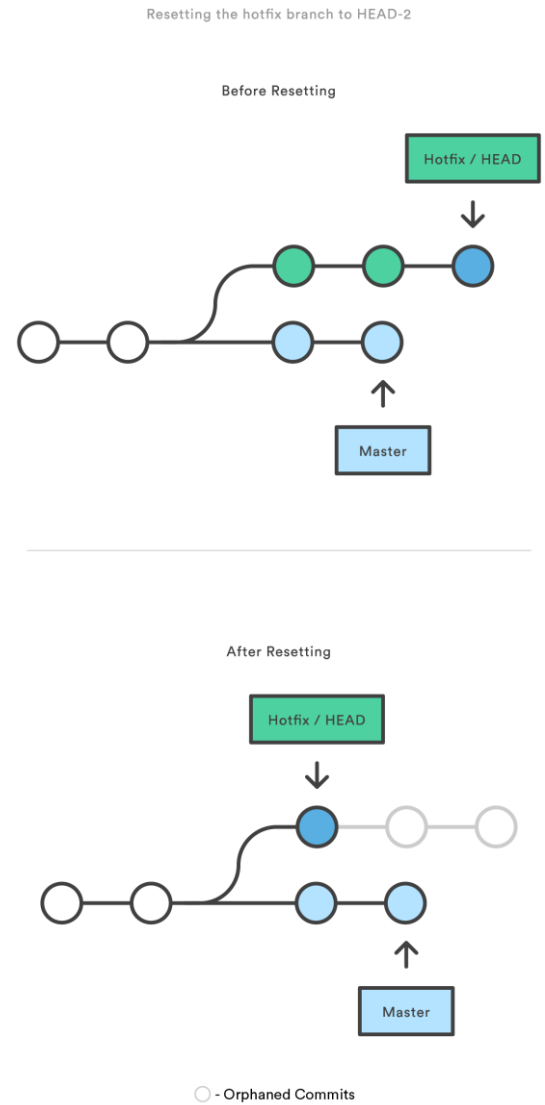
- Checkout and reset and generally used for making local or private "undo" operations
  - They modify the history of a repository
- Revert is a safe operation that allows "public undo", because it creates a new reverse commit without rewriting history

Resetting, Checking Out & Reverting

- Common use cases
  - git reset
    - Commit-level: discard commits in a private branch, or throw away uncommitted changes
    - File-level: unstage a file
  - git checkout
    - Commit-level: switch between branches
    - File-level: discard changes in the working tree
  - git revert
    - Commit-level: undo commits in a public branch

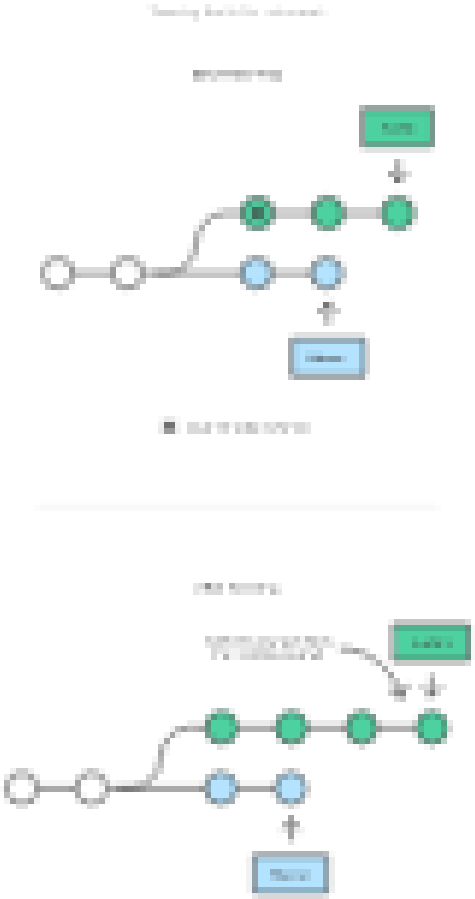
Resetting, Checking Out & Reverting

- `git reset HEAD~2`
  - Move the tip of the current branch 2 commits back, discarding the last 2 commits
- Options for git reset:
  - --soft: don't change the staging area and the working tree
  - --mixed (default): don't change the working tree, but update staging area to the specified commit
  - --hard: both staging area and working tree are updated to the specified commit



Resetting, Checking Out & Reverting

- `git revert HEAD~2`
  - Creates a new commit on the current branch that reverses the changes that were introduced 2 commits ago
  - Doesn't alter the history



Resetting, Checking Out & Reverting

- Aborting merges / rebases
  - Another common use-case is during a merge/rebase that involves a lot of conflicts
  - The easiest way to abort the operation is to reset:
    - `git checkout foo && git fetch && git pull`
    - `git merge master  # too many conflicts, abort!`
    - `git reset --hard origin/foo`

Undoing Things

trainologic

Undoing Things

- We already covered a bunch of ways to undo operations, in different contexts
- In this section we'll focus on the "undo" aspect of things, revisiting some methods we already covered, and adding some new ones

Undoing Things

- Unstaging a staged modifications
  - To move changes in file “foo” from the staging area back to the working tree:
    - `git reset -- foo`

Undoing Things

- Discarding unstaged modifications
  - To throw away modifications to file “foo” that are not staged, and revert “foo” back to its state in HEAD:
    - `git checkout -- foo`

Undoing Things

- Fixing the commit message in the last (private) commit
  - `git commit --amend`
  - Can also be used to add more changes to the last commit
  - Note that it rewrites the last commit, so don't do that if the last commit was published

Undoing Things

- Undo a public commit
  - To "undo" the commit "HEAD~2" that was already published:
    - `git revert HEAD~2`
  - To "undo" the last commit that is a merge-commit, assuming the mainline branch was checked out for the merge:
    - `git revert -m 1 HEAD^`

Undoing Things

- Undo a private commit
  - To discard the last 2 commits on the current (private) branch:
    - `git reset --hard HEAD~2`
  - To discard the last 2 commits on the current (private) branch, but keep the content of the commits in the working tree:
    - `git reset --mixed HEAD~2`
  - Same thing, but keep the content in the staging area:
    - `git reset --soft HEAD~2`

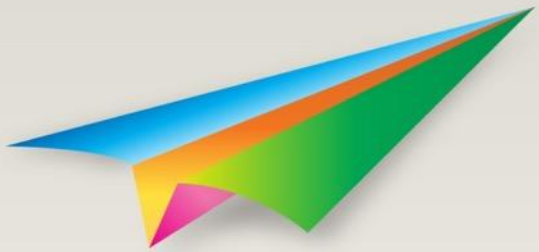
Undoing Things

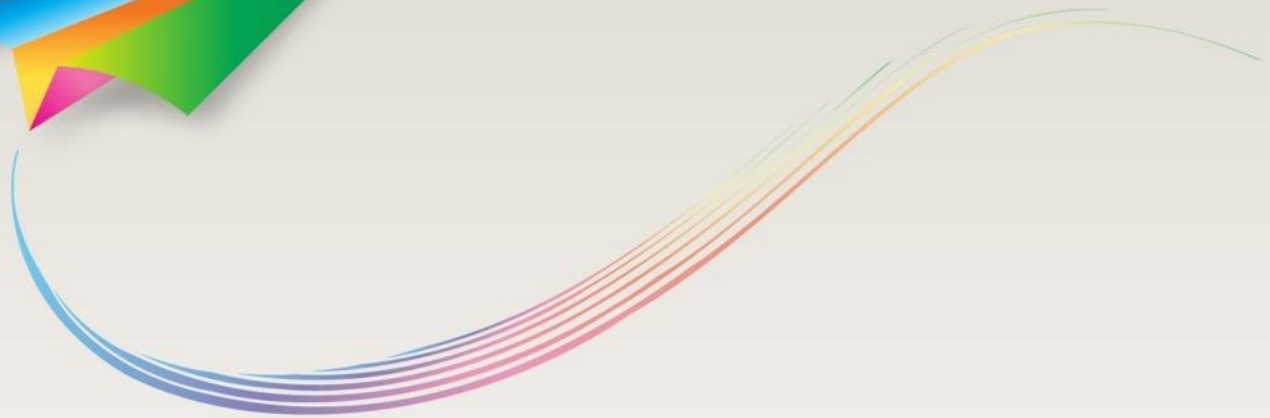
- Go back to a specific point in history
  - Change the current branch back to commit 9e7afac
    - `git reset --hard 9e7afac`

Undoing Things

- Going nuclear – purging the file "foo" from the entire history of the repository (**massive history rewrite!!!**)
  - `git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch foo' --prune-empty --tag-name-filter cat -- --all`
  - `git for-each-ref --format='delete %(refname)' refs/original | git update-ref -stdin`
  - `git reflog expire --expire=now -all`
  - `git gc --prune=now`
  - `git push origin --force --all --tags`

Exercise #2





trainologic

History, Logs, Archive

trainologic

Advanced Git Log

- The purpose of any VCS is to record and keep track of changes over time
- Having all the history is useless if you don't know how to navigate it
- For that, `git log` is your best friend
  - Basic usage is pretty straight forward
  - This can hide the great powers inside git log

Advanced Git Log

- Advanced features of git log can be divided into
  - Formatting how each commit is displayed
  - Filtering which commits are included in the output
- Together, you can find any information that you could possibly need

Advanced Git Log

- Advanced formatting options
  - --oneline condenses each commit to a single line
  - --decorate adds to each commit all references (branches etc.) that point to it
  - --stat displays the number of insertions and deletions to each file in each commit
  - -p shows the actual diffs for each commit

Advanced Git Log

- The shortlog
  - git shortlog is a special version of git log intended for creating release announcements
  - It groups commits by author and displays the first line of each commit message
  - The output is sorted by author name, or by number of commits (by passing -n)

Advanced Git Log

- Graphs
  - The --graph option draws an ASCII graph representing the branch structure of the commit history
  - A useful combination is `--decorate --oneline --graph` – try it

Advanced Git Log

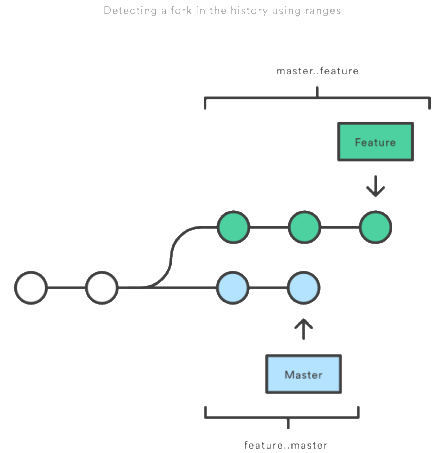
- Custom formatting
  - Any other formatting can be expressed with --pretty=format:”<string>”, using printf-style placeholders
  - Example:
  - `git log --pretty=format:"%cn committed %h on %cd”`
  - See also: https://www.kernel.org/pub/software/scm/git/docs/git-log.html#_pretty_formats

Advanced Git Log

- Filtering the commit history
  - By amount: -<n>
  - By date: --after <date> / --before <date>
    - <date> can be things like “2018-03-15”, “yesterday”, “1 week ago”, etc.
    - Can be combined to create a filter window
    - Synonymous with --since / --until flags
  - By author: --author=“<pattern>” / --committer=“<pattern>”
    - <pattern> can be a string (contains “John”), or regexp (“John\|Mary”)
    - author includes name & email

Advanced Git Log

- Filtering the commit history
  - By message: --grep="<pattern>" [-i]
  - By file: git log -- <file> <file...>
  - By content: -S"<string>" / -G"<regexp>"
    - Called "pickaxe"
    - Filter for commits to add OR remove <string> in the code
  - By range: git log <since>..<until>
    - For example: git log master..fix_foo
  - Without merge commits: --no-merges
  - Only merge commits: --merges



Advanced Git Log

- Filtering tips & tricks:
  - ”behind” – show all commits that are in the remote branch and not in the local branch (e.g. commits to pull): `git log ..@{u}`
  - “ahead” – show all commits that are in the local branch and not in the remote (e.g. commits to push): `git log @{u}..`

Advanced Git Log

- Git blame
  - For every file included, show for each line in what revision and by what author it was last modified
  - `git blame -- <file..>`
  - Many more options – see `git help blame`

Advanced Git Log

- Git bisect
  - Powerful debugging tool
  - Use binary search on the history to find a commit that introduced a bug
  - `git bisect start`
  - `git bisect bad  # mark HEAD as bad`
  - `git bisect good v1.2.3  # ref v.1.2.3 is known good`
  - Automatically checks out commit in the middle, let's you test it and mark it good/bad, and continue recursively until the offending commit is found

Advanced Git Log

- Git archive
  - The ”git archive” command creates an archive of files from a named tree
  - Useful for creating source release packages
  - --format=<tag|zip>
  - By default writes to stdout - bypass by passing -o=<filename>
  - Example – create compressed tarball for v1.4.0 release (tag), putting everything under a top-level dir named “v1.4.0”:

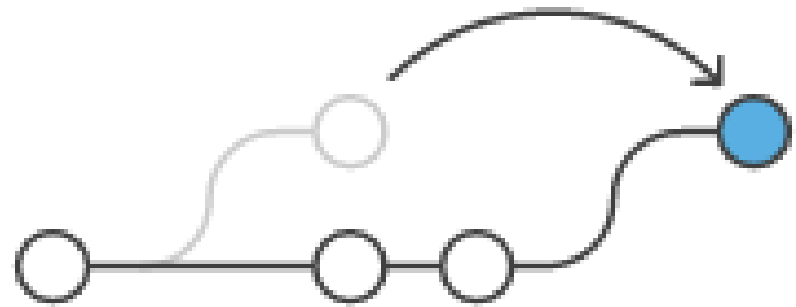
```
git archive --prefix=v1.4.0/ -o foo-1.4.0.tar.gz v1.4.0
```

Refs & Reflog

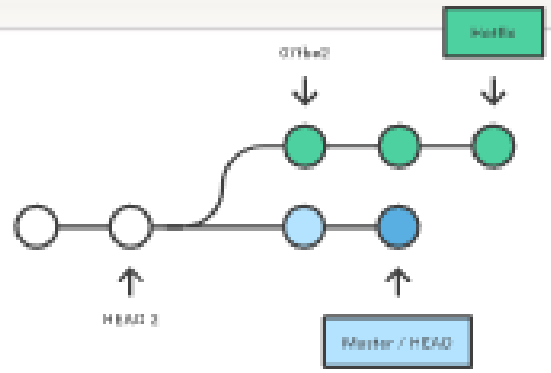
trainologic

Refs & Reflog

- Refs
  - Git is all about commits
  - The majority of the git commands operate on a commit in some form or another
  - Many commands accept a commit reference (“ref”) as a parameter
  - Understanding all the many ways to refer to a commit can help making all these commands much more powerful

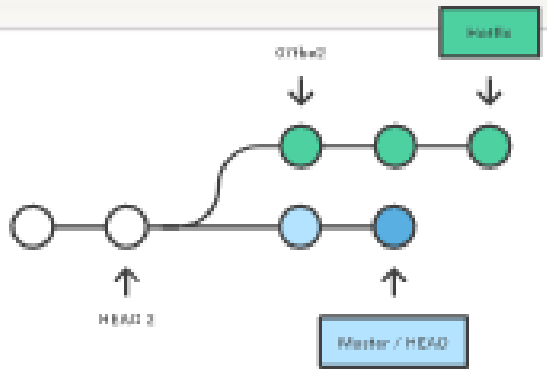


Refs & Reflog



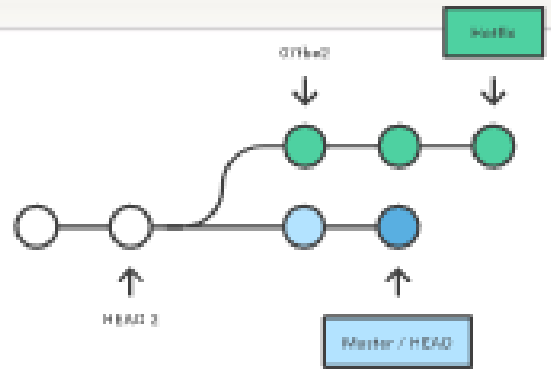
- Refs - Hashes
  - The SHA-1 hash of a commit is the most direct way to refer to a commit, as it is the unique ID of a commit
  - The git log output usually shows the hashes of the commits it displays
  - Tip: when using the hash as a commit reference, you only need to specify enough characters of the hash to uniquely identify it

Refs & Reflog



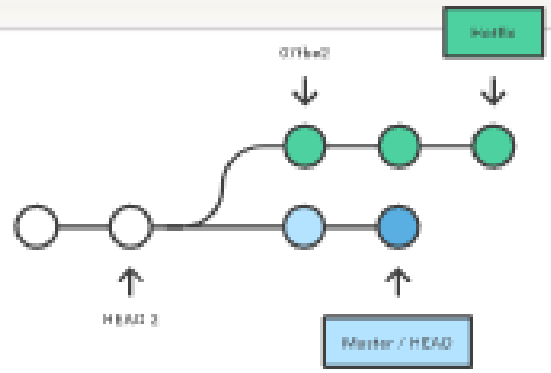
- Refs – Resolving refs
  - It is sometimes needed to resolve some indirect ref (“HEAD”, “master”, etc.) to a commit hash
  - This can be done with the rev-parse command:
    - `git rev-parse <ref-name>`
  - Can be useful in scripts and tools that operate on commits

Refs & Reflog



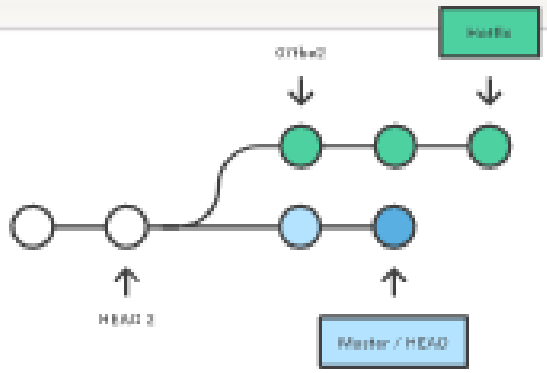
- Refs – So what's a ref??
  - A "ref" is an indirect way of referring to a commit
  - A "user friendly" alias for a commit hash
  - Git's internal mechanism of representing branches and tags
  - Stored under ".git/refs"
    - heads – all local branches in the repository
    - tags – all local tags in the repository
  - Moving a branch or creating a branch is just writing the hash of a commit to some file under ".git/refs/heads", which is why git can be so fast

Refs & Reflog



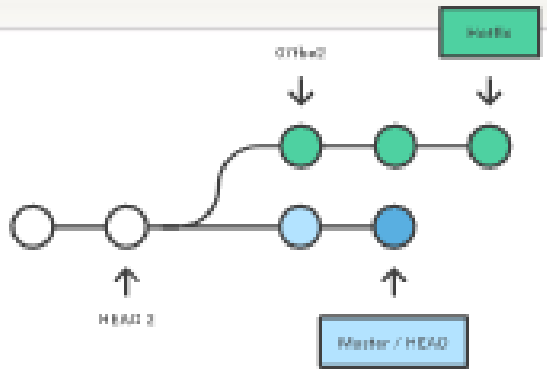
- Refs – short names
  - Git commands that take refs can work with short names and full names
  - Short names should be familiar – “master”, “foo”, …
    - `git show foo-branch`
  - Full refs include the full path to the ref file
    - `git show refs/heads/foo-branch`
  - Can be useful when there may be ambiguity
    - Branch and tag with the same name

Refs & Reflog



- Packed refs
  - For large repositories, git may periodically perform garbage collection to remove unnecessary objects and compress refs into a single file for efficiency
  - Manually invoking garbage collection: `git gc`
  - This moves all individual ref files to a single “packed-refs” file

Refs & Reflog



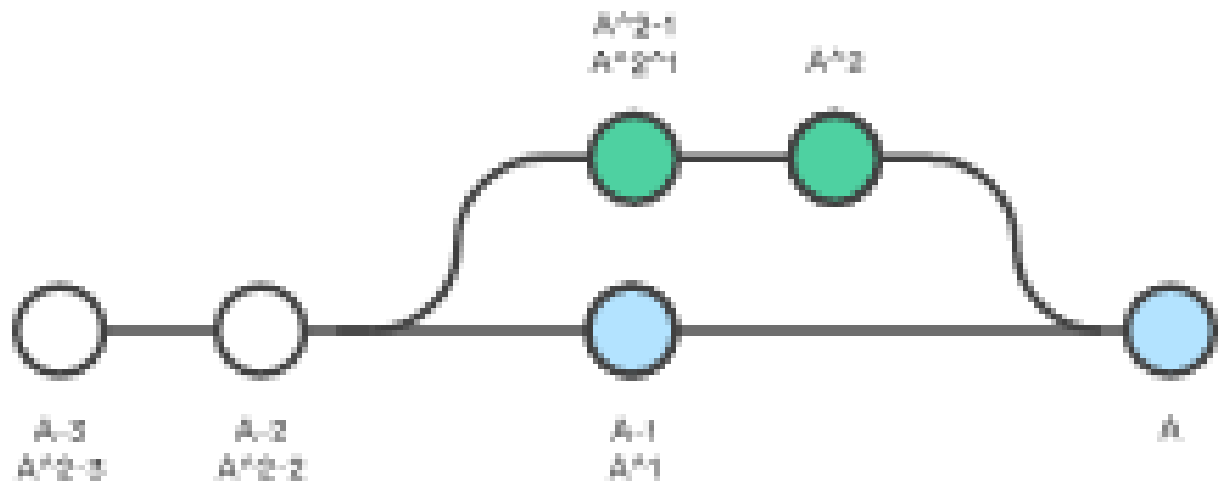
- Special refs
  - There are a few special refs that can be used as well:
    - HEAD – current checked out branch / commit
    - FETCH_HEAD – most recent fetched branch from a remote
    - ORIG_HEAD – backup of HEAD before major changes
    - MERGE_HEAD – commit(s) being merged
    - CHERRY_PICK_HEAD – commit being cherry-picked
  - These are all automatically managed by git
  - Usually only HEAD is used directly in git commands

Refs & Reflog

- Relative refs
  - A powerful tool is the ability to refer to commits relative to another commit
  - The tilde (~) character is used to refer to parent commits
    - `git show HEAD~2` refers to the grandparent of HEAD
  - This gets a little more complicated when there are merge commits in the ancestry, since they have 2 (or more) parents
  - For 3-way merges, the first parent is always the branch that was checked out when performing the merge, and second parent is the branch that was passed to "git merge"
  - ~ will always traverse ancestors through the first parent

Refs & Reflog

- Relative refs
  - ^ can be used to traverse a different parent
    - `git show HEAD^2` refers to the second parent of HEAD
  - ^ can be used multiple times to move more generations
    - `git show HEAD^2^1`



Refs & Reflog

- The Reflog
  - The Reflog is Git's safety net
  - It records changes made to the repository
  - It's a chronological history of everything you've done to your local repo
  - Viewing the reflog:
    - `git reflog`
  - `HEAD{<n>}` refers to an entry in the reflog, as shown in the reflog output, and can be used to revert to state that would be otherwise lost (e.g. `git checkout HEAD{1}`)

Refs & Reflog

- The Reflog
  - `HEAD{<n>}` refers to an entry in the reflog, as shown in the reflog output

Submodules

trainologic

Submodules

- The motivation
  - It often happens that while working on a project, you need to use another project (e.g. a library) within the current one
  - You can “install” that “other project” using whatever method is available for it (like installing a Ruby Gem)
    - Impossible to customize & modify
    - More difficult to deploy and manage across a team
  - You can copy the source code of that “other project” into a subdirectory of the current project
    - Difficult to manage local customizations together with upstream changes

Submodules

- The solution using submodules
  - Submodules allow you to keep a Git repository as a subdirectory of another Git repository
  - `git submodule add <other-repo-url> <sub-dir>`
  - Submodules configuration is stored in a "`.gitmodules`" file that should be committed with the rest of the project
  - The subdirectory entry has a special mode, indicating it's a "placeholder" for a directory

Submodules

- Cloning a repository with submodules
  - When cloning, subdirs for the submodules are created empty
  - Every contributor will need to run 2 commands:
    - `git submodule init`
      - To initialize the local configuration file
    - `git submodule update`
      - To fetch data from the other projects
  - Tip: running git clone with `--recurse-submodules` takes care of init & update automatically

Submodules

- Working with submodules – pulling upstream changes
  - Simple model of just "consuming" another project, updating it from time to time
  - To update – cd into the subdir, fetch & merge the upstream
  - This will update the submodule definition, which you'll need to commit and push so others get the update
  - Tip: `git submodule update --remote <subdir>` updates the submodule without needing to manually fetch & merge

Submodules

- Working with submodules – making customizations
  - A submodule is checked out in a detached HEAD state, so you can't make new commits without risking losing them on the next "update"
  - To hack on a submodule, you will need to:
    - Check out a branch to work on
    - Tell git whether to merge or rebase your commits whenever new commits are pulled from the upstream:

```
git submodule update --remote --merge <subdir>
git submodule update --remote --rebase <subdir>
```

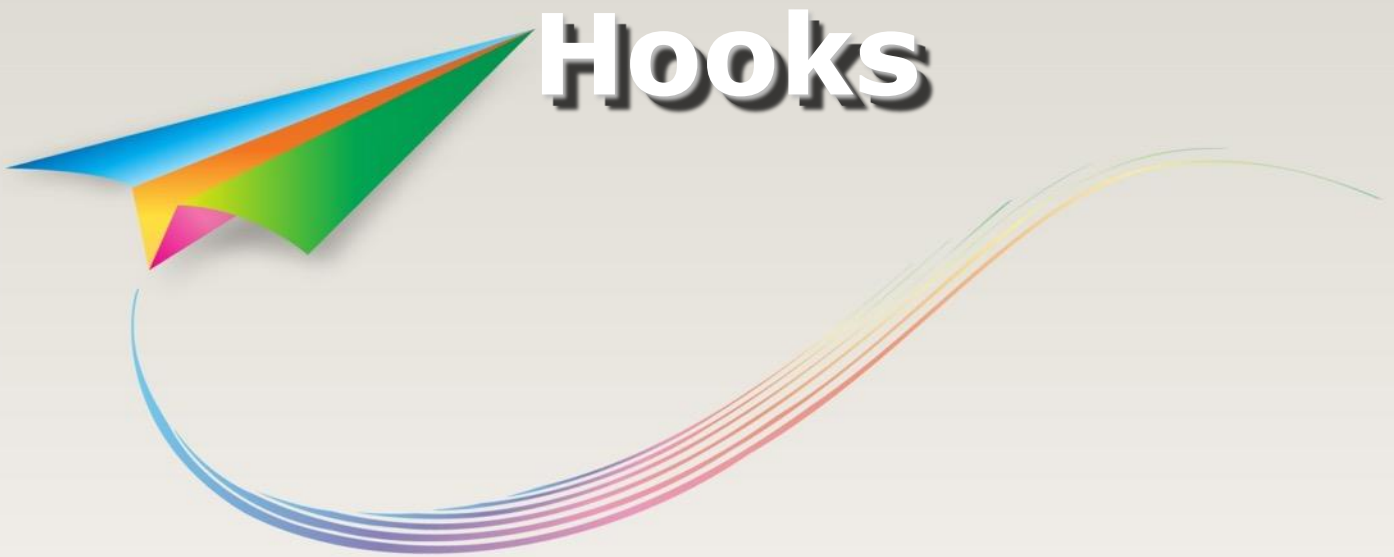
      - By default, Git will reset to the detached HEAD state

Submodules

- Working with submodules – publishing customizations
  - git push takes a –recurse-submodule option that can be either:
    - "check" makes the push fail if there local submodule commits that were not pushed
    - "on-demand" automatically pushes any local submodule commits (equivalent to cd'ing into the subdirs and pushing)
  - Publishing customizations requires write access to the upstream
  - Often, projects submodule a "private fork" of a 3rd party project so they can have such write access

Submodules

- Challenges
  - It’s not possible to merge divergent branches that both changed a submodule
  - Switching between branches with & without submodules is tricky
  - Switching or merging branches that replace a tracked subdirectory with a submodule with the same name is difficult



Hooks

trainologic

Git Hooks

- What are Git hooks?
  - Scripts that Git executes before or after certain events
  - Hooks can be client side or server side
  - Example events: commit, push, receive
  - Example use-cases:
    - pre-commit: check commit message before committing
    - post-receive: trigger an automated deployment

Git Hooks

- Installing Git hooks
  - In every Git repository, under the "`.git`" directory, there's a "`hooks`" directory
  - By default, this directory contains example scripts for the available hooks (like "`commit-msg.sample`")
  - To install a hook, just put an executable file with the correct name (like "`commit-msg`") under "`.git/hooks`"
  - It can be any kind of executable – shell script, Python, Ruby, etc.
  - Hooks must be installed by every contributor in every repository to be effective – they are not part of the repository

Client Side Hooks

- Committing-workflow hooks
  - pre-commit runs first before even typing a commit message
  - prepare-commit-msg runs before the commit message editor is opened, and after a default message is created
  - commit-msg runs before committing, and gets a path to a temp file with the commit message that was written
  - post-commit runs after the commit process completes
  - Whenever these hooks exit with non-zero code – Git aborts the commit

Client Side Hooks

- Other client hooks
  - pre-rebase runs before starting a rebase, and can be used to halt the process
  - post-rewrite runs during commands that replace commits
  - post-checkout runs after a successful checkout, and can be used to assist in setting up the project environment
  - post-merge runs after a successful merge command
  - pre-push runs during git push before transferring objects
  - pre-auto-gc runs just before a garbage collection takes place

Server Side Hooks

- Server hooks
  - pre-receive handles pushes from clients, and can be used t reject pushes based on policies
  - update is similar to pre-receive, but runs once for every pushed branch
  - post-receive runs after the entire process completes, and can be used to notify others about the push
  - Naturally, server-side hooks require administrator access to the Git server

Exercise #3

trainologic