ADVANCED GIT
NINA ZAKHARENKO

 @NNJA

frontendmasters.com/workshops/git-indepth/

git.io/advanced-git

HI

- who am I?
- @nnja

nina@nnja.io

REQUIREMENTS

- command line that supports unix style commands
- git version > 2.0
- github.com account
- clone git.io/advanced-git for slides and exercises
- (osx only) install homebrew - https://brew.sh/


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

WAT?

This is NOT real git documentation! Read carefully, and click the button to generate a new man page.

git-man-page-generator.lokaltog.net

git-acquire-pack(1) Manual Page

Permalink | Generate new man page

NAME

`git-acquire-pack` — acquire the non-remoted applied packs to some blamed unstaged logs

SYNOPSIS

```
git-acquire-pack [ --handle-manage-remote ] [ --scoop-base | --hail-commit | --induce-index ]
```

DESCRIPTION

`git-acquire-pack` acquires a few downstream packs from various grepped tips, and the object to be cloned can be specified in several ways.

PORCELAIN VS. PLUMBING

CLASS OVERVIEW – PART 1

- What is Git?
- Working Area, Staging Area, Repository
- Staging Area Deep Dive
- References, Commits, Branches
- Stashing

CLASS OVERVIEW – PART 2

- Merging + Rebasing
- History + Diffs
- Fixing Mistakes
- Rebase
- Forks, Remote Repositories

CLASS OVERVIEW – PART 3

- The Danger Zone
- Advanced Tools
- Customization - Config, Ignore, Hooks, Templates
- Social Git: Integrating GitHub with CI Tools
- Advanced GitHub: Working with the GitHub API

WHY USE THE COMMAND LINE?

WHAT IS GIT?

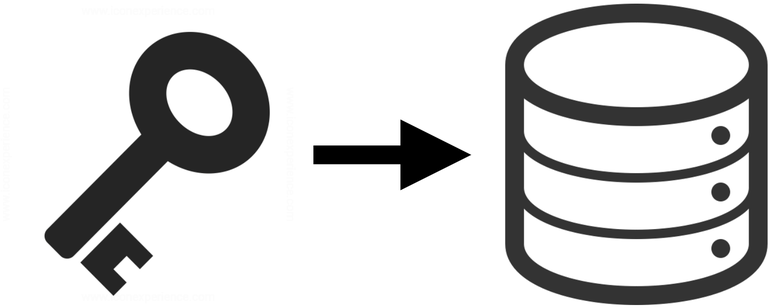
Distributed Version Control System

HOW DOES GIT STORE INFORMATION?

- At its core, git is like a key value store.

- The **Value = Data**
- The **Key = Hash of the Data**

- You can use the key to retrieve the content.

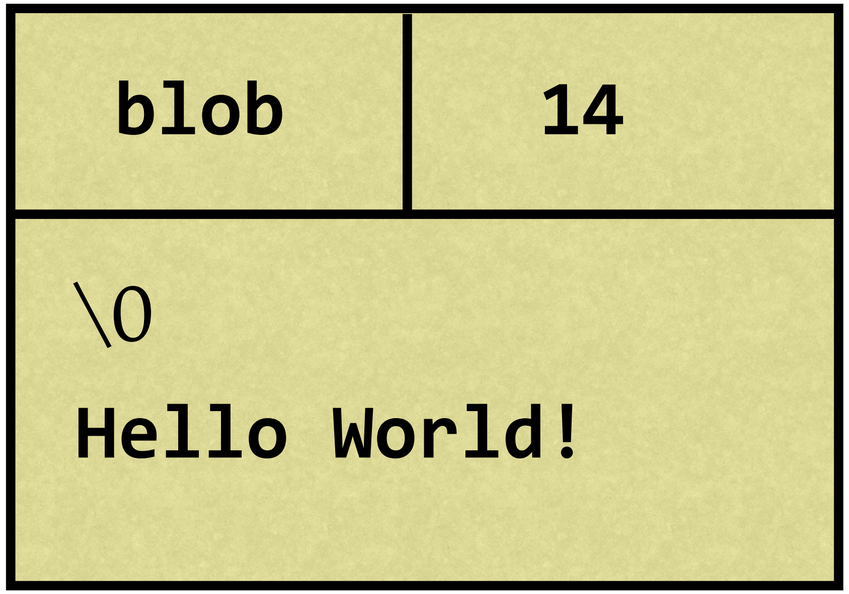


THE KEY – SHA1

- Is a cryptographic hash function.
- Given a piece of data, it produces a 40-digit hexadecimal number.
- This value should always be the same if the given input is the same.

THE VALUE – BLOB

- git stores the ***compressed*** data in a blob, along with metadata in a header:
  - the identifier **`blob`**
  - the size of the content
  - \0 delimiter
  - content



UNDER THE HOOD - GIT HASH-OBJECT

Asking Git for the SHA1 of contents:

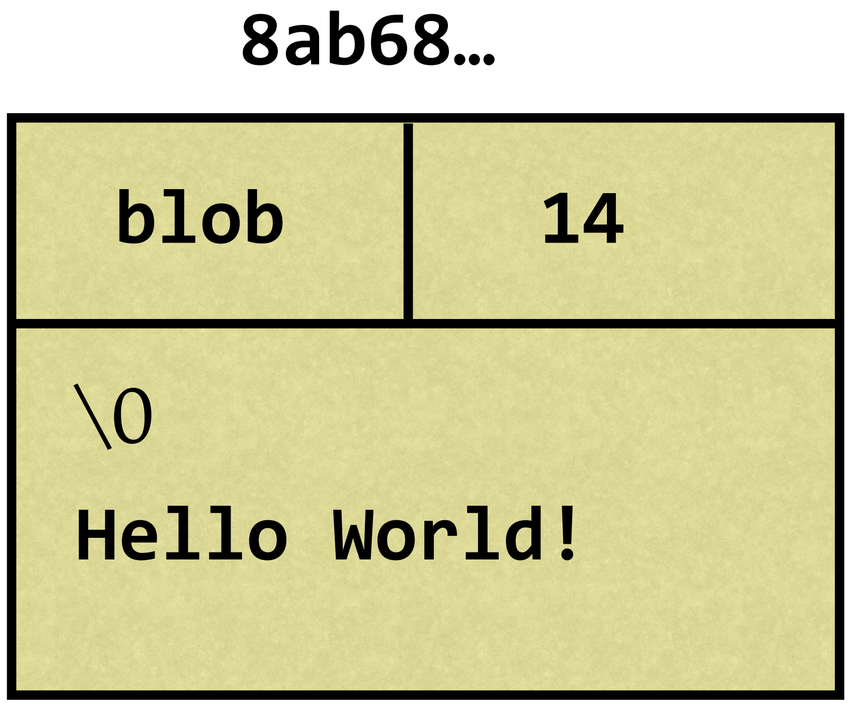
```
❯ echo 'Hello, World!' | git hash-object --stdin
8ab686eafeb1f44702738c8b0f24f2567c36da6d
```

Generating the SHA1 of the contents, with metadata:

```
❯ echo 'blob 14\0Hello, World!' | openssl sha1
8ab686eafeb1f44702738c8b0f24f2567c36da6d
```

It's a Match!

THE VALUE – BLOB



WHERE DOES GIT STORE ITS DATA?

```
~/projects/sample
❯ git init
Initialized empty Git repository in /Users/nina/projects/sample/.git/
```

.git directory contains data about our repository

WHERE ARE BLOBS STORED?

```
❯ echo 'Hello, World!' | git hash-object -w --stdin
8ab686eafeb1f44702738c8b0f24f2567c36da6d
```

-w flag means write

WHERE ARE BLOBS STORED?

```
❯ echo 'Hello, World!' | git hash-object -w --stdin
8ab686eafeb1f44702738c8b0f24f2567c36da6d

❯ rm -rf .git/hooks
```

we don’t need this for now

WHERE ARE BLOBS STORED?

```
❯ echo 'Hello, World!' | git hash-object -w --stdin
8ab686eafeb1f44702738c8b0f24f2567c36da6d

❯ rm -rf .git/hooks

❯ tree .git
.git
├── HEAD
├── config
├── description
├── info
│   └── exclude
├── objects
│   ├── 8a
│   │   └── b686eafeb1f44702738c8b0f24f2567c36da6d
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

our blob is stored in objects

the directory starts with the first 2 chars of the hash.

WE NEED SOMETHING ELSE

The blob is missing information!

- filenames
- directory structures

Git stores this information in a **tree**.

TREE

a **`tree`** contains pointers (using SHA1):

- to blobs
- to other trees

and metadata:

- *type* of pointer (**`blob`** or **`tree`**)
- *filename* or directory name
- *mode* (executable file, symbolic link, …)

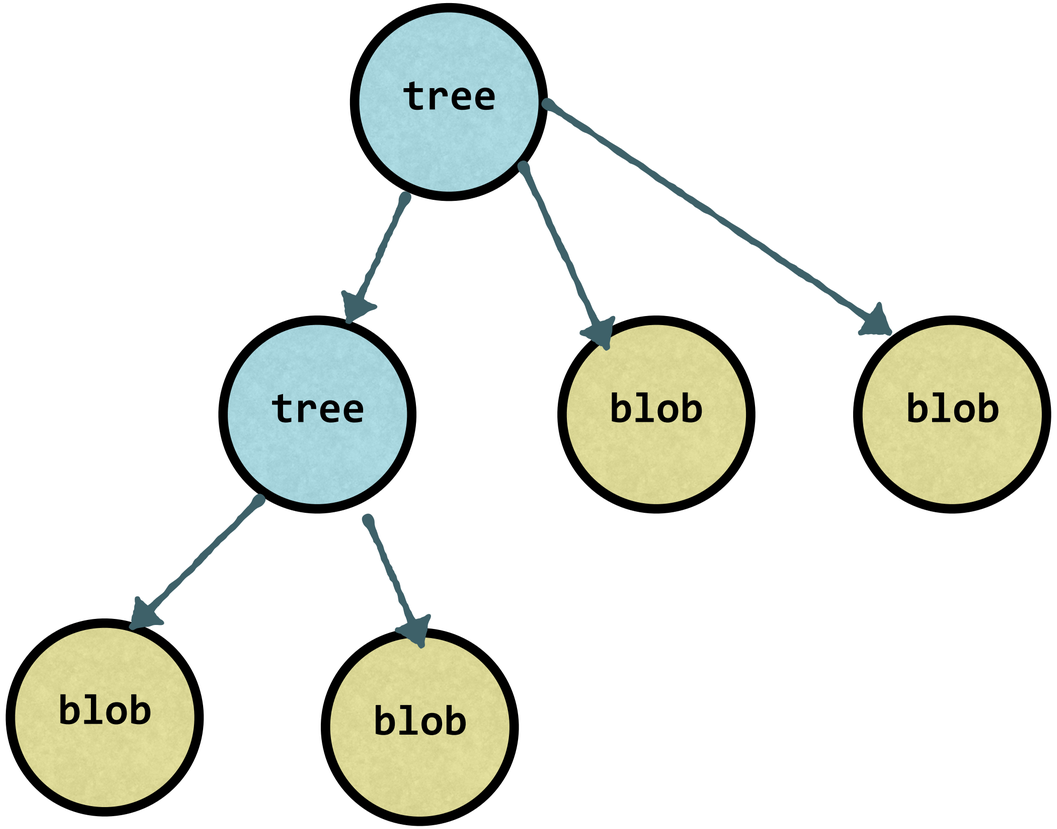
TREE

```
.
├── copies
│   └── hello-copy.txt
└── hello.txt

1 directory, 2 files
```

4ad20…

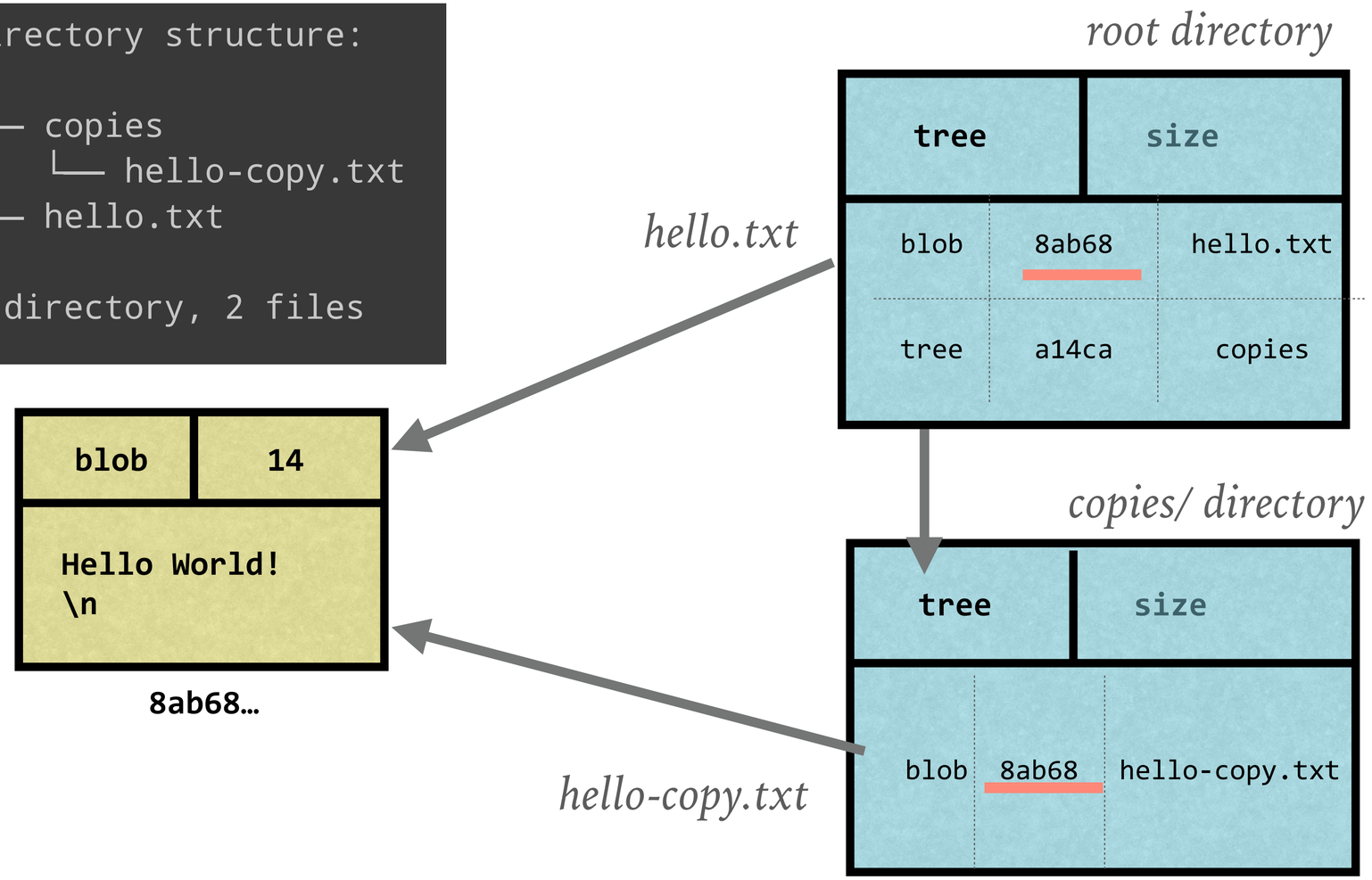
tree	**<size>**	
\0		
blob	8ab68	hello.txt
tree	a14ca	copies

TREES POINT TO BLOBS & TREES



IDENTICAL CONTENT IS ONLY STORED ONCE!

```
Directory structure:
.
├── copies
│   └── hello-copy.txt
└── hello.txt

1 directory, 2 files
```



OTHER OPTIMIZATIONS – PACKFILES, DELTAS

- Git objects are compressed
- As files change, their contents remain mostly similar.
- Git optimizes for this by compressing these files together, into a Packfile
- The Packfile stores the object, and “deltas”, or the differences between one version of the file and the next.

- Packfiles are generated when:
  - you have too many objects, during gc, or during a push to a remote

COMMITS

COMMIT OBJECT

a **`commit`** points to:

- a tree

and contains metadata:

- *author and committer*
- *date*
- *message*
- *parent commit (one or more)*

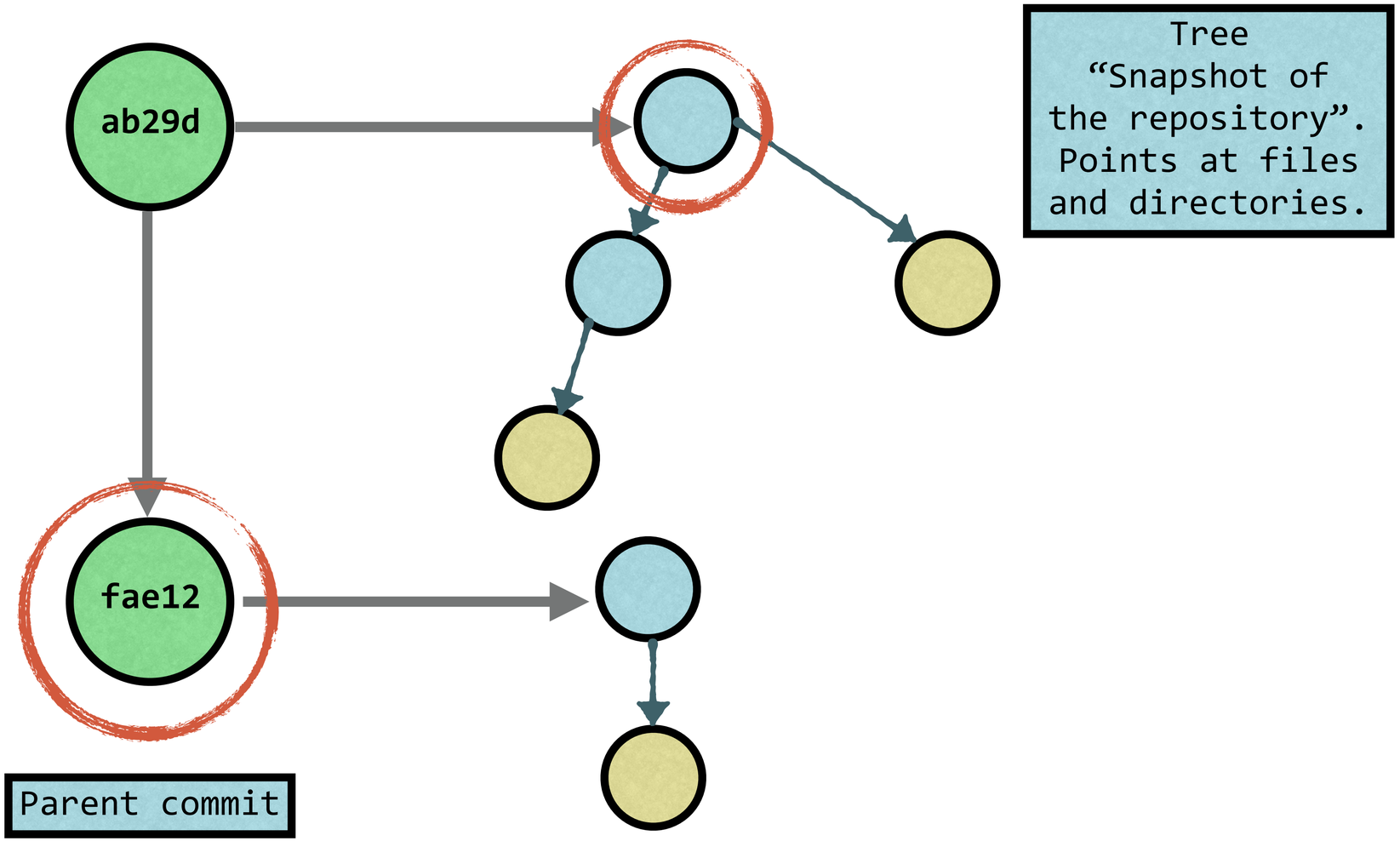
the SHA1 of the commit is the hash of all this information

COMMIT

fae12…

commit	size
tree	8ab68
parent	a14ca
author	Nina
message	“Initial Commit”

COMMITS POINT TO PARENT COMMITS AND TREES



A COMMIT IS A CODE SNAPSHOT

COMMITS UNDER THE HOOD – MAKE A COMMIT

```
❯ echo 'Hello World!' > hello.txt

❯ git add hello.txt

❯ git commit -m "Initial commit"
[master (root-commit) adf0e13] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 hello.txt
```

COMMITS UNDER THE HOOD – LOOK IN .GIT/OBJECTS

```
❯ tree .git/objects
.git/objects
├── 58
│   └── 1caa0fe56cf01dc028cc0b089d364993e046b6
├── 98
│   └── 0a0d5f19a64b4b30a87d4206aade58726b60e3
├── ad
│   └── f0e13658d9b4424efe03c3ac3281facf288c13
├── info
└── pack
```

COMMITS UNDER THE HOOD – LOOKING AT OBJECTS

```
❯ cat .git/objects/98/0a0d5f19a64b4b30a87d4206aade58726b60e3
xKOR04fH/IQHak%
```

oops!
remember, git objects are **compressed.**

GIT CAT-FILE -T (TYPE) AND -P (PRINT THE CONTENTS)

```
❯ git cat-file -t 980a0
blob


❯ git cat-file -p 980a0
Hello World!
```

-t
print the **type**

-p
print the **contents**

GIT CAT-FILE -T (TYPE) AND -P (PRINT THE CONTENTS)

```
❯ git cat-file -t 581caa
tree

❯ git cat-file -p 581caa
100644 blob 980a0d5f19a64b4b30a87d4206aade58726b60e3
hello.txt
```

GIT CAT-FILE -T (TYPE) AND -P (PRINT THE CONTENTS)

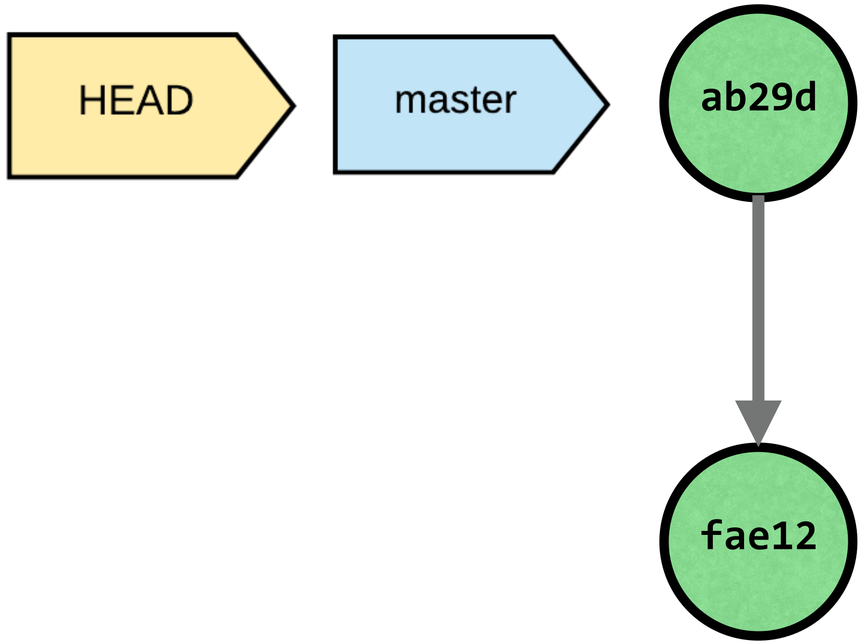
```
❯ git cat-file -t adf0e1
commit

❯ git cat-file -p adf0e1
tree 581caa0fe56cf01dc028cc0b089d364993e046b6
author Nina <nina@nnja.io> 1506214053 -0700
committer Nina <nina@nnja.io> 1506214053 -0700

Initial commit
```

WHY CAN'T WE "CHANGE" COMMITS?

- If you change any data about the commit, the commit will have a new SHA1 hash.
- Even if the files don't change, the created date will.

REFERENCES – POINTERS TO COMMITS

- Tags
- Branches
- HEAD - a pointer to the current commit



REFERENCES - UNDER THE HOOD

```
❯ tree .git
.git
├── HEAD
└── refs
    ├── heads
    │   └── master
    └── tags
```

refs/heads is where branches live

REFERENCES – UNDER THE HOOD

```
❯ tree .git
.git
├── HEAD
└── refs
    ├── heads
    │   └── master
    └── tags


❯ git log --oneline
adf0e13 (HEAD -> master) Initial commit


❯ cat .git/refs/heads/master
adf0e13658d9b4424efe03c3ac3281facf288c13
```

/refs/heads/master contains which commit the branch points to

REFERENCES – UNDER THE HOOD

```
❯ tree .git
.git
├── HEAD
└── refs
    ├── heads
    │   └── master
    └── tags


❯ git log --oneline
adf0e13 (HEAD -> master) Initial commit


❯ cat .git/refs/heads/master
adf0e13658d9b4424efe03c3ac3281facf288c13


❯ cat .git/HEAD
ref: refs/heads/master
```

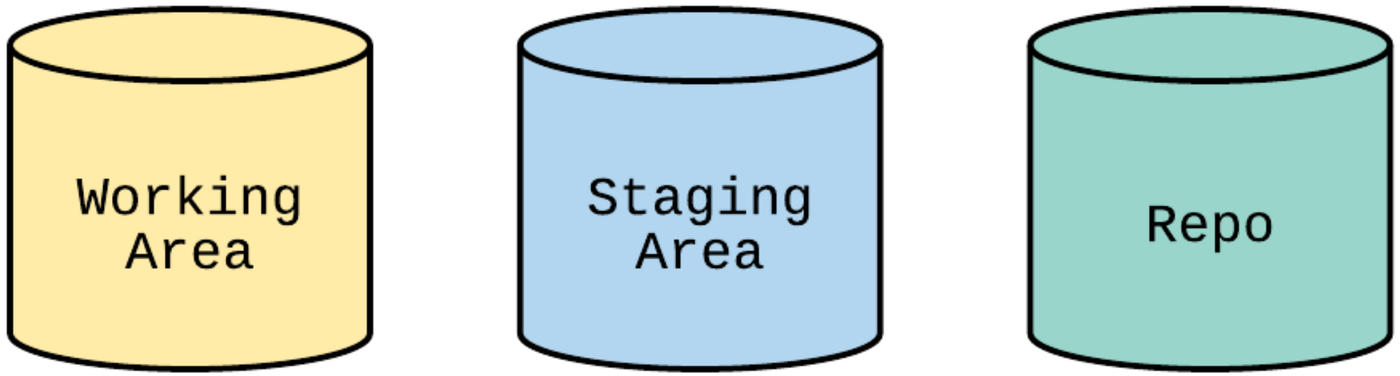
HEAD is *usually* a pointer to the current branch

EXERCISE

1. Configure your editor
   - ➤ git.io/config-editor
2. Complete exercise
   - ➤ https://github.com/nnja/advanced-git/blob/master/exercises/Exercise1-SimpleCommit.md
3. Check out cheat sheet for using less
   - ➤ git.io/use-less

THREE AREAS WHERE CODE LIVES

1. Working Area
2. Staging Area
3. Repository



THE WORKING AREA

- The files in your working area that are *also not* in the staging are are not handled by git.
- Also called **untracked files**

THE STAGING AREA (A.K.A INDEX, CACHE)

- What files are going to be part of the next commit
- The staging area is how git knows what will change between the current commit and the next commit.

THE REPOSITORY

- The files git knows about!
- Contains all of your commits

CLOSER LOOK: THE STAGING AREA

THE STAGING AREA

- The staging area is how git knows what will change between the current commit and the next commit.

- Tip: a “clean” staging area isn’t empty!
- Consider the baseline staging area as being an exact copy of the latest commit.

```
› git ls-files -s
100644 b8b2583b242add97d513d8d8b85a46b9b5fb29cb 0 index.txt
100644 ed52afd72f64b1f8575e81bb4cb02ef1215739f6 0 posts/
welcome.txt
```

The plumbing command ls-files -s will show you what’s in the staging area.

MOVING FILES IN & OUT OF THE STAGING AREA

- add a file to the next commit:
  - `git add <file>`

- delete a file in the next commit:
  - `git rm <file>`

- rename a file in the next commit:
  - `git mv <file>`

GIT ADD -P

- one of my favorite tools
- allows you to stage commits in hunks
- interactively!

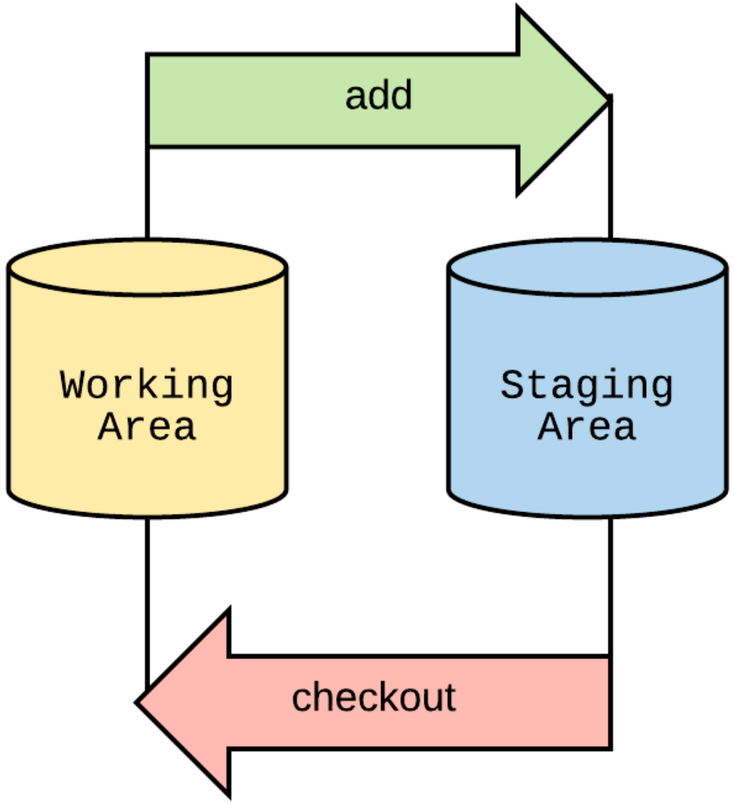
- It’s especially useful if you’ve done too much work for one commit.

USING GIT ADD -P

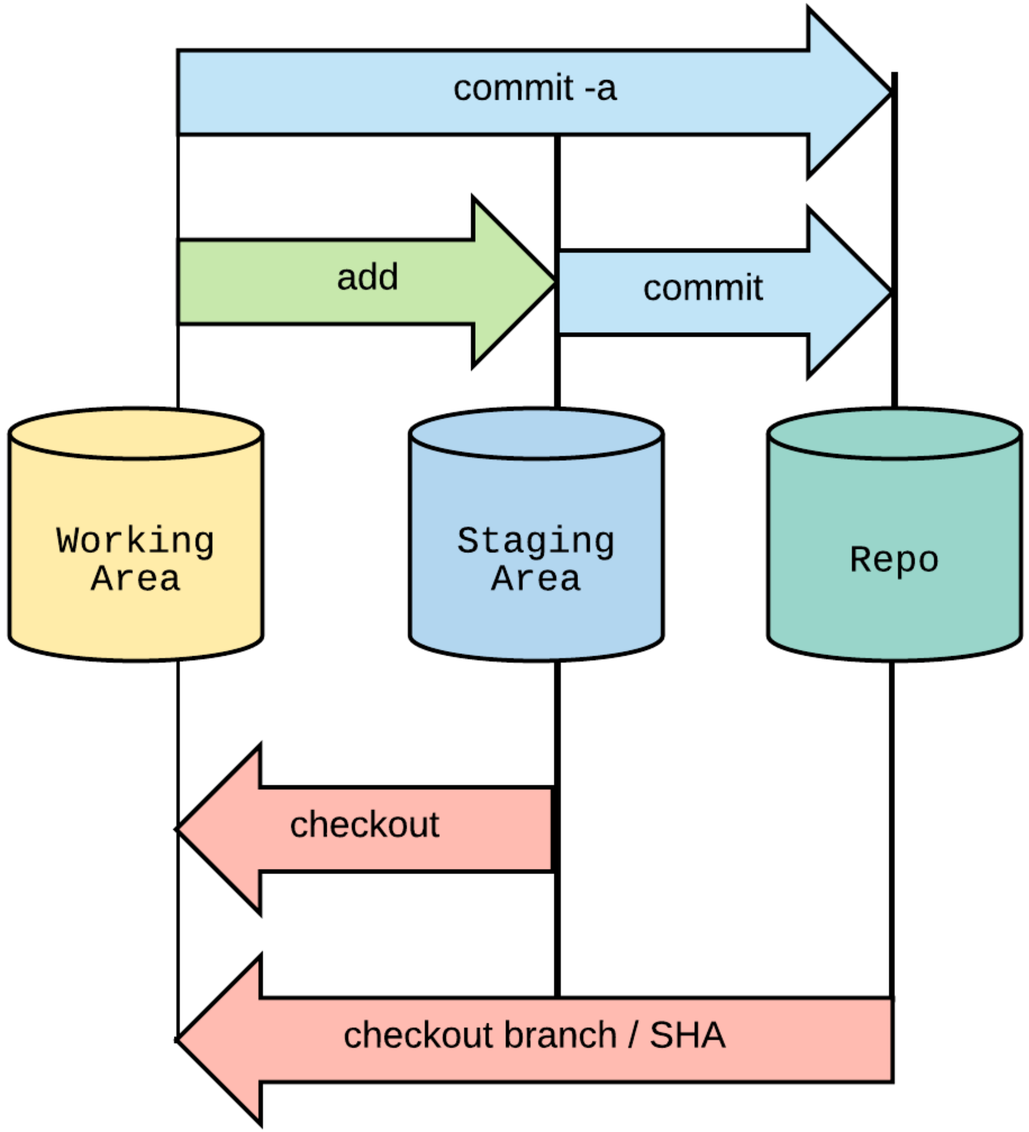
```
❯ git add -p

.
.
.
diff displayed
.
.
.

Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

“UNSTAGE” FILES FROM THE STAGING AREA

- Not removing the files.
- You’re replacing them with a copy that’s currently in the repository.



BASICS: HOW CONTENT MOVES IN GIT



STASHING

GIT STASH

- Save un-committed work.
- The stash is **safe** from destructive operations.

GIT STASH – BASIC USE

- stash changes
  - `git stash`
- list changes
  - `git stash list`
- show the contents
  - `git stash show stash@{0}`
- apply the last stash
  - `git stash apply`
- apply a specific stash
  - `git stash apply stash@{0}`

ADVANCED STASHING – KEEPING FILES

- Keep untracked files
  - `git stash --include-untracked`
- Keep all files (even ignored ones!)
  - `git stash --all`

ADVANCED STASHING - OPERATIONS

- Name stashes for easy reference
  - `git stash save "WIP: making progress on foo"`

- Start a new branch from a stash
  - `git stash branch <optional stash name>`

    *git stash branch <branch-name> <optional stash name>

- Grab a single file from a stash
  - `git checkout <stash name> -- <filename>`

ADVANCED STASHING – CLEANING THE STASH

- Remove the last stash and applying changes:
  - `git stash pop`
  - tip: doesn't remove if there's a merge conflict
- Remove the last stash
  - `git stash drop`
- Remove the nth stash
  - `git stash drop stash@{n}`
- Remove ***all*** stashes
  - `git stash clear`

EXAMINING STASH CONTENTS - GIT SHOW

```
❯ git stash list
stash@{0}: WIP on master: 9703ef3 example
stash@{1}: WIP on example3: 0be5e31 hello
stash@{2}: WIP on example3: 0be5e31 a commit

❯ git stash show stash@{2}

 goodbye.txt  | 1 +
 hello.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)
```

GIT STASH – WHAT I USE

- Keep untracked files
  - `git stash --include-untracked`

- Name stashes for easy reference
  - `git stash save "WIP: making progress on foo"`

    Just like git add -p, we can also use git stash -p

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise2-StagingAndStashing.md

REFERENCES

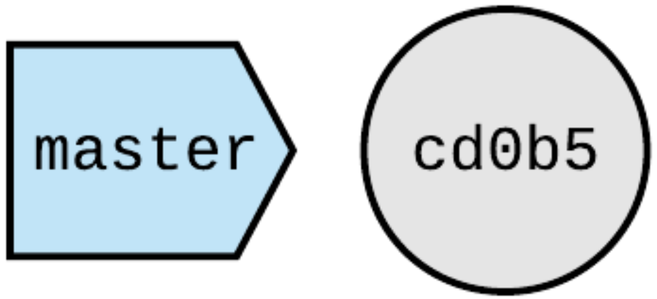
(pointers to commits)

THREE TYPES OF GIT REFERENCES

- Tags & Annotated Tags
- Branches
- HEAD

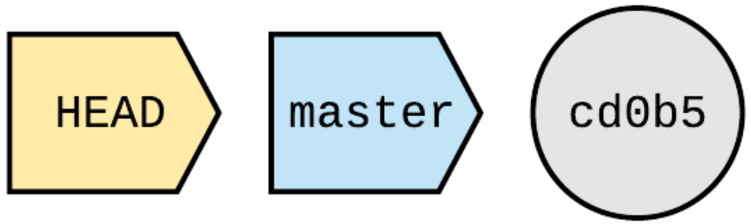
WHAT'S A BRANCH?

- A branch is just a pointer to a particular commit.
- The pointer of the current branch changes as new commits are made.



WHAT IS HEAD?

- **HEAD** is how git knows what branch you're currently on, and what the next parent will be.
- It's a pointer.
  - It usually points at the ***name*** of the current branch.
  - But, it can point at a commit too (detached HEAD).
- It moves when:
  - You make a commit in the currently active branch
  - When you checkout a new branch



HEAD

```
❯ git checkout master

❯ cat .git/HEAD
ref: refs/heads/master

❯ git checkout feature
Switched to branch 'feature'

❯ cat .git/HEAD
ref: refs/heads/feature
```

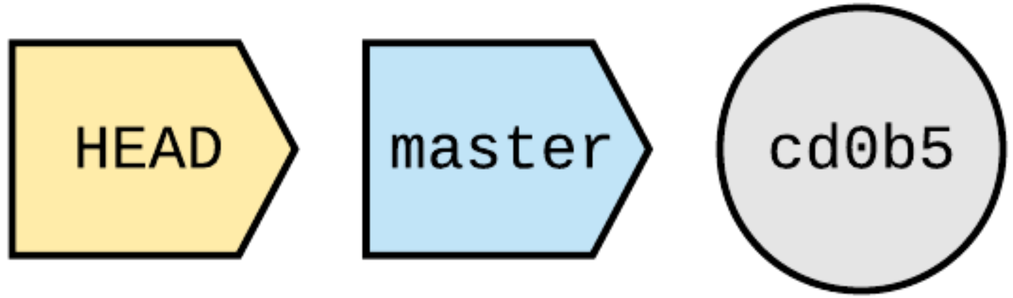
SAMPLE REPO – A SIMPLE BLOG

- We're working on a simple blog.
- **index.txt** contains links to our entries, and post titles
- the **posts/** directory contains our blog entries.

```
❯ mkdir posts

❯ echo 'This is my very first blog entry.' > posts/welcome.txt

❯ echo 'welcome.txt Welcome to my blog' > index.txt
```

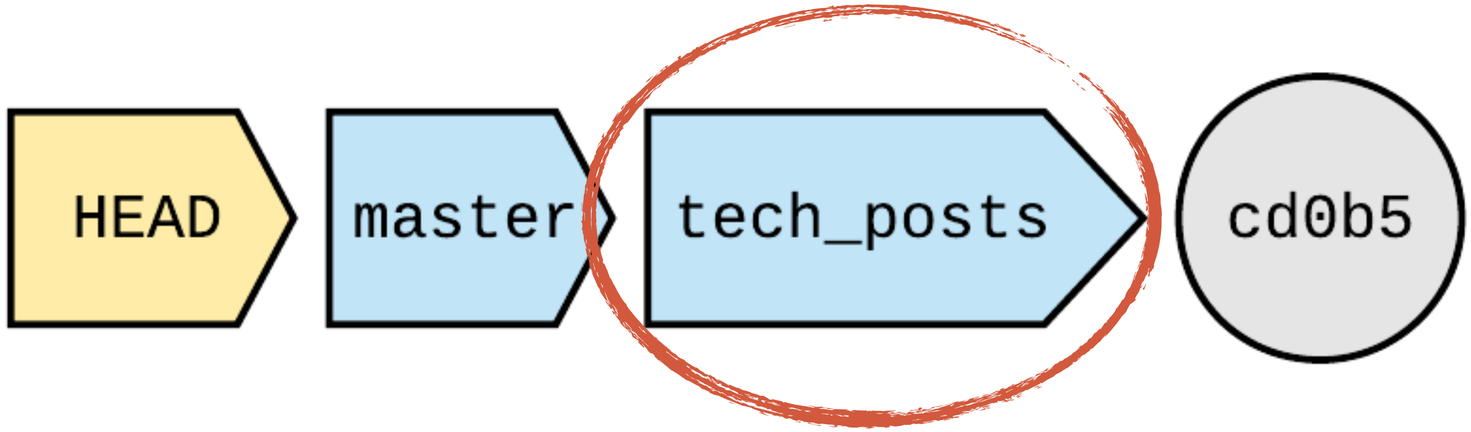
CURRENT REPO STATE

```
› git init

› git add .

› git commit -m "Initial commit"
[master (root-commit) cd0b57c] Initial commit
 2 files changed, 2 insertions(+)
 create mode 100644 index.txt
 create mode 100644 posts/welcome.txt
```



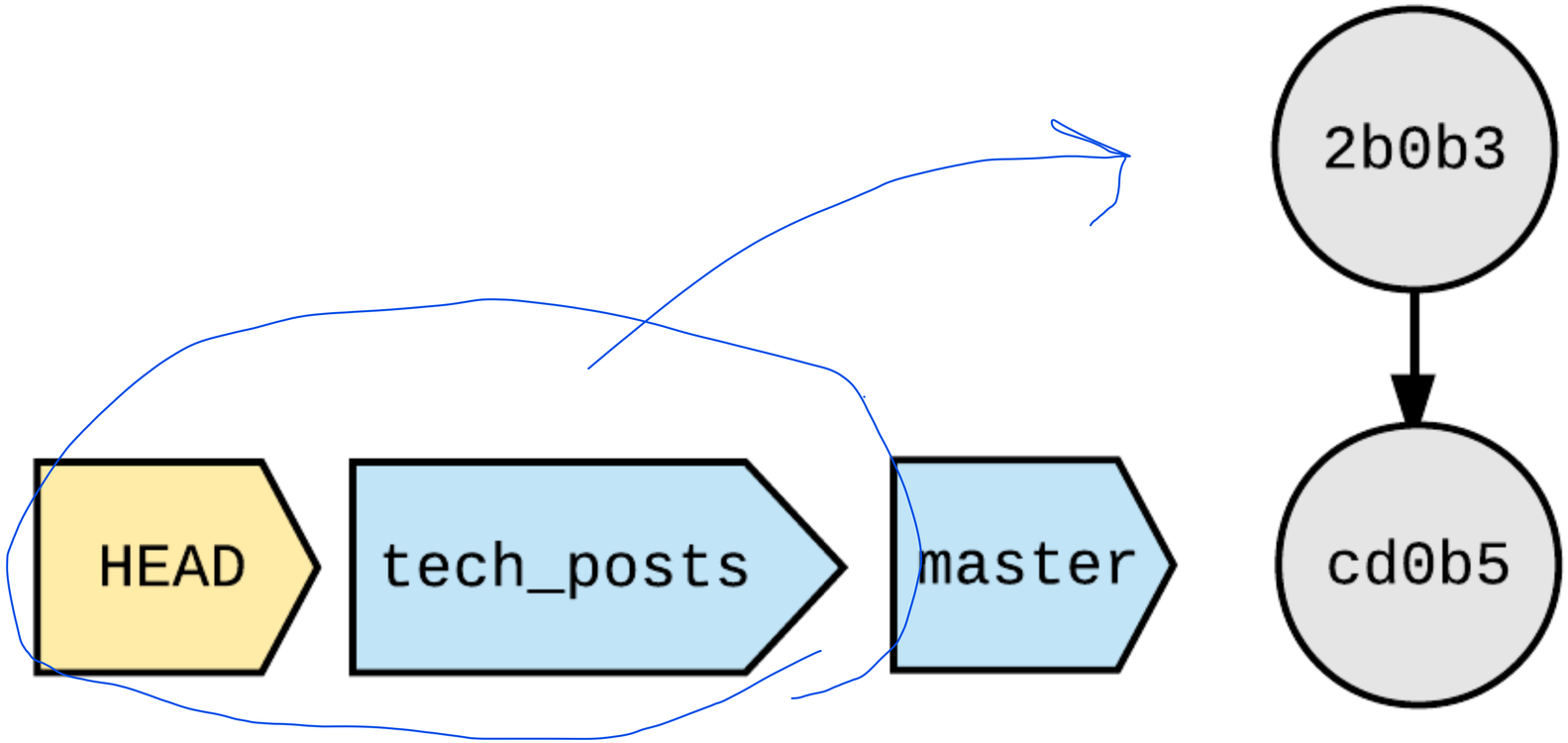
CURRENT REPO STATE

```
❯ git checkout -b tech_posts
Switched to a new branch 'tech_posts'
```



CURRENT REPO STATE

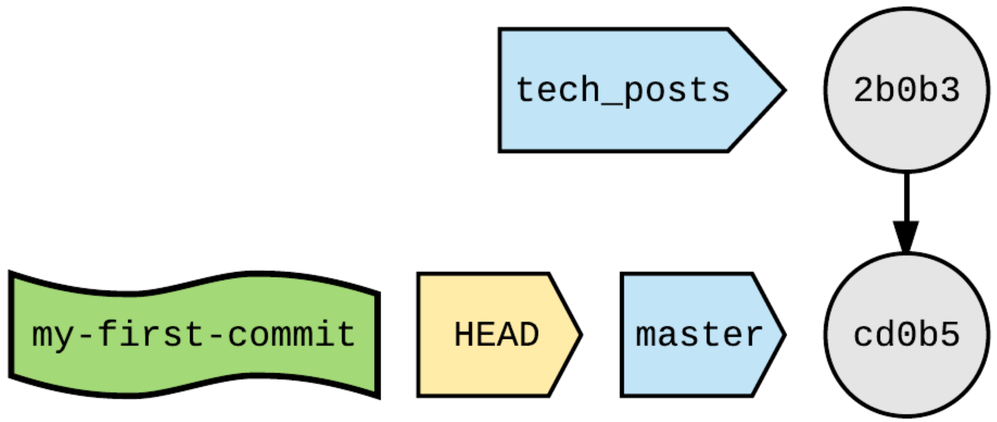
```
❯ git add posts/python.txt
❯ git commit -m "New blog post about python"
[tech_posts 2b0b3f2] New blog post about python
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 posts/python.txt
```



LIGHTWEIGHT TAGS

- Lightweight tags are just a simple pointer to a commit.
- When you create a tag with no arguments, it captures the value in HEAD

```
❯ git checkout master
Switched to branch 'master'

❯ git tag my-first-commit
```



ANNOTATED TAGS: GIT TAG -A

- Point to a commit, but store additional information.
  - Author, message, date.

```
❯ git tag -a v1.0 -m "Version 1.0 of my blog"

❯ git tag
my-first-commit
v1.0

❯ git show v1.0
tag v1.0
Tagger: Nina Zakharenko <nina@nnja.io>
Date:   Sun Sep 24 17:01:21 2017 -0700

Version 1.0 of my blog
```

WORKING WITH TAGS

- List all the tags in a repo
  - `git tag`
- List all tags, and what commit they're pointing to
  - `git show-ref --tags`
- List all the tags pointing at a commit
  - `git tag --points-at <commit>`
- Looking at the tag, or tagged contents:
  - `git show <tag-name>`

TAGS & BRANCHES

- **Branch**
  - The current branch pointer moves with every commit to the repository

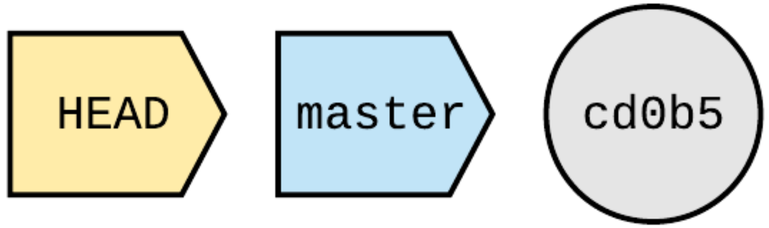
- **Tag**
  - The commit that a tag points doesn’t change.
  - It’s a snapshot!

HEAD-LESS / DETACHED HEAD

- Sometimes you need to checkout a specific commit (or tag) instead of a branch.
- git moves the HEAD pointer to that commit
- as soon as you checkout a different branch or commit, the value of HEAD will point to the new SHA
- There is no reference pointing to the commits you made in a detached state.

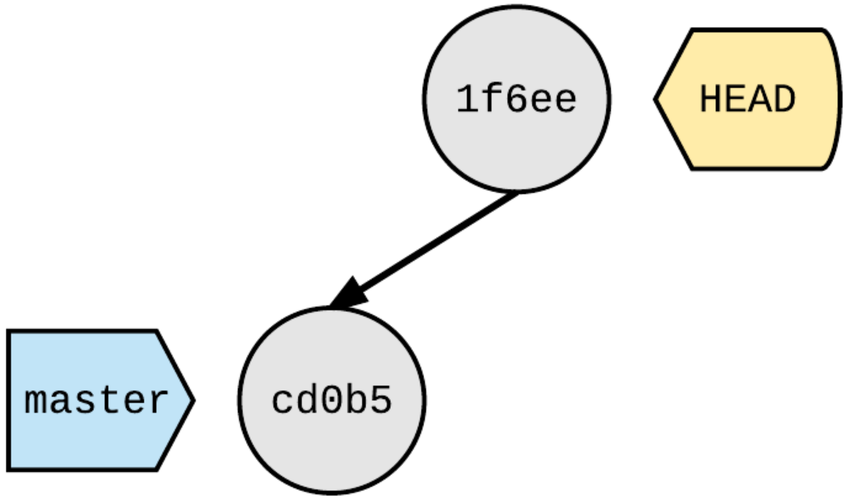
HEAD-LESS / DETACHED HEAD

```
❯ git checkout cd0b57
Note: checking out 'cd0b57'.

You are in 'detached HEAD' state. You can
look around, make experimental
changes and commit them, and you can
discard any commits you make in this
state without impacting any branches by
performing another checkout.
```



HEAD-LESS / DETACHED HEAD

```
❯ git checkout cd0b57
Note: checking out 'cd0b57'.

You are in 'detached HEAD' state. You
can look around, make experimental
changes and commit them, and you can
discard any commits you make in this
state without impacting any branches by
performing another checkout.

❯ git add posts/second-post.txt

❯ git commit -m "My second post"
[detached HEAD 1f6ee83] My second post
 1 file changed, 1 insertion(+)
 create mode 100644 posts/second-
post.txt
```



HEAD-LESS / DETACHED HEAD

Save your work:

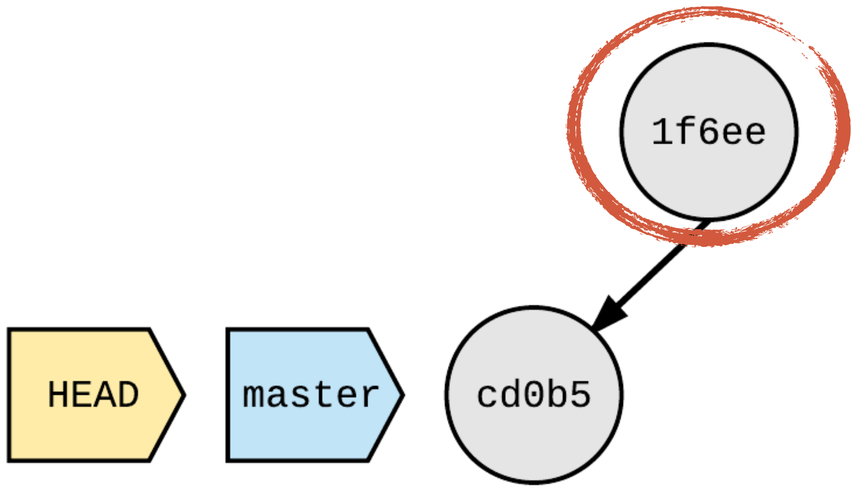
- Create a new branch that points to the last commit you made in a detached state.
  - `git branch <new-branch-name> <commit>`
- Why the last commit?
  - Because the other commits point to their parents.

DANGLING COMMITS

Discard your work:

- If you don't point a new branch at those commits, they will no longer be referenced in git. (dangling commits)
- Eventually, they will be garbage collected.

```
❯ git checkout master
Warning: you are leaving 1 commit behind,
not connected to
any of your branches:

  1f6ee83 My second post
```



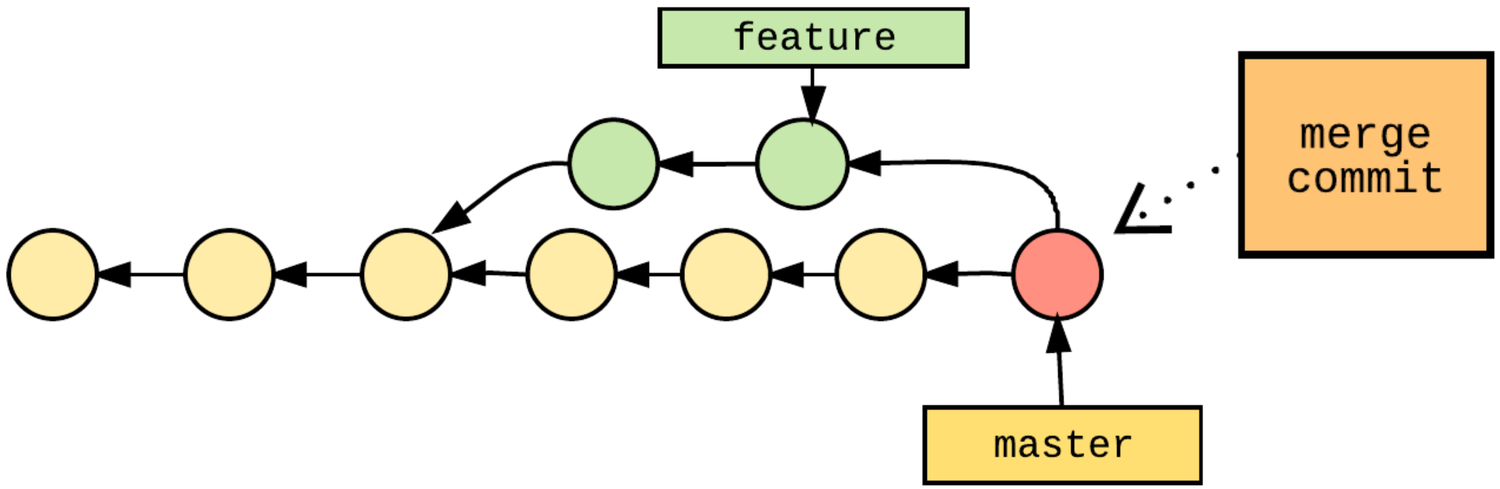
EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise3-References.md

MERGING

UNDER THE HOOD – MERGE COMMITS ARE JUST COMMITS

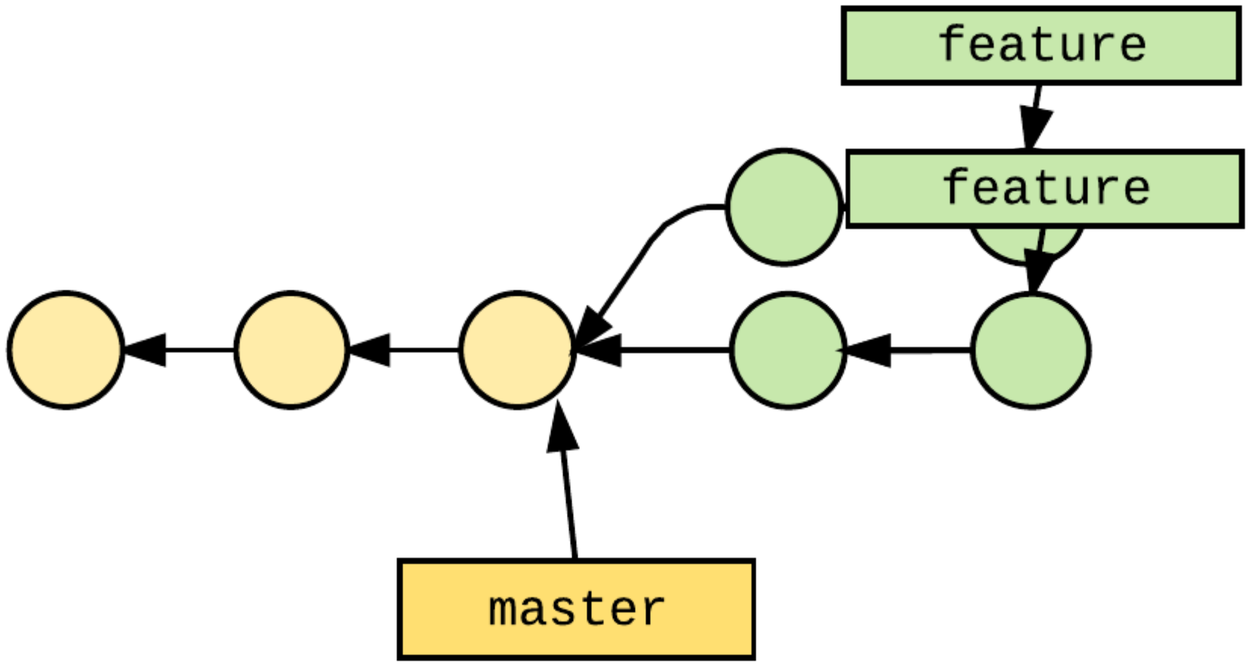
```
❯ git cat-file -p 649efc8c9
tree 8c90eb0c4af7e0b05f736c933dc55e14e45031f2
parent 14176d1d63b6a85308fd61a9a20c8aeed398043b
parent 1ea4e219580910921dce70e96e5da96f12f5066b
author Kenneth Reitz <me@kennethreitz.org> 1499378257 -0400
committer Kenneth Reitz <me@kennethreitz.org> 1499378257 -0400

Merge branch 'master' of github.com:kennethreitz/requests
```



FAST FORWARD

```
❯ git checkout master
❯ git merge feature
Updating 2733233..9703930
Fast-forward
 index.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

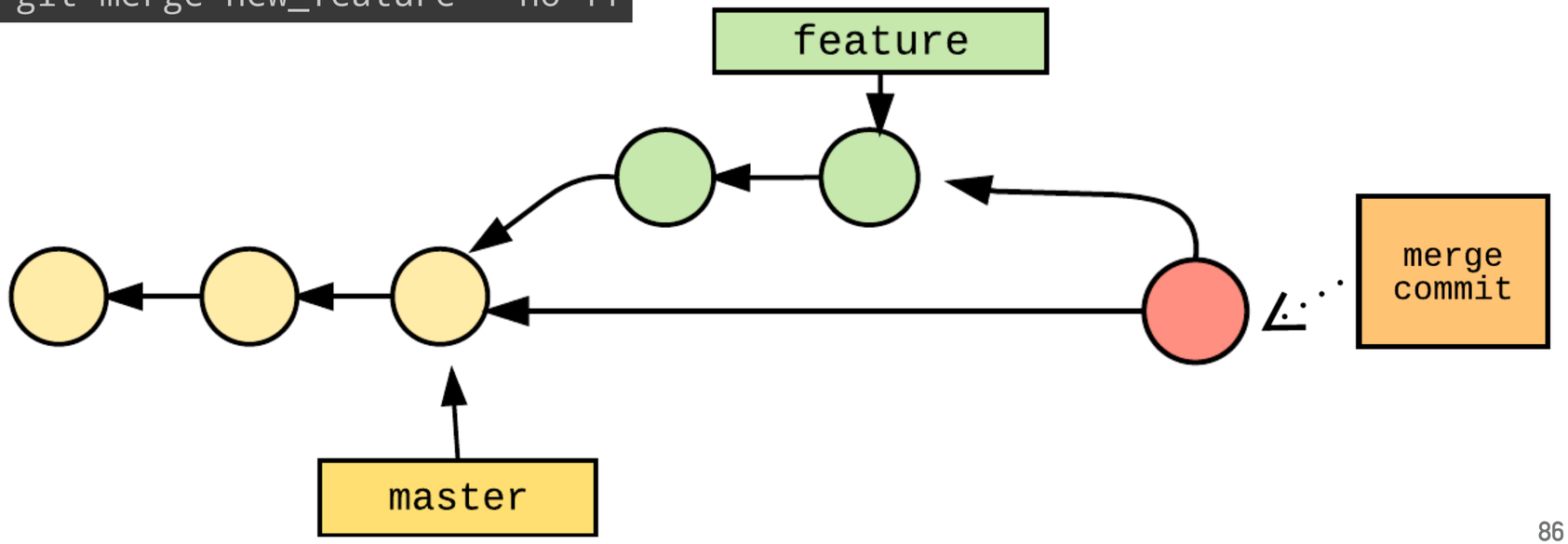
Fast-forward happens when there are no commits on the base branch that occurred after the feature branch was created.



GIT MERGE --NO-FF (NO FAST FORWARD)

- To retain the history of a merge commit, even if there are no changes to the base branch:
  - use `git merge --no-ff`
- This will *force* a merge commit, even when one isn't necessary.

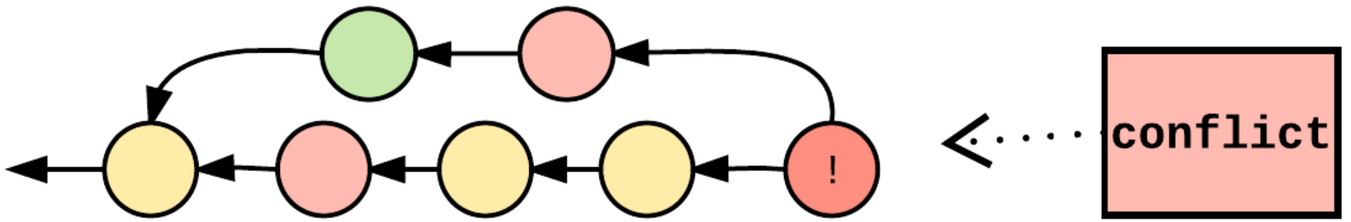
```
❯ git checkout master
❯ git merge new_feature --no-ff
```



MERGE CONFLICTS

- Attempt to merge, but files have diverged.
- Git stops until the conflicts are resolved.

```
› git merge feature
Auto-merging feature
CONFLICT (add/add): Merge conflict in feature
```



GIT RERERE – REUSE RECORDED RESOLUTION

- git saves how you resolved a conflict
- next conflict: reuse the same resolution

- useful for:
  - long lived feature branch (like a refactor)
  - rebasing

GIT RERERE – REUSE RECORDED RESOLUTION

Turn it on:

- **`git config rerere.enabled true`**
- use **`--global`** flag to enable for *all* projects

```
❯ git config rerere.enabled true

❯ git checkout master
❯ git merge feature
Auto-merging feature
CONFLICT (add/add): Merge conflict in file
Recorded preimage for 'file'
Automatic merge failed; fix conflicts and then commit the result.
### Fix the merge conflict in file.
❯ git add file
❯ git commit -m "Resolve conflict"
Recorded resolution for 'feature'.
[master 0fe266d] Resolve conflict
```

When I commit the conflict resolution, it's **recorded.**

GIT RERERE – REUSE RECORDED RESOLUTION

Now: Feature wasn't ready, I undid the merge.

When I try to merge again:

```
❯ git merge feature
Auto-merging feature
CONFLICT (add/add): Merge conflict in feature
Resolved 'feature' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.

❯ git add file

❯ git diff --staged
diff --git a/file b/file
index 587be6b..a354eda 100644
--- a/file
+++ b/file
@@ -1 +1 @@
-The old change
+This is how I resolved my conflict.
\ No newline at end of file
```

The resolution is **automatically reapplied.**

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise4-MergingAndReReRe.md

HISTORY & DIFFS

BAD COMMIT MESSAGES

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

GOOD COMMITS ARE IMPORTANT

- Good commits help preserve the history of a code base.
- They help with:
  - debugging & troubleshooting
  - creating release notes
  - code reviews
  - rolling back
  - associating the code with an issue or ticket

A GOOD COMMIT MESSAGE

Commit message is in future tense. ‘Fix’ vs ‘Fixed’

Short subject, followed by a blank line.

```
git-rebase: don't ignore unexpected command line arguments

Currently, git-rebase will silently ignore any unexpected command-line
switches and arguments (the command-line produced by git rev-parse).
This allowed the rev-parse bug, fixed in the preceding commits, to go
unnoticed.  Let's make sure that doesn't happen again.  We shouldn't be
ignoring unexpected arguments.  Let's not.
```

A description of the current behavior,
a short summary of **why** the fix is needed.
Mention side effects.

The description is broken into 72 char lines.

ANATOMY OF A GOOD COMMIT

- Good commit message
- Encapsulates one logical idea
- Doesn't introduce breaking changes
  - i.e. tests pass

GIT LOG

- git log - the basic command that shows you the history of your repository

```
› git log
commit 25b381008970939463641 2adf3d887cb55cb47eb (HEAD -> test)
Author: Nina Zakharenko <nina@nnja.io>
Date:   Wed Sep 27 22:46:38 2017 -0700

    Add a README file to the project

commit cd0b57c21487871b6353a58366414b72f2e4ee22 (tag: v1.0, tag: my-first-
commit)
Author: Nina Zakharenko <nina@nnja.io>
Date:   Sun Sep 24 14:44:28 2017 -0700

    Initial commit
```

GIT LOG --SINCE

- The site is slow. What changed since yesterday?
  - `git log --since="yesterday"`
  - `git log --since="2 weeks ago"`

GIT LOG --FOLLOW

- Log files that have been moved or renamed
  - `git log --name-status --follow -- <file>`

```
› git log --name-status --follow -- <file>
```

GIT LOG --GREP

- Search for commit messages that match a regular expression:
  - `git log –grep <regexp>`
  - Can be mixed & matched with other git flags.
- Example:

```
› git log --grep=mail --author=nina --since=2.weeks
```

GIT LOG DIFF-FILTER

- Selectively include or exclude files that have been:
- (A)dded, (D)eleted, (M)odified & more…

```
❯ git log --diff-filter=R --stat


5e1ced3 make session serializer extensible support serializing 1-item dicts
with tag as key refactor serializer into flask.json.tag module continues
#1452, closes #1438, closes #1908
 flask/{json.py => json/__init__.py} | 13 ++-----------
 1 file changed, 2 insertions(+), 11 deletions(-)
8cf32bc Adds in blueprints and an application factory
 examples/flaskr/flaskr/{ => blueprints}/flaskr.py | 55 +++++++++++++++
+-----------------------------------------
 1 file changed, 15 insertions(+), 40 deletions(-)
92fa444 Moves largerapp into patterns dir and add test
 examples/{ => patterns}/largerapp/setup.py                          | 0
 examples/{ => patterns}/largerapp/yourapplication/__init__.py       | 0
 examples/{ => patterns}/largerapp/yourapplication/static/style.css  | 0
 examples/{ => patterns}/largerapp/yourapplication/templates/index.html | 0
```

GIT LOG: REFERENCING COMMITS

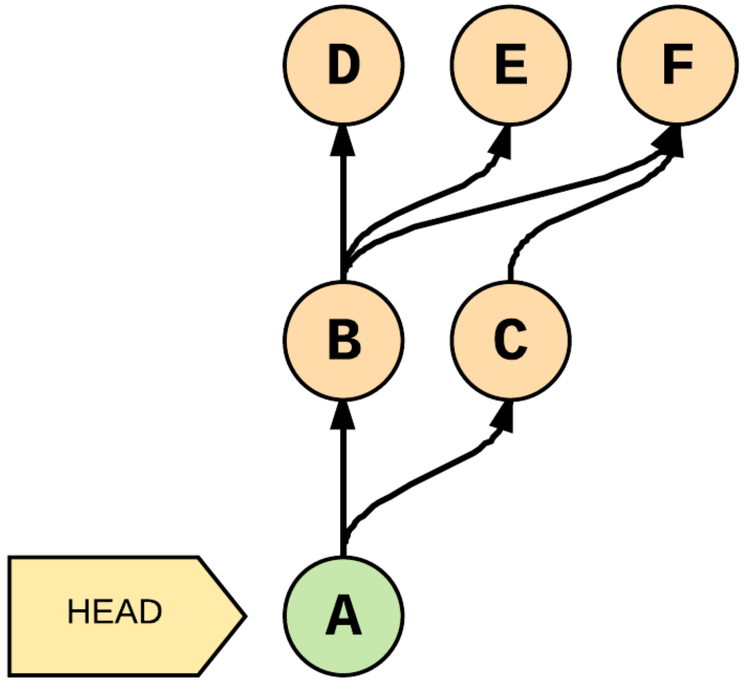
- ^ or ^**n**
  - no args: == ^1: the first parent commit
  - **n**: the nth parent commit

- **~ or ~n**
  - no args: == ~1: the first commit back, following 1st parent
  - **n:** number of commits back, ***following only 1st parent***

note: ^ and ~ can be combined

REFERENCING COMMITS

* Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.
* A is the latest commit.




```
A =       = A^0
B = A^    = A^1     = A~1
C = A^2
D = A^^   = A^1^1   = A~2
E = B^2   = A^^2
F = B^3   = A^^3   = A^2^1
```

an illustration, by Jon Loeliger, from the git-rev-parse docs

GIT SHOW: LOOK AT A COMMIT

- show commit and contents:
  - `git show <commit>`

- show files changed in commit:
  - `git show <commit> --stat`

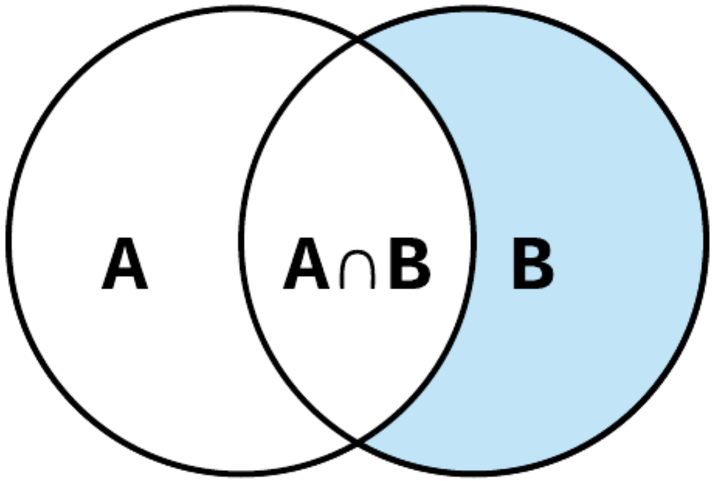
- look at a file from another commit:
  - `git show <commit>:<file>`

DIFF

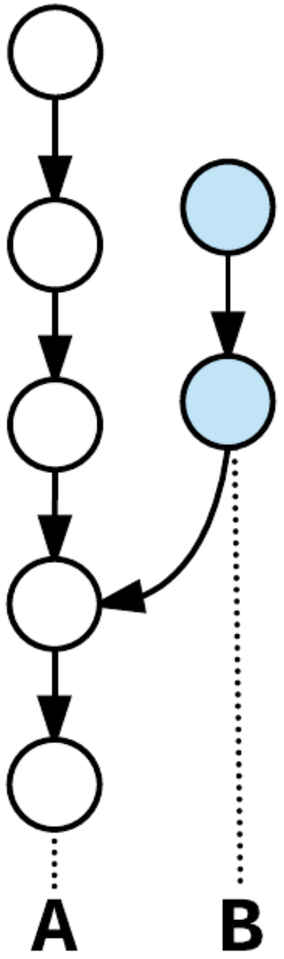
- Diff shows you changes:
  - between commits
  - between the staging area and the repository
  - what's in the working area

- unstaged changes
  - `git diff`
- staged changes
  - `git diff --staged`

DIFF COMMITS AND BRANCHES




```
git diff A B
git diff A..B
```



“DIFF” BRANCHES

- Which branches are merged with master, and can be cleaned up?
  - `git branch --merged master`

- Which branches aren’t merged with master yet?
  - `git branch --no-merged master`

EXERCISE

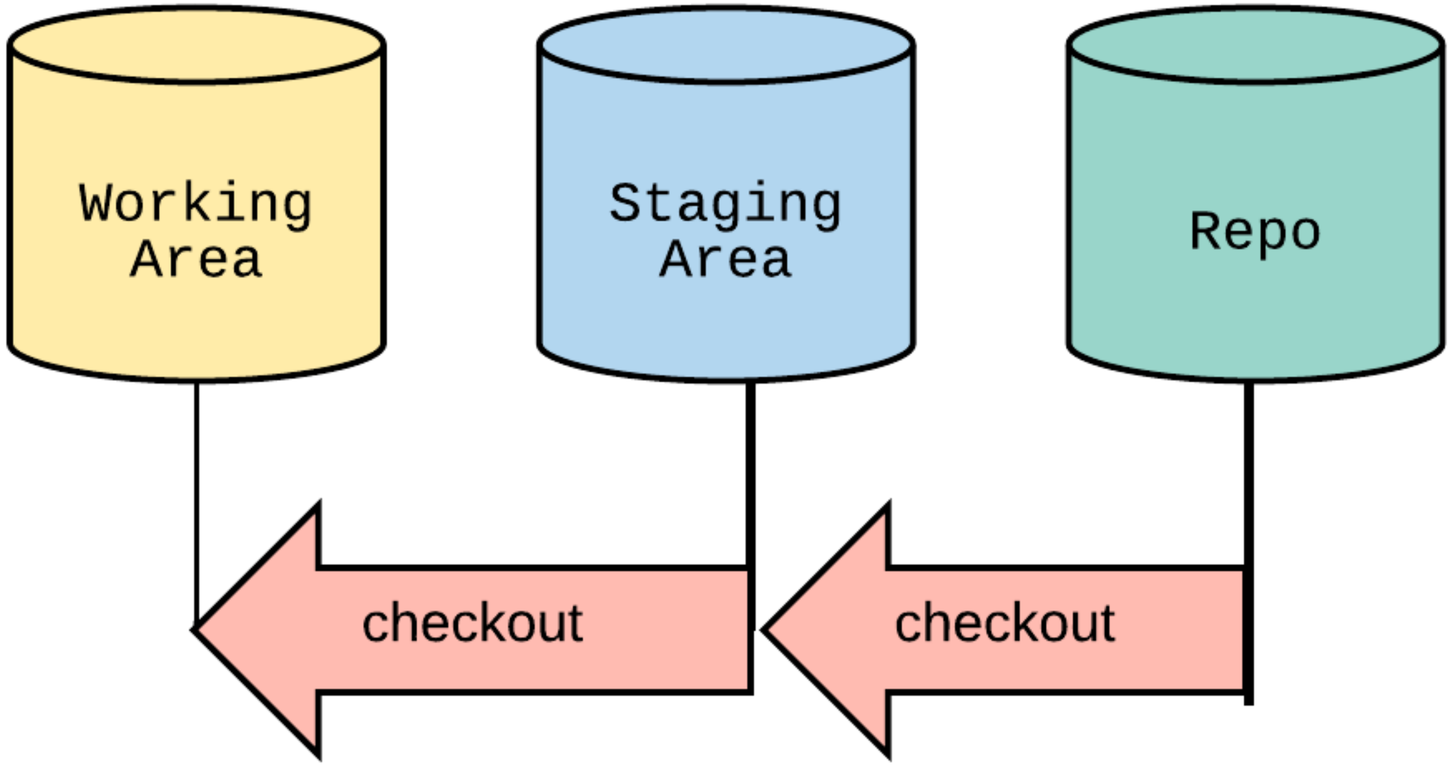
- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise5-HistoryAndDiffs.md

FIXING MISTAKES

checkout
reset
revert
clean

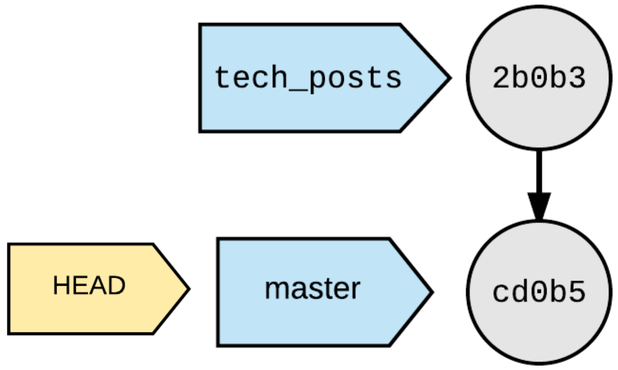
GIT CHECKOUT

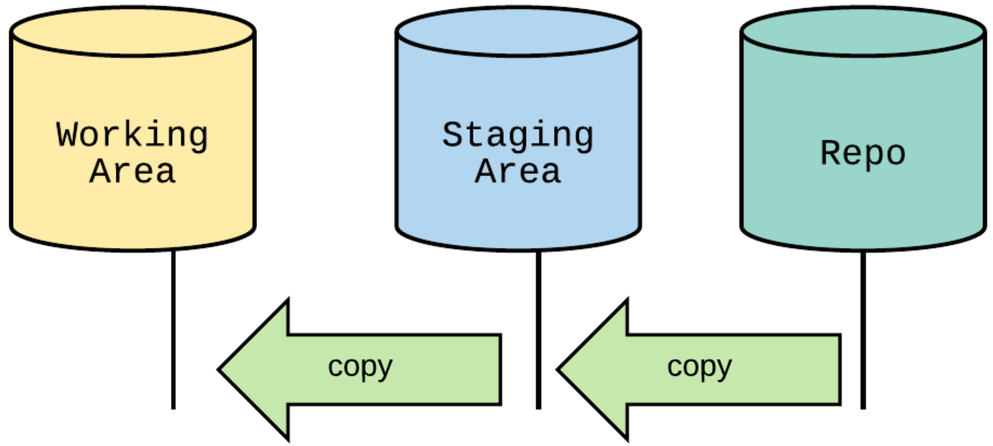
- Restore working tree files *or* switch branches



WHAT HAPPENS WHEN YOU GIT CHECKOUT A BRANCH?

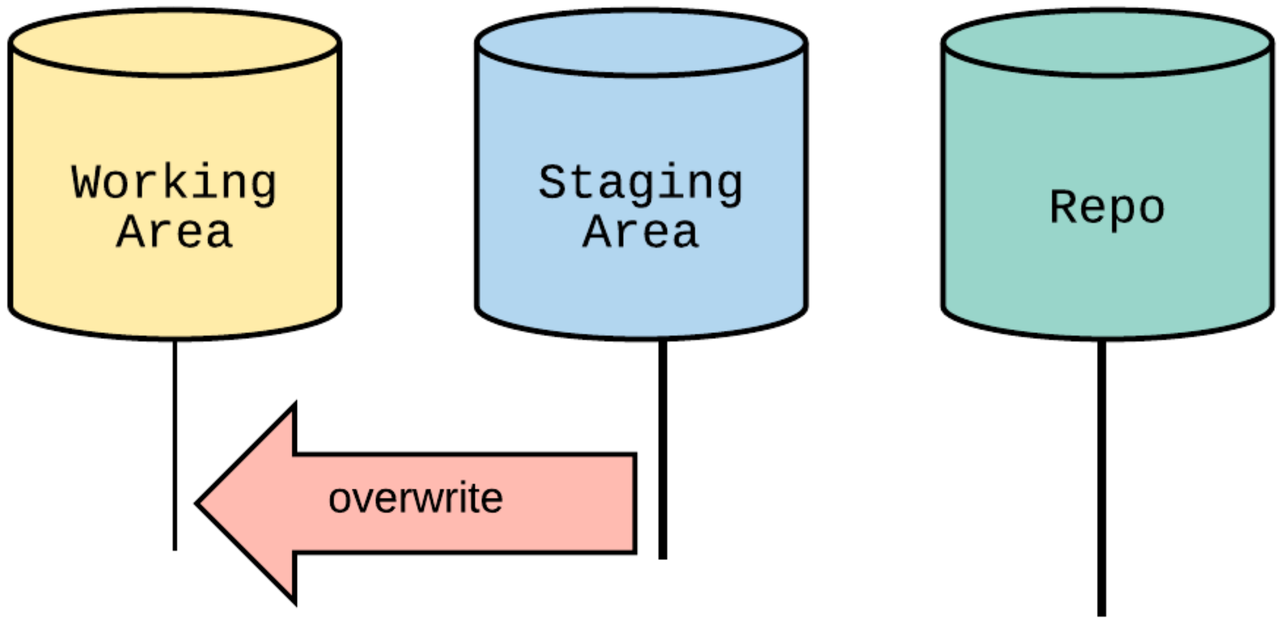
1. Change HEAD to point to the new branch
2. Copy the commit snapshot to the staging area
3. Update the working area with the branch contents





WHAT HAPPENS WHEN YOU GIT CHECKOUT -- FILE?

Replace the working area copy with the version from the current staging area



Warning:
This operation overwrites
files in the working directory without warning!

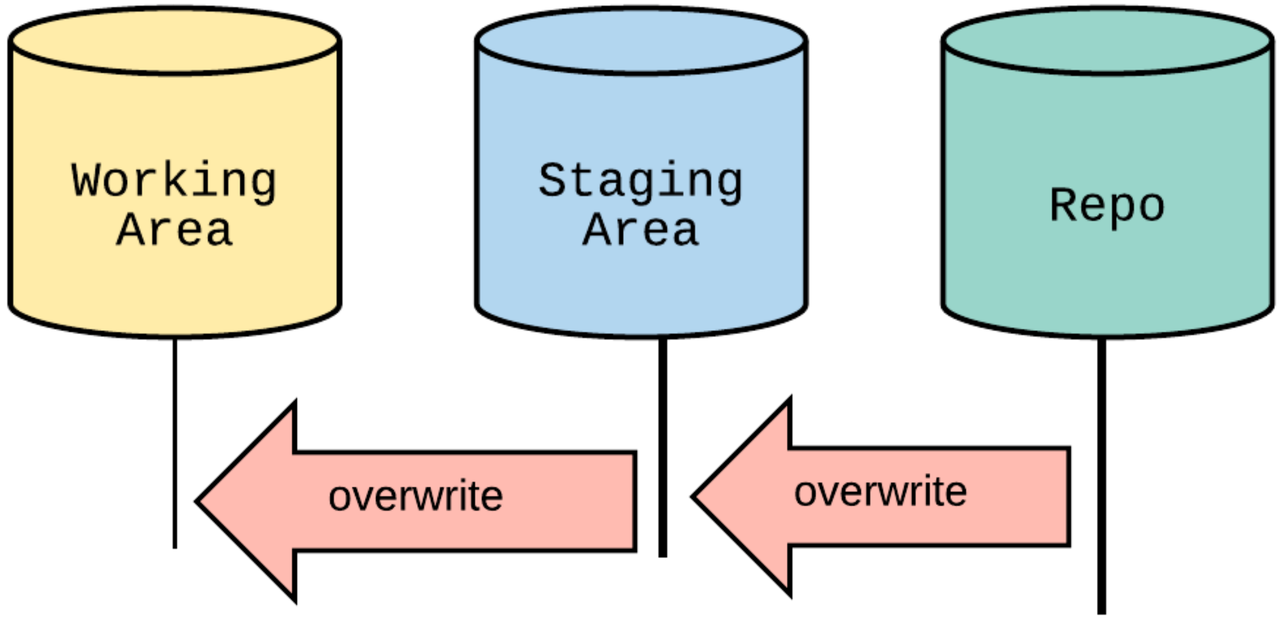
GIT CHECKOUT: OVERWRITE FILES WITH STAGING AREA COPY

- Overwrite the working area file with the staging area version from the last commit
  - `git checkout -- <file_path>`

Warning:
This operation overwrites
files in the working directory without warning!

WHAT HAPPENS WHEN YOU GIT CHECKOUT <COMMIT> -- FILE?

1. Update the staging area to match the commit
2. Update the working area to match the staging area.



Warning:
This operation overwrites
files in the staging area and working directory without warning!

GIT CHECKOUT: FROM A SPECIFIC COMMIT

- Checkout a file from a specific commit
  - *copies to both working area & staging area*
  - `git checkout <commit> -- <file_path>`

- Restore a deleted file
  - `git checkout <deleting_commit>^ -- <file_path>`

Warning:
This operation will overwrite
files in the working directory and staging area with no warning!

GIT CLEAN

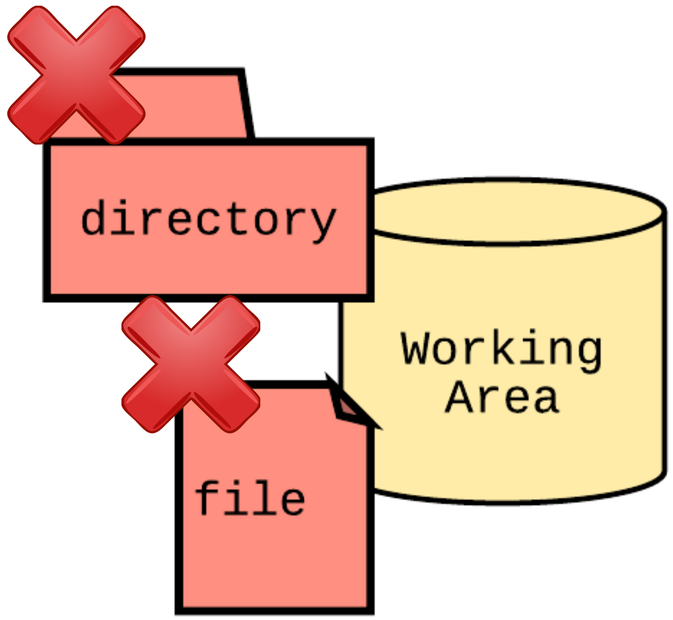
- Git clean will clear your working area by **deleting** untracked files.

```
Warning: this operation cannot be undone!
```

- Use the —dry-run flag to see what would be deleted
  - The -f flag to do the deletion
- The -d flag will clean directories

GIT CLEAN

```
❯ git clean --dry-run
Would remove a-note.txt

❯ git clean -d --dry-run
Would remove a-note.txt
Would remove scratch/


❯ git clean -d -f
Removing a-note.txt
Removing scratch/

❯ git status
nothing to commit, working tree clean
```

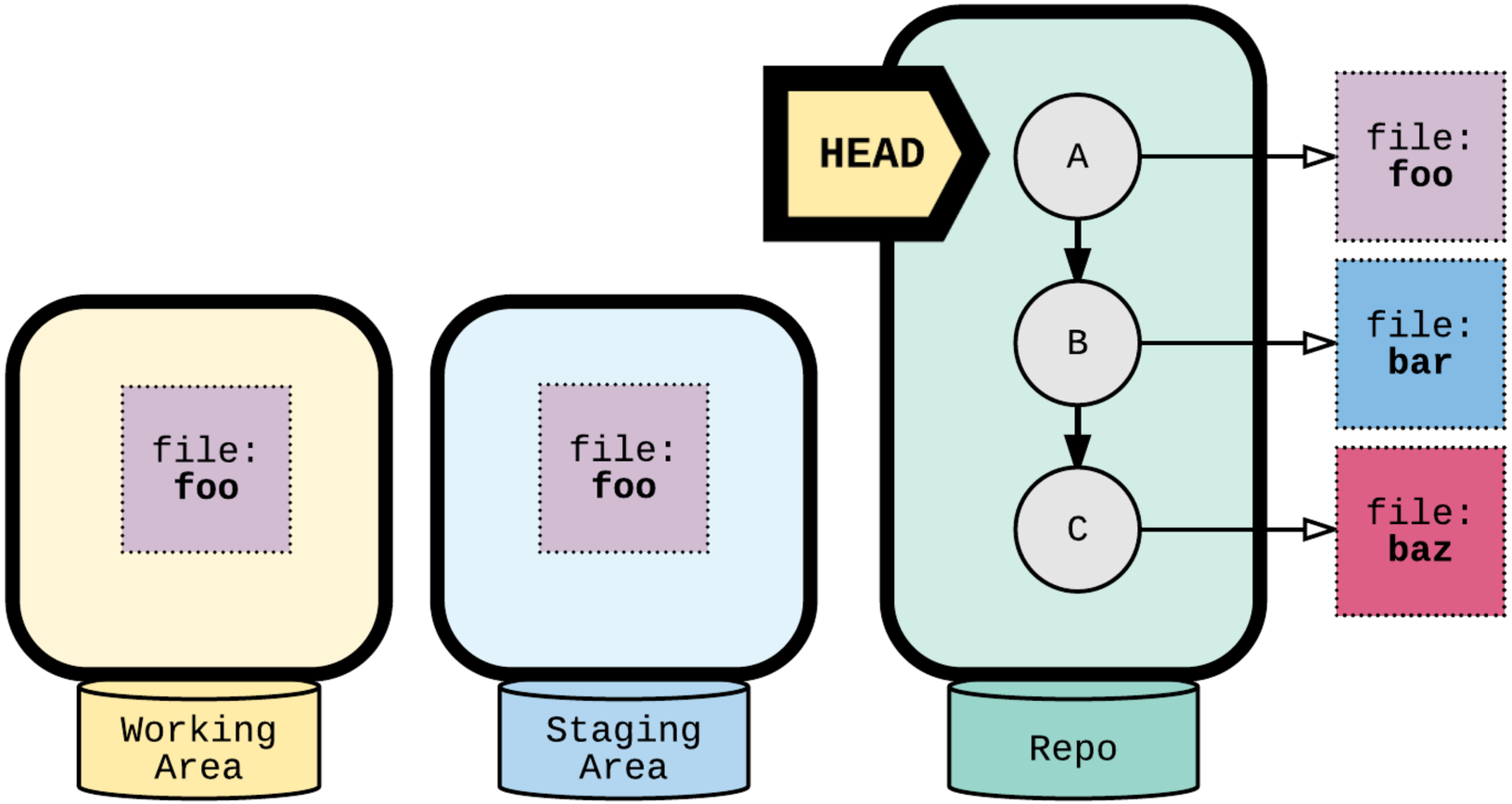


GIT RESET

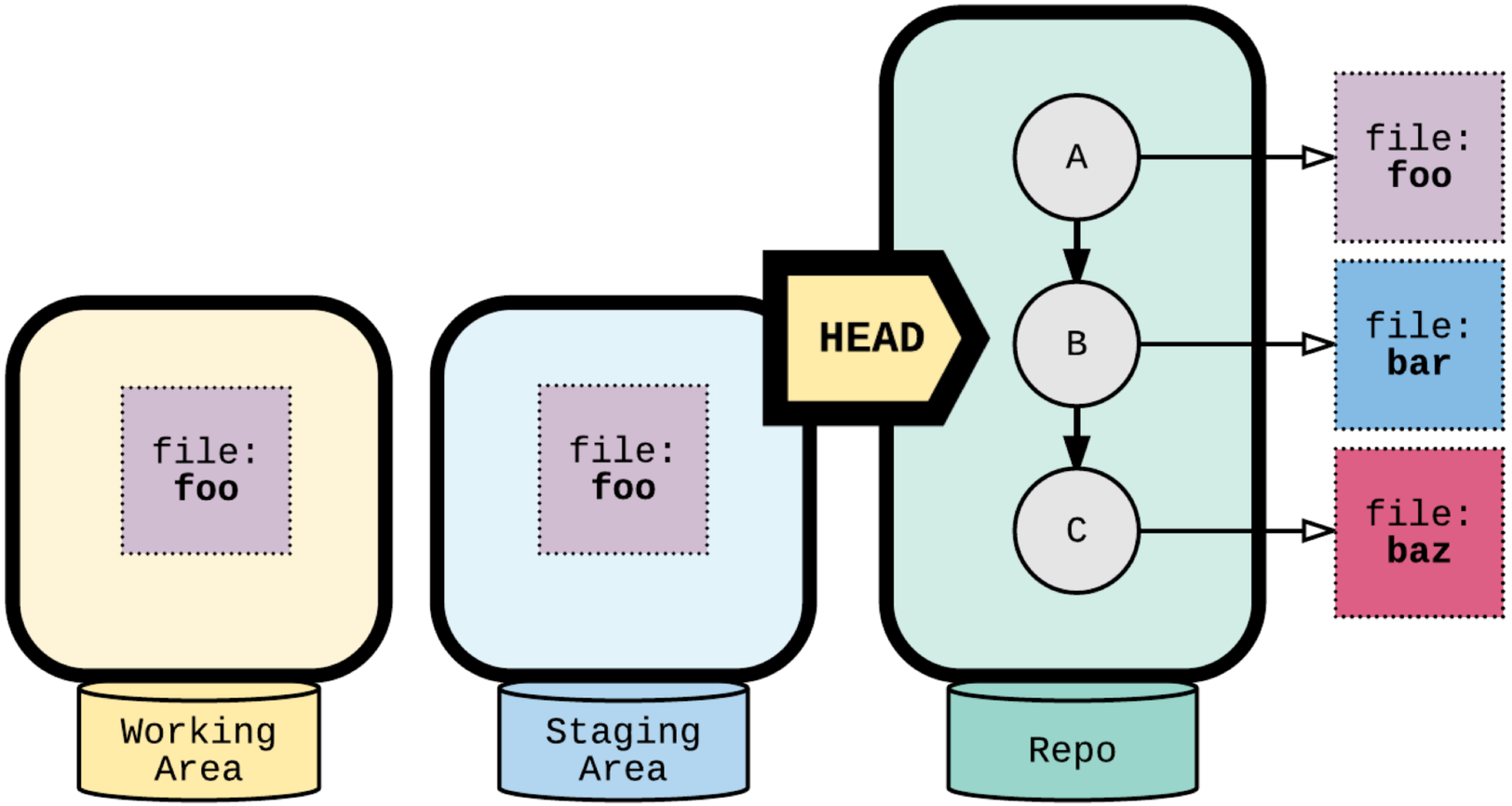
- Reset is another command that performs different actions depending on the arguments.
  - with a path
  - without a path
    - By default, git performs a `git reset –mixed`
- For commits:
  - Moves the `HEAD` pointer, optionally modifies files
- For file paths:
  - Does not move the `HEAD` pointer, modifies files

RESET --SOFT: MOVE HEAD

```
❯ git reset --soft HEAD~
```

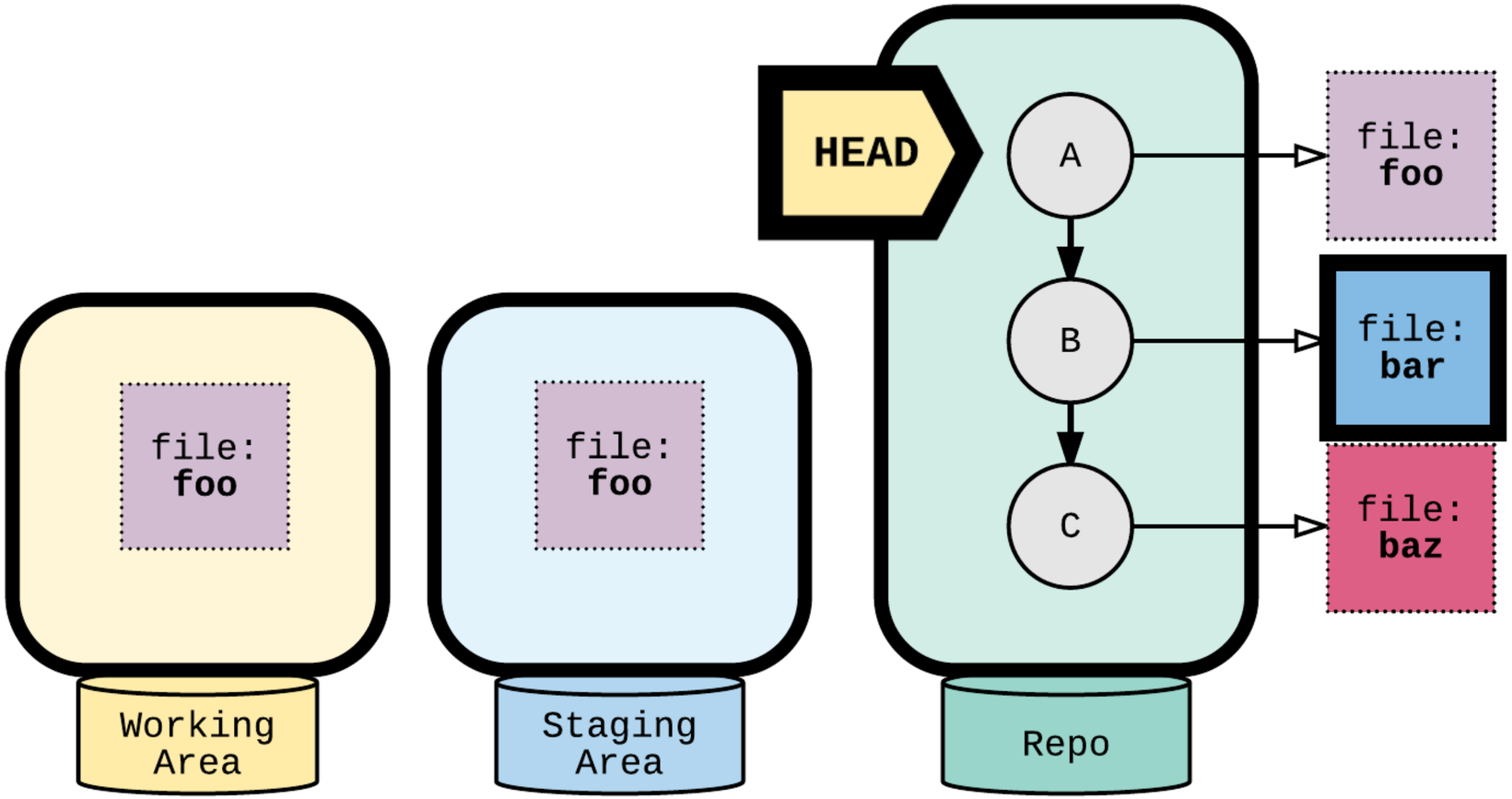


RESET --SOFT: RESULT

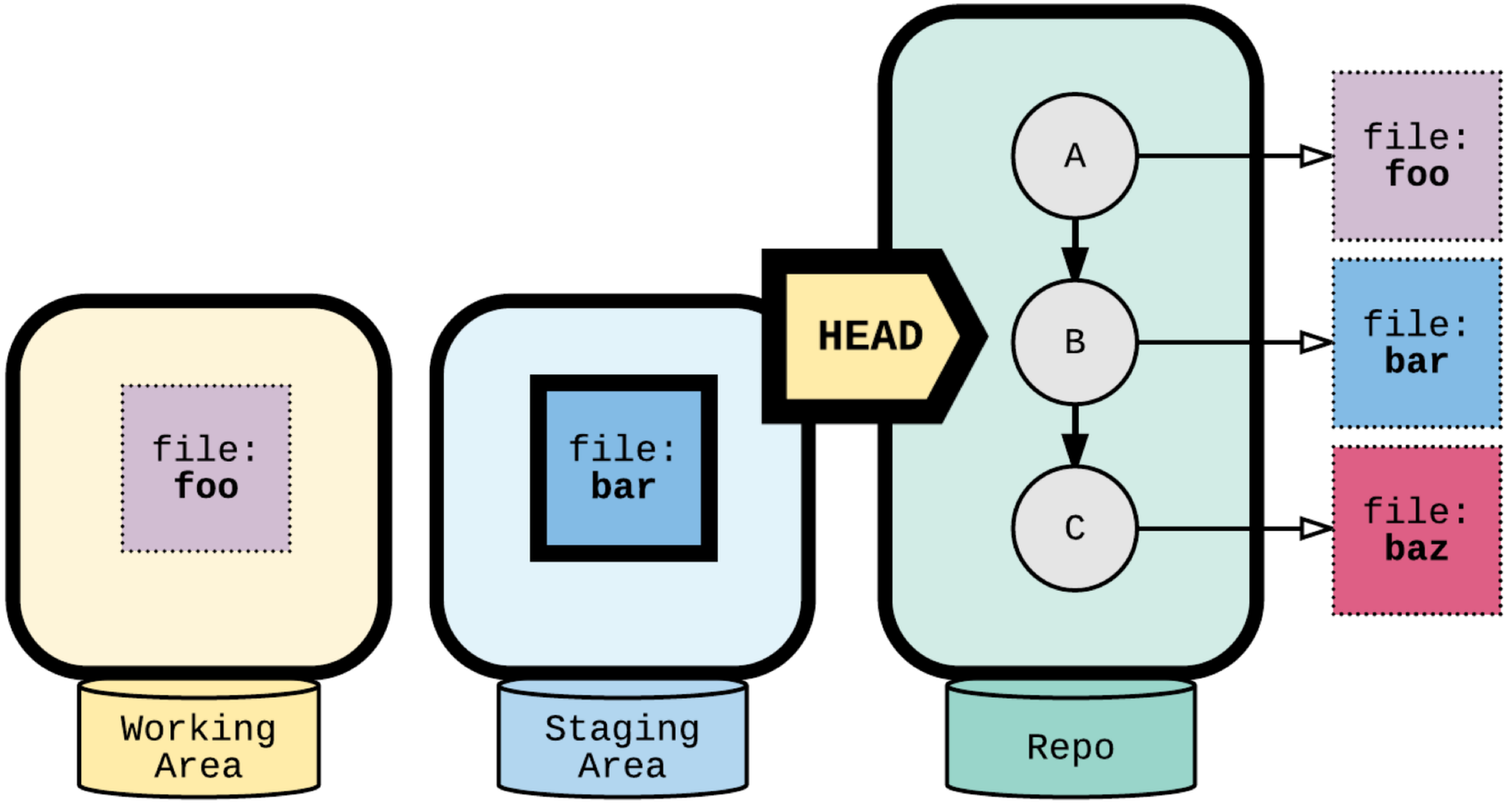


RESET --MIXED: MOVE HEAD, COPY FILES TO STAGE

```
❯ git reset HEAD~
❯ git reset --mixed HEAD~
```



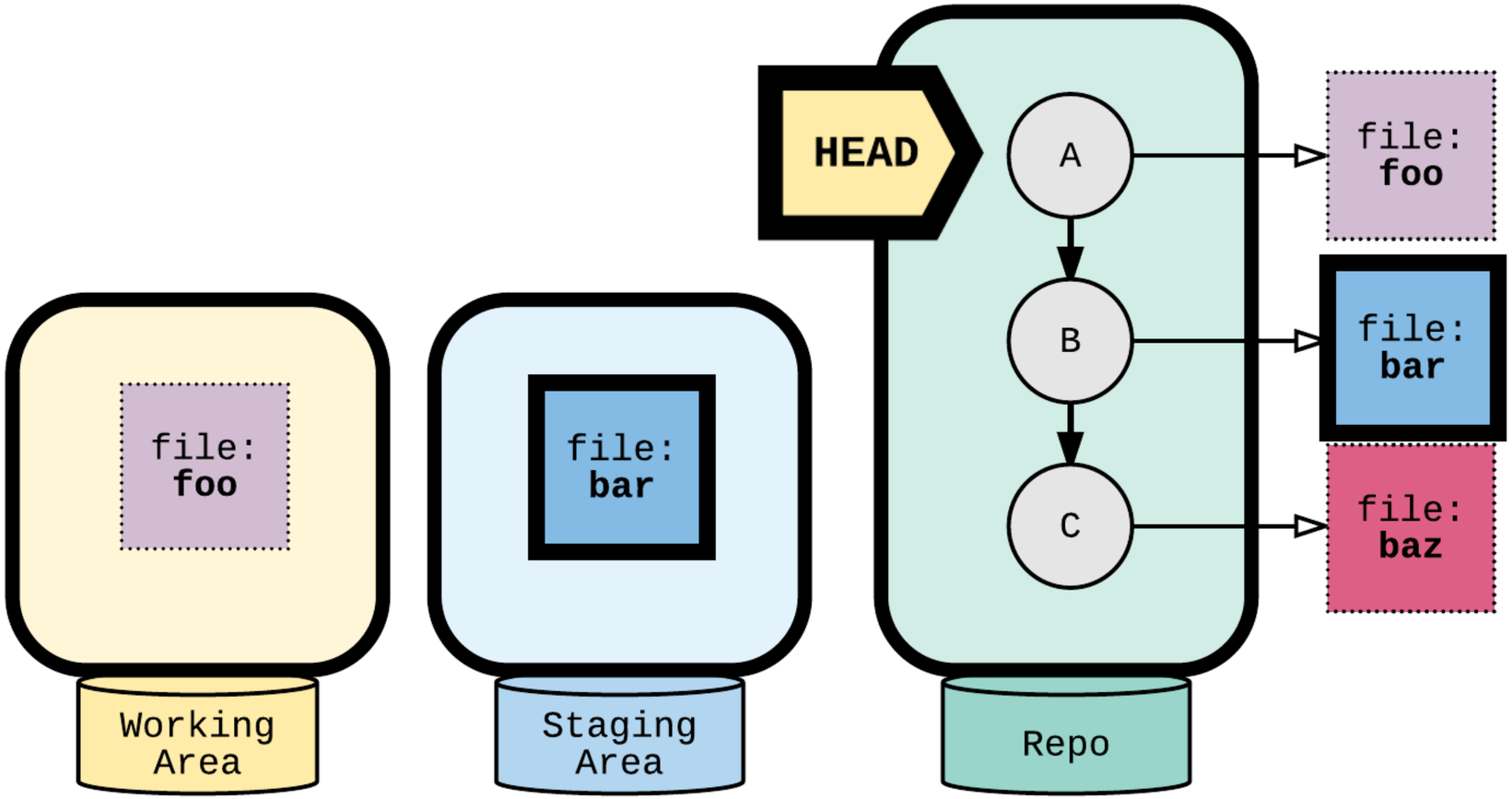
RESET --MIXED: RESULT



RESET --HARD: MOVE HEAD, COPY FILES TO STAGE & WORKING

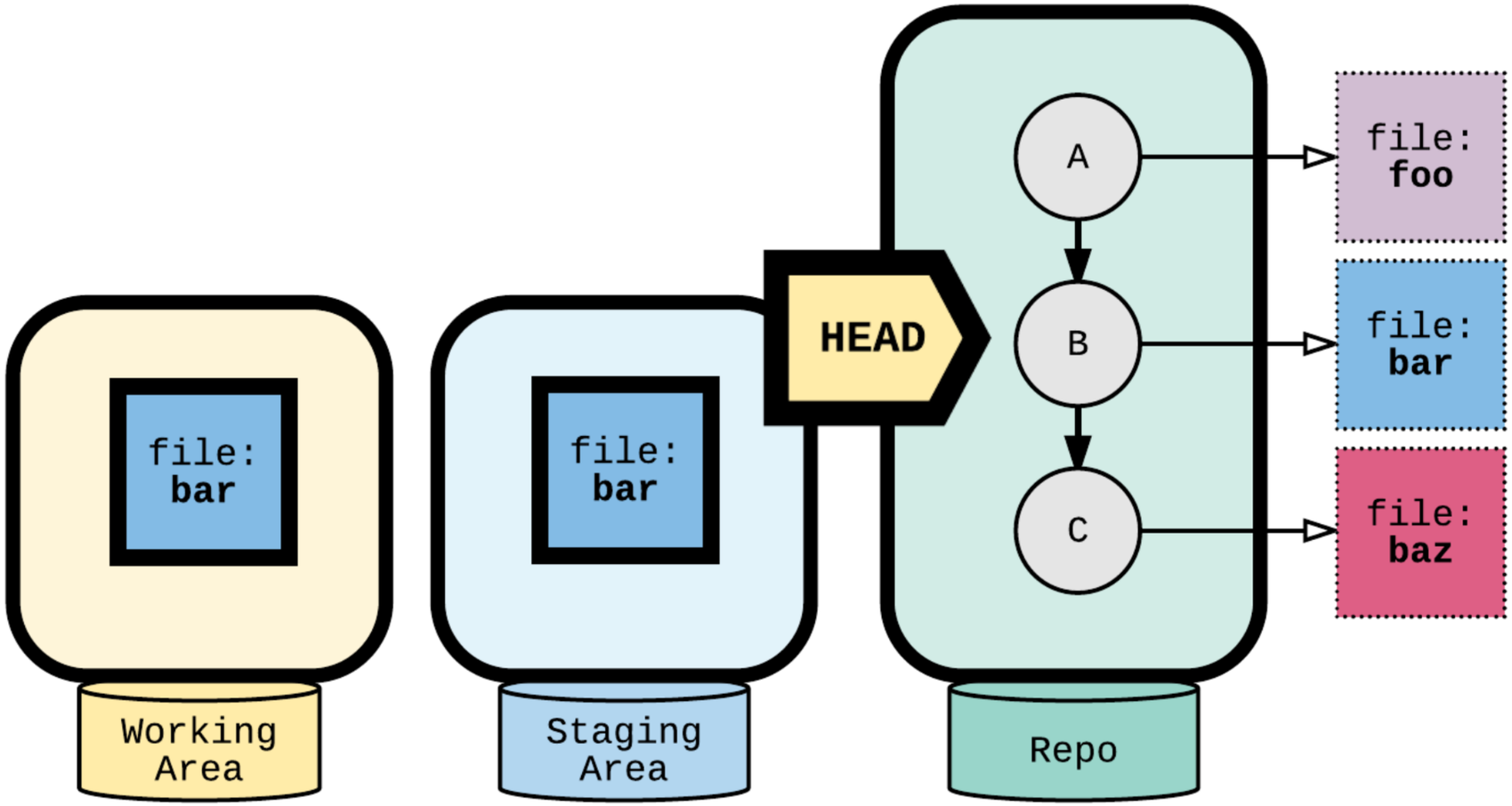
```
❯ git reset --hard HEAD~
```

Warning: this overwrites files and cannot be undone!



RESET --HARD: RESULT

Warning: this overwrites files and cannot be undone!



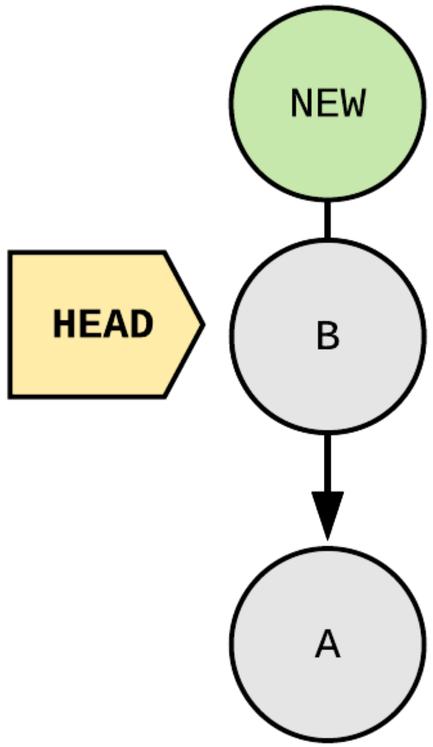
GIT RESET <COMMIT> CHEAT CHEAT

1. Move HEAD and current branch
2. Reset the staging area
3. Reset the working area

```
--soft = (1)
--mixed =(1) & (2) (default)
--hard = (1) & (2) & (3)
```

DANGER: GIT RESET CAN CHANGE HISTORY!

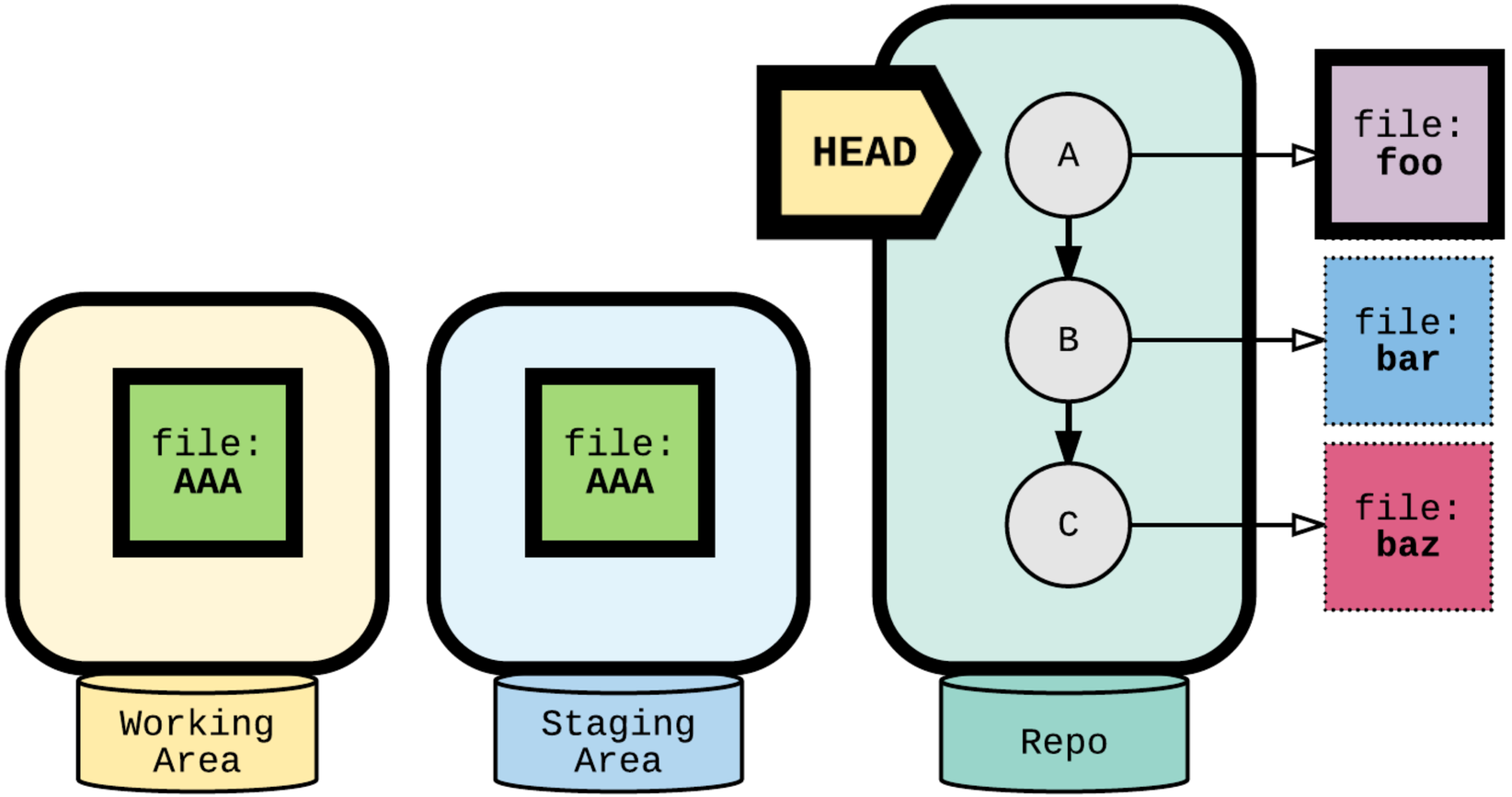
```
❯ git reset --soft HEAD~
❯ git commit -m “new commit”
```



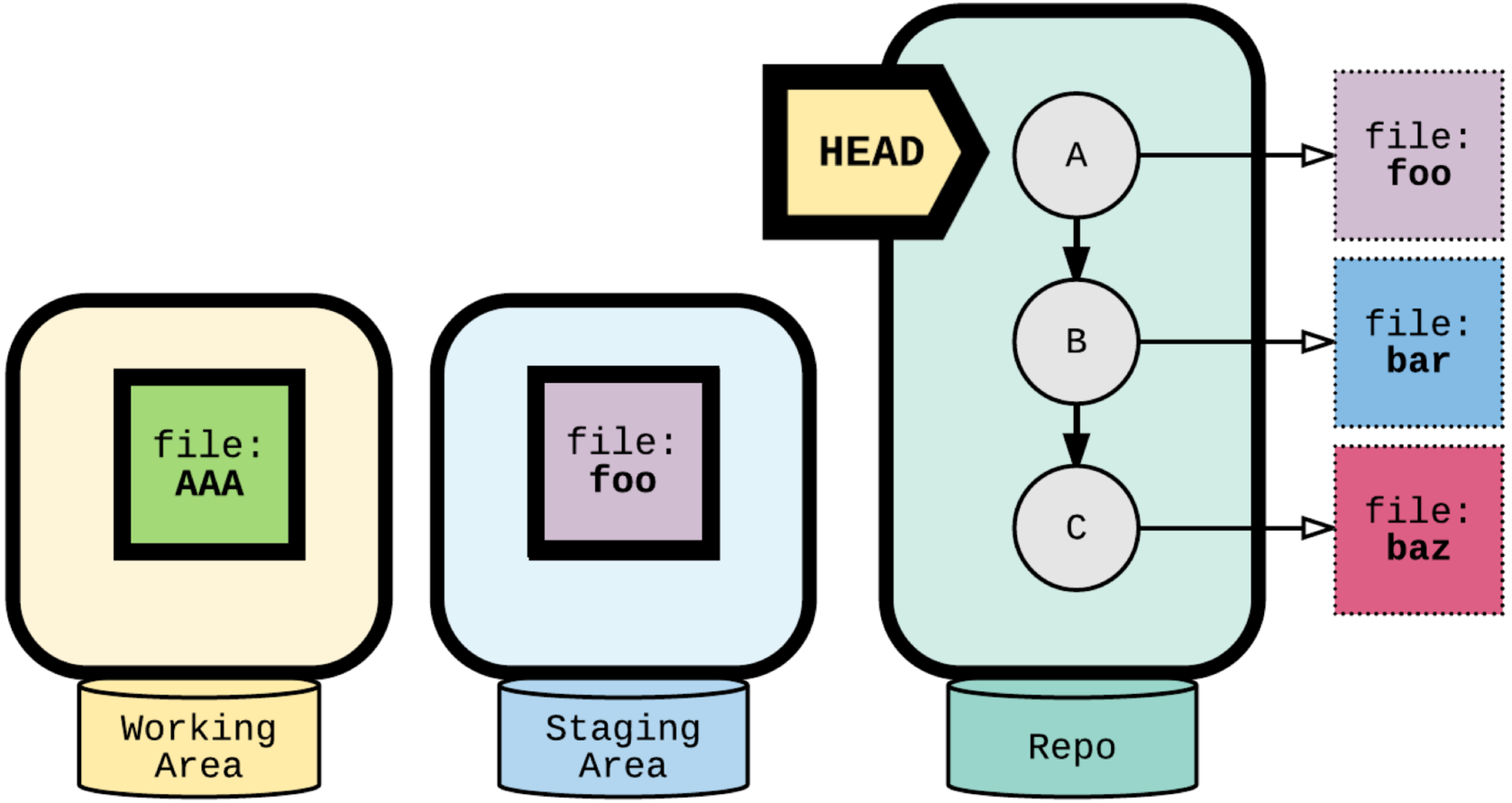
Warning: never push changed history to a shared or public repository!

GIT RESET -- <FILE>

```
› git reset <file>
```

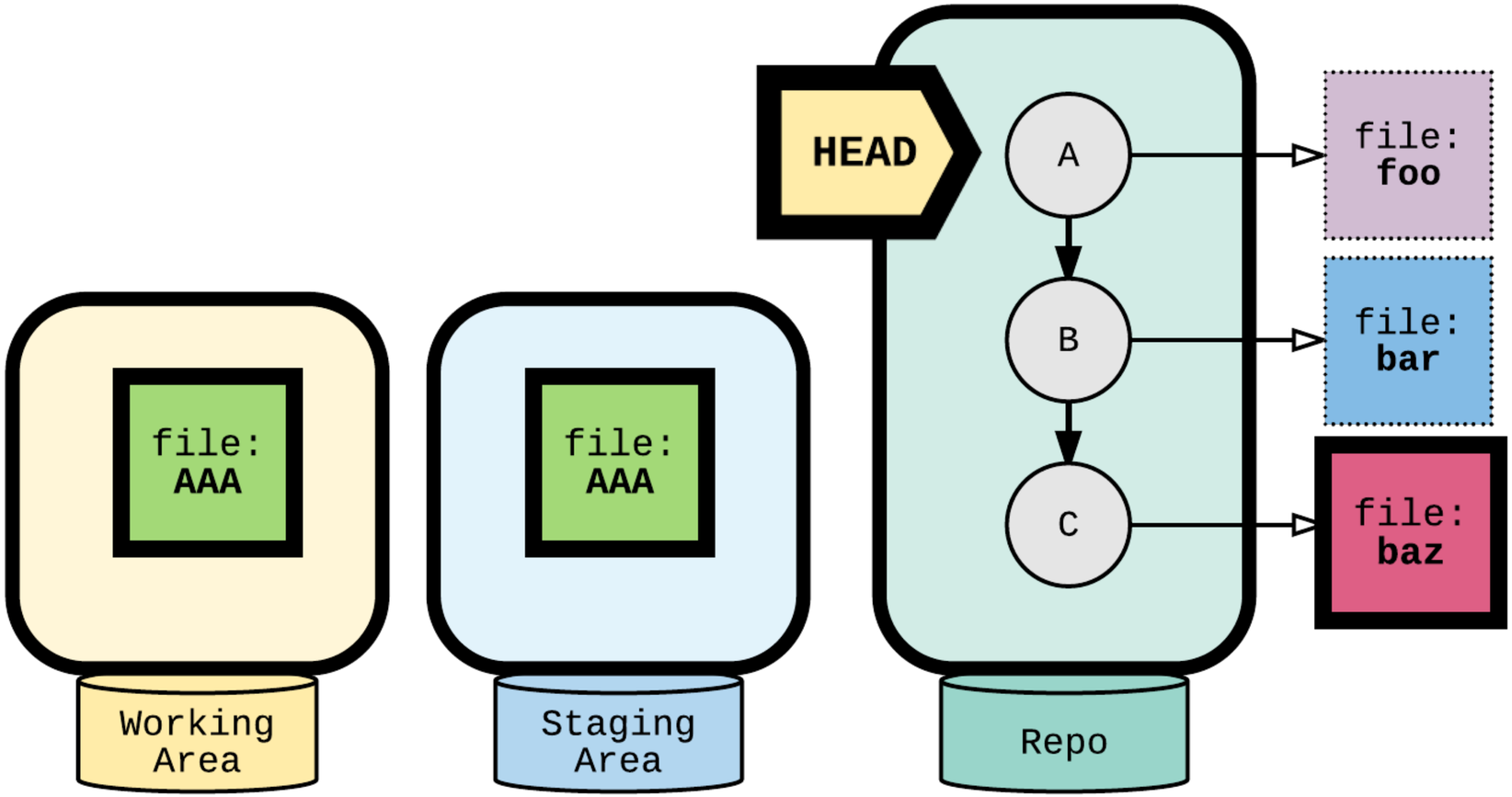


GIT RESET -- <FILE>: RESULT

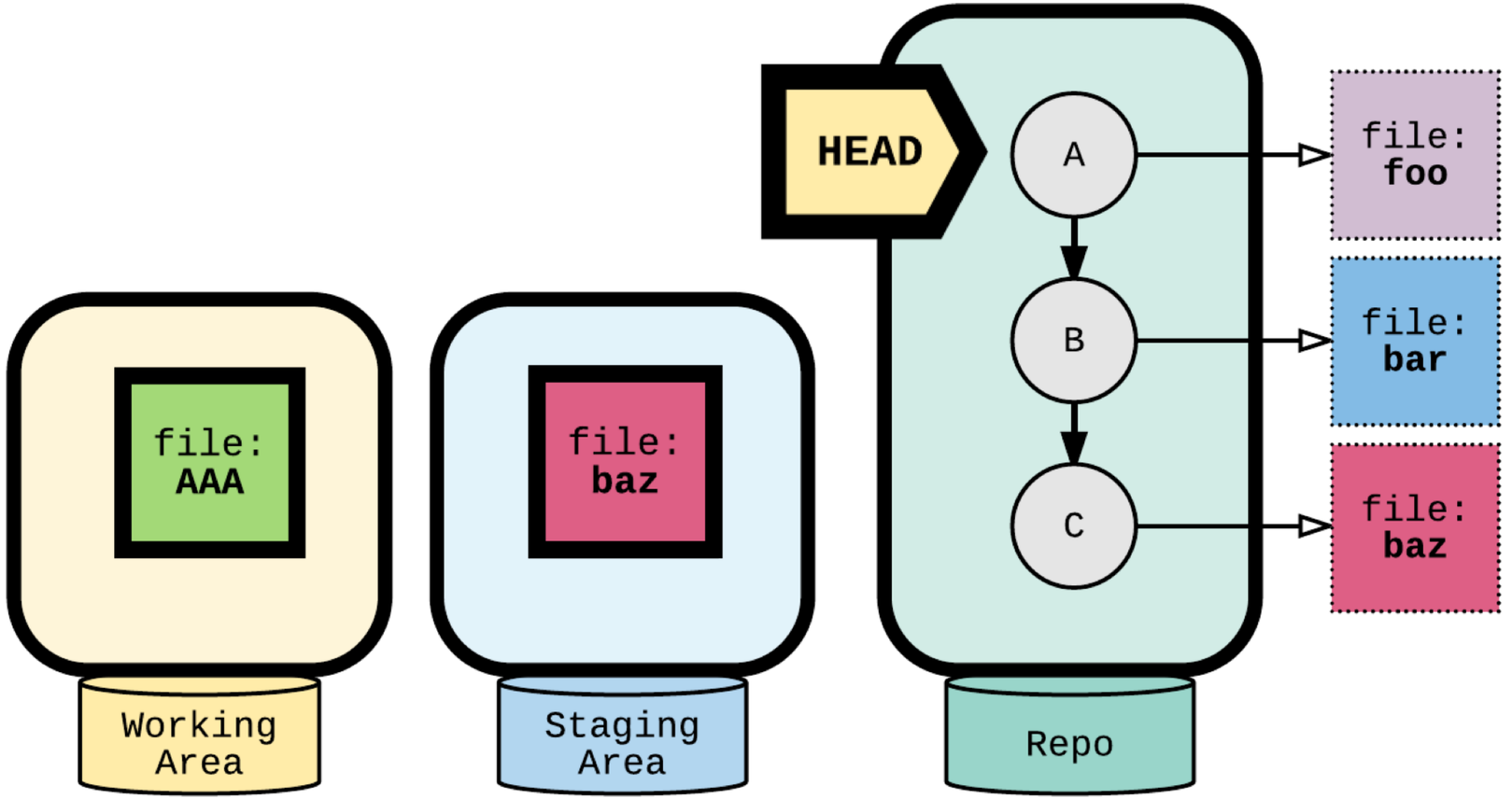


GIT RESET <COMMIT> -- <FILE>

```
❯ git reset <commit> -- <file>
```



GIT RESET <COMMIT> -- <FILE>: RESULT



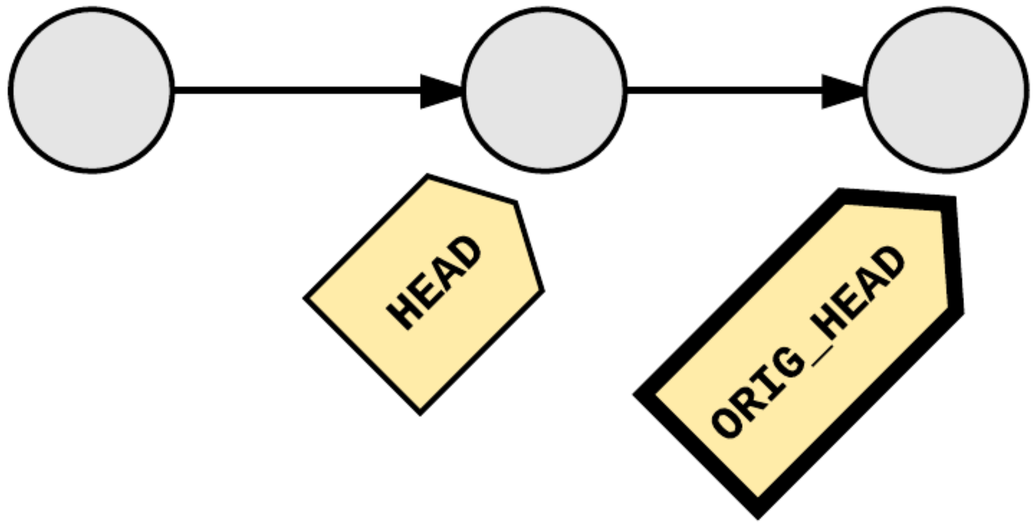
GIT RESET <COMMIT> -- <FILE> CHEAT CHEAT

1. ~~Move HEAD and current branch~~
2. **Reset the staging area**
3. ~~Reset the working area~~

This operation does not work with flags!

UNDO A GIT RESET WITH `ORIG_HEAD`

- In case of an accidental `git reset -`
- Git keeps the previous value of `HEAD` in variable called `ORIG_HEAD`
- To go back to the way things were:
  - **`git reset ORIG_HEAD`**



GIT REVERT – THE “SAFE” RESET

- Git revert creates a new commit that introduces the opposite changes from the specified commit.
- The original commit stays in the repository.

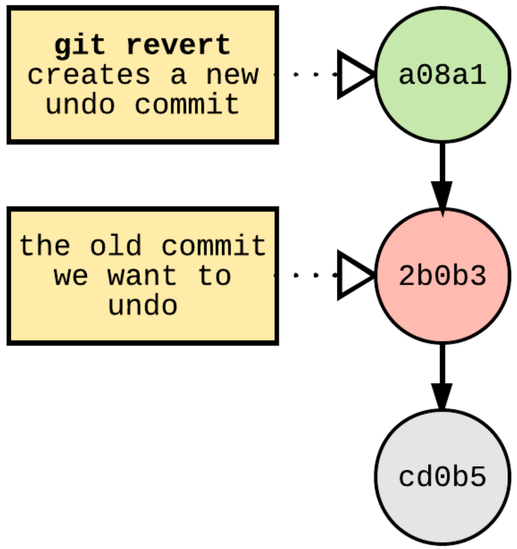
- Tip:
  - Use revert if you’re undoing a commit that has already been shared.
  - Revert ***does not*** change history.

GIT REVERT - PICK A COMMIT TO UNDO

```
❯ git log --oneline
2b0b3f2 (HEAD -> tech_posts) New blog post about python
cd0b57c (tag: v1.0, tag: my-first-commit, master) Initial commit

❯ git show 2b0b3f2
commit 2b0b3f24f3d7e8df809e46eb10c11ba66139acb8 (HEAD -> tech_posts)
Author: Nina Zakharenko <nina@nnja.io>
Date:   Sun Sep 24 14:55:35 2017 -0700

    New blog post about python

diff --git a/posts/python.txt b/posts/python.txt
```

GIT REVERT – THE “SAFE” RESET

```
❯ git revert 2b0b3f2
[tech_posts a08a108] Revert "New blog post about python"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 posts/python.txt

❯ git log --oneline
a08a108 (HEAD -> tech_posts) Revert "New blog post about python"
2b0b3f2 New blog post about python
cd0b57c (tag: v1.0, tag: my-first-commit, master) Initial commit
```



SUMMARY: COMMON WAYS OF UNDOING CHANGES

- `checkout`
- `reset`
- `revert`

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise6-FixingMistakes.md

REBASE, AMEND

Rewrite history!

AMEND A COMMIT

- Amend is a quick and easy shortcut that lets you make changes to the previous commit.

```
❯ cat index.txt
welcome.txt Welcome to my blog
python.txt Why python is my favorite language

❯ git commit -m "Add a blog post about Python"
[tech_posts 4080a79] Add a blog post about Python
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 A

❯ git add posts/python.txt

❯ git commit --amend
[tech_posts de53317] Add a blog post about Python
 Date: Wed Sep 27 22:12:31 2017 -0700
 2 files changed, 1 insertion(+)
 create mode 100644 A
 create mode 100644 posts/python.txt
```

oops! I forgot to add the blog post.

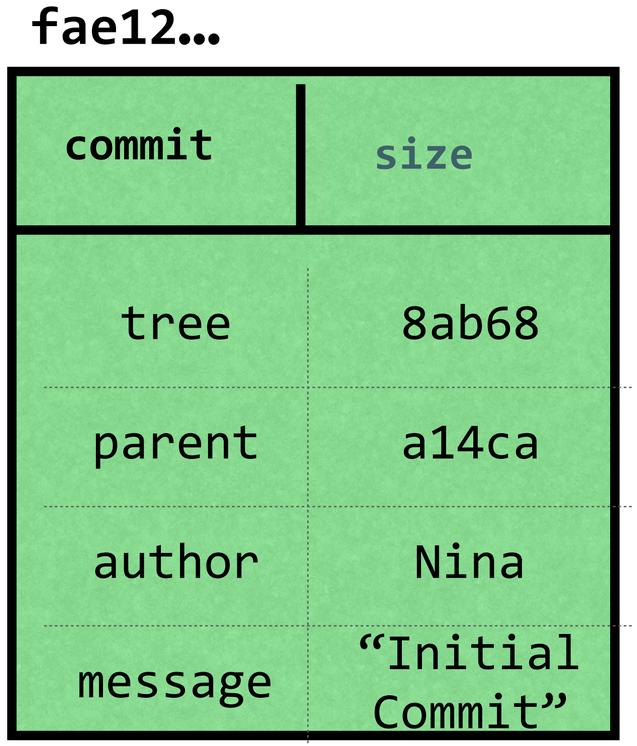
add new changes to the staging area

then, amend the last commit

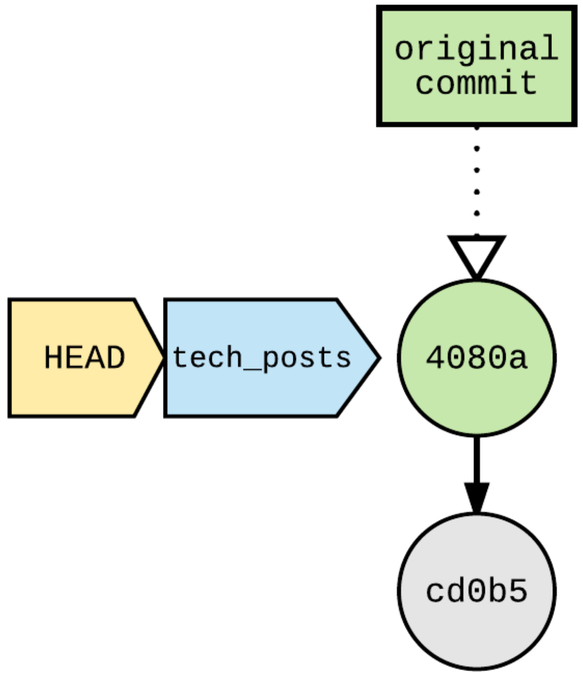
wait, why are the SHAs different?

COMMITS CAN'T BE EDITED!

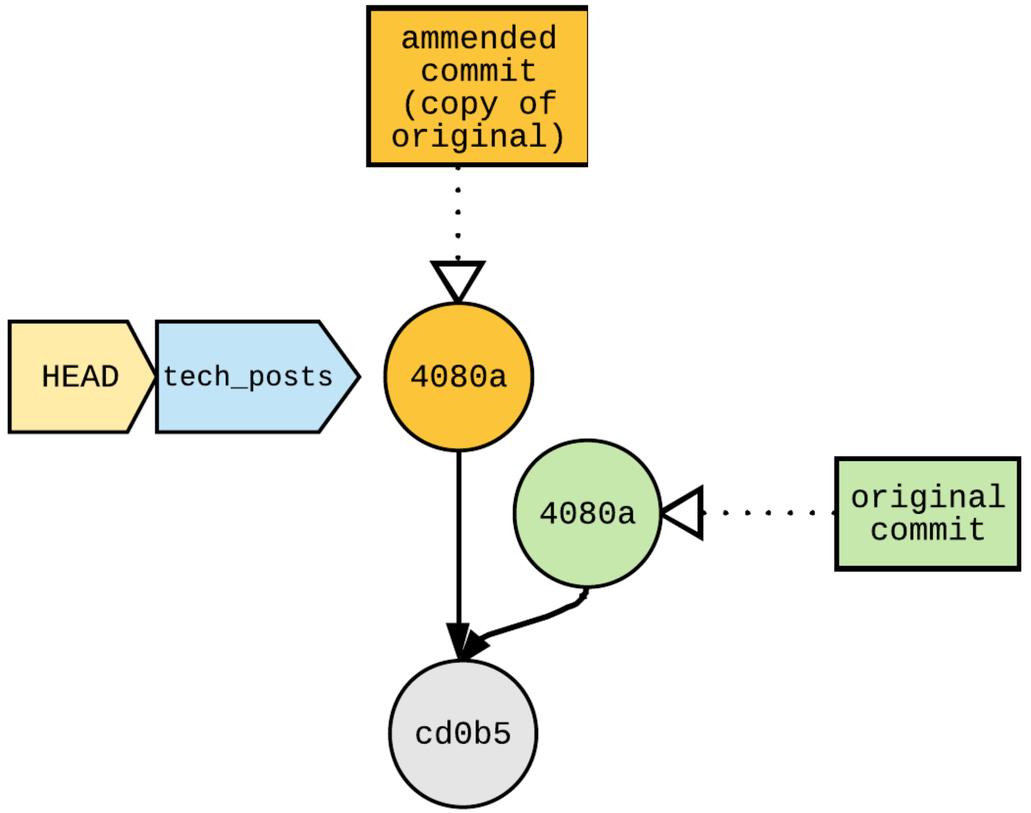
- Remember, **commits can't be edited!**
- A commit is referenced by the SHA of all its data.
- Even if the tree the commit points to is the same, and the author is the same, the date is still different!
- A new commit is created.



BEFORE



AFTER



original commit has no references pointing to it, and will eventually be garbage collected.

WHAT IS REBASE ANYWAY?

- Imagine our tech_posts and master branch have diverged.
- We don't want a messy merge commit in our history.

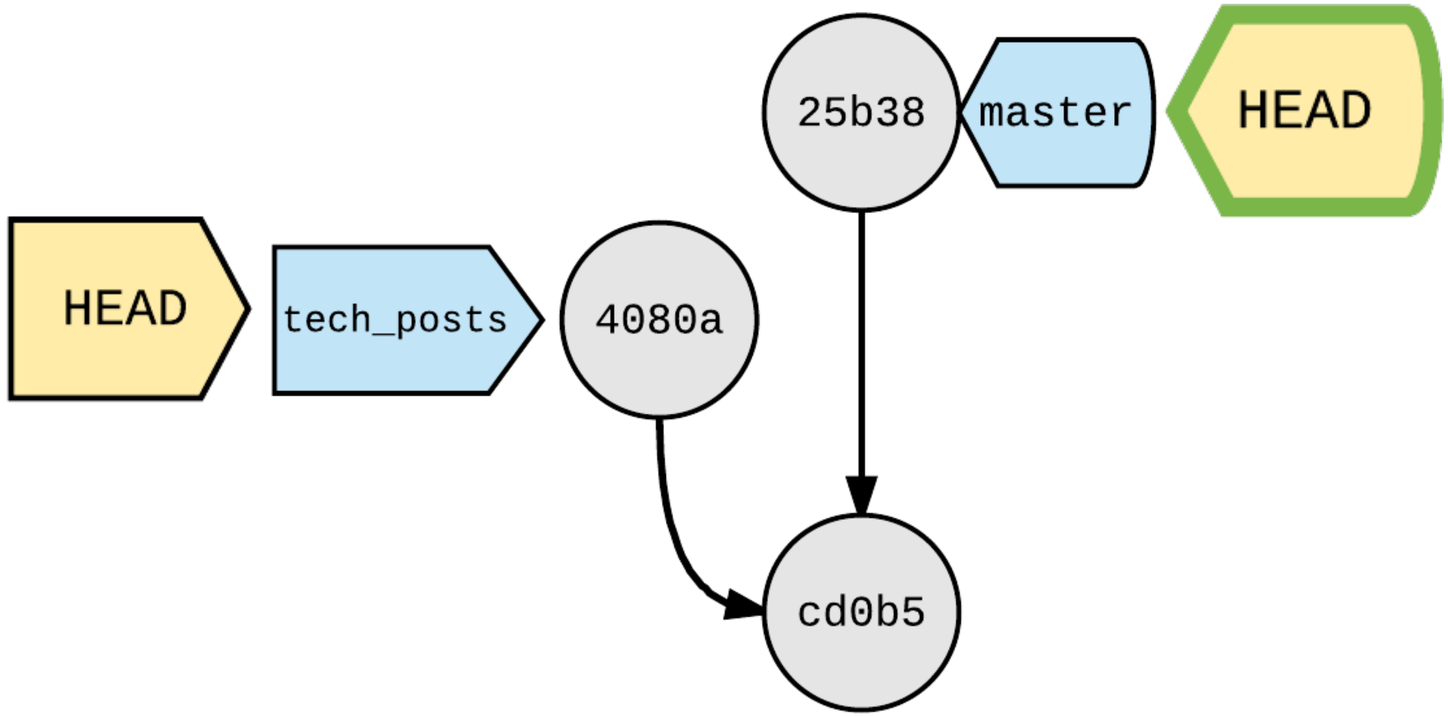
- We can pull in all the latest changes from master, and apply our commits on top of them by changing the parent commit of our commits.

Rebase = give a commit a new parent

(i.e. a new "base" commit)

REBASE: REWINDING HEAD

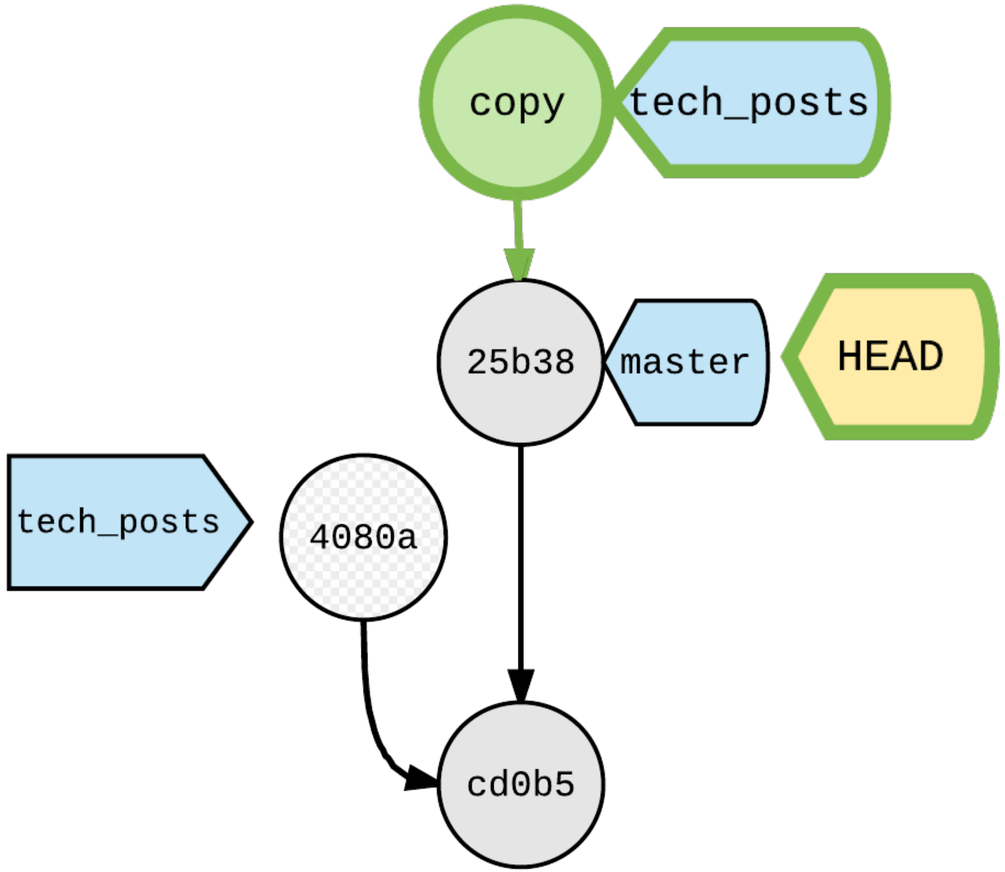
```
❯ git checkout tech_posts
Switched to branch ‘tech_posts'

❯ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add a blog post about Python
```



REBASE: APPLY NEW COMMITS

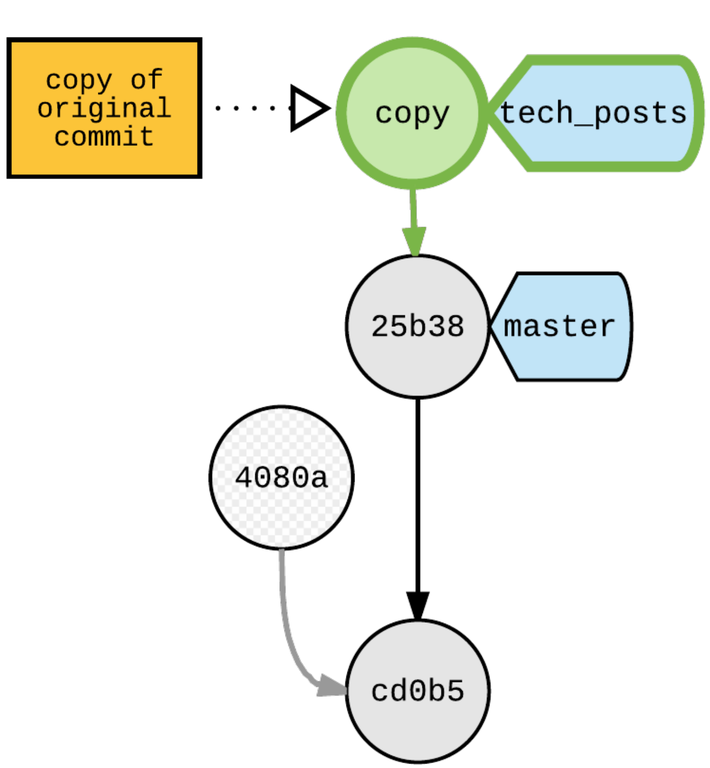
```
❯ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add a blog post about Python
```

(commit 4080a)

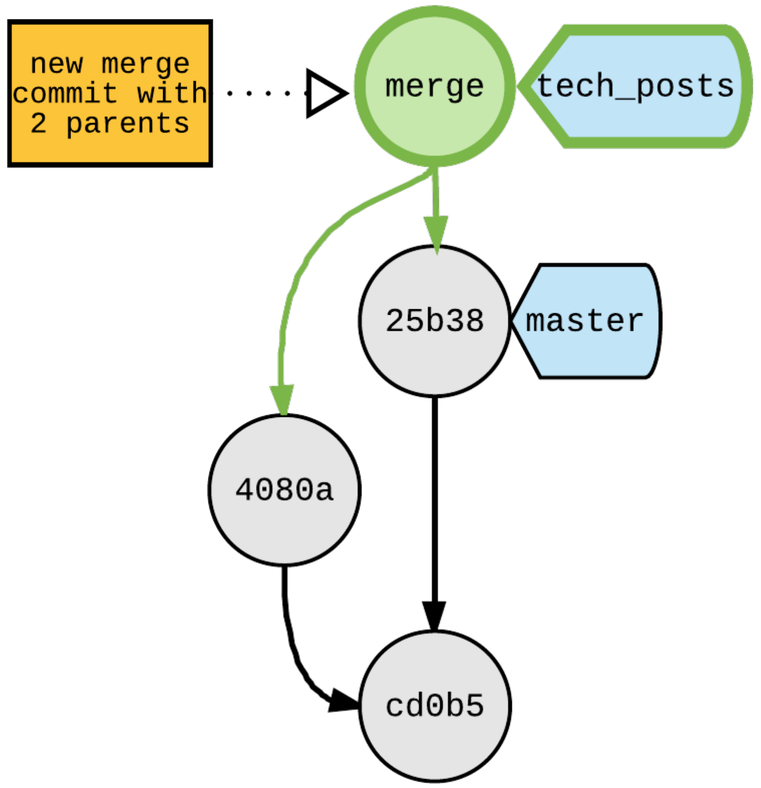


MERGE VS REBASE

scenario: on tech_posts branch



git rebase master



git merge master

POWER OF REBASING – REPLAY COMMITS

- Commits can be:
  - edited
  - removed
  - combined
  - re-ordered
  - inserted
- **Before** they're *"replayed"* on top of the new HEAD.

INTERACTIVE REBASE (REBASE -I OR REBASE --INTERACTIVE)

- Interactive rebase opens an editor with a list of “todos”
  - in the format of: `<command> <commit> <commit msg>`
  - git will pick the commits in the specified order, or stop to take an action when `editing` or a conflict occurs.

- interactive rebase with a shortcut:
  - `git rebase -i <commit_to_fix>^`
  - (the `^` specifies the parent commit)

REBASE OPTIONS

- **pick**
  - keep this commit
- **reword**
  - keep the commit, just change the message
- **edit**
  - keep the commit, but stop to edit more than the message
- **squash**
  - combine this commit with the previous one. stop to edit the message
- **fixup**
  - combine this commit with the previous one. keep the previous commit message
- **exec**
  - run the command on this line after picking the previous commit
- **drop**
  - remove the commit (*tip:* if you remove this line, the commit will be dropped too!)

EXAMPLE REBASE

```
❯ git log --oneline
a37be8d (HEAD -> tech_posts) Oops, I forgot an index entry for my new post
6857c3d posts/django-framework.txt
2733233 Add a blog post about Python

❯ git rebase -i 6857c3d^
```

editor will open

```
reword 6857c3d posts/django-framework.txt
squash a37be8d Oops, I forgot an index entry for my new post
```

editor opens twice.
1. reword (change commit)
2. squash (combine commits)

```
❯ git log --oneline
9413427 (HEAD -> tech_posts) Add a new blog post explaining the pros and cons
of django
2733233 (master) Add a blog post about Python
25b3810 Add a README file to the project
cd0b57c (tag: v1.0, tag: my-first-commit) Initial commit
```

TIP: USE REBASE TO SPLIT COMMITS

Editing a commit can also split it up into multiple commits!

1. Start an interactive rebase with `rebase -i`
2. mark the commit with an `edit`
3. `git reset HEAD^`
4. `git add`
5. `git commit`
6. repeat (4) & (5) until the working area is clean!
7. `git rebase --continue`

TIP: “AMEND” ANY COMMIT WITH FIXUP & AUTOSQUASH!

What if we want to amend an arbitrary commit?

1. `git add` new files
2. `git commit --fixup <SHA>`
    1. this creates a new commit, the message starts with ‘fixup!’
3. `git rebase -i --autosquash <SHA>^`
4. git will generate the right todos for you! just save and quit.

FIXUP & AUTOSQUASH EXAMPLE

```
❯ git log --oneline
9413427 (HEAD -> tech_posts) Add a new blog post explaining the pros and cons
of django
2733233 (master) Add a blog post about Python
25b3810 Add a README file to the project
cd0b57c (tag: v1.0, tag: my-first-commit) Initial commit
```

I want to edit commit 27332 (Add blog post about Python)

```
❯ git add posts/python.txt
❯ git commit --fixup 2733233
[tech_posts 5e980a7] fixup! Add a blog post about Python
 1 file changed, 1 insertion(+), 1 deletion(-)

❯ git log --oneline
5e980a7 (HEAD -> tech_posts) fixup! Add a blog post about Python
9413427 Add a new blog post explaining the pros and cons of django
2733233 (master) Add a blog post about Python
25b3810 Add a README file to the project
cd0b57c (tag: v1.0, tag: my-first-commit) Initial commit
```

git added new commit 5e980
message is: fixup! Add a blog post about Python

FIXUP & AUTOSQUASH EXAMPLE

```
› git rebase -i --autosquash 25b3810^
```

use the -i and --autosquash flags

use ref of the parent commit

```
pick 25b3810 Add a README file to the project
pick 2733233 Add a blog post about Python
fixup 5e980a7 fixup! Add a blog post about Python
pick 9413427 Add a new blog post explaining the pros and cons of django
```

git will automatically reorder commits, and mark the commit with fixup action

```
› git log --oneline
68377f4 (HEAD -> tech_posts) Add a new blog post explaining the pros and cons
of django
a8656d9 Add a blog post about Python
25b3810 Add a README file to the project
cd0b57c (tag: v1.0, tag: my-first-commit) Initial commit
```

shiny new fixed up commit!

REBASE --EXEC (EXECUTE A COMMAND)

```
$ git rebase -i –exec “run-tests” <commit>
```

2 options for exec:

1. add it as a command when doing interactive rebase
2. use it as a flag when rebasing

- When used as a flag, the command specified by exec will run after every commit is applied.
- This can be used to run tests.
- The rebase will stop if the command fails, giving you a chance to fix what’s wrong.

PULL THE RIP CORD!

- At any time before rebase is done, if things are going wrong:
  - `git rebase --abort`

REBASE PRO TIP

- Before you rebase / fixup / squash / reorder:
- Make a copy of your current branch:
  - `git branch my_branch_backup`
- git branch will make a new branch, without switching to it

- If rebase “succeeds” but you messed up…
- `git reset my_branch_backup --hard`

- You’re back in business!

REBASE ADVANTAGES

- Rebase is incredibly powerful!
- You can slice and dice your git history.
- It’s easy to fix previous mistakes in code.
- You can keep your git history neat and clean.

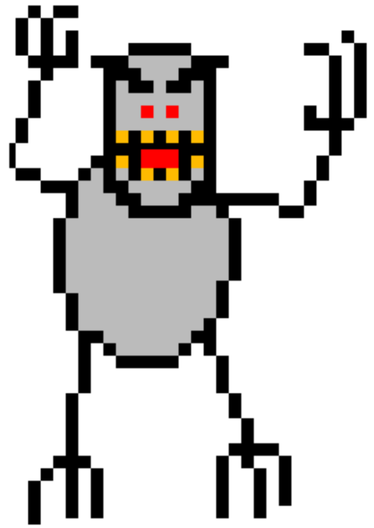
COMMIT EARLY & OFTEN VS GOOD COMMITS

- Git Best Practice:
  - “commit often, perfect later, publish once”

- When working locally:
  - Commit whenever you make changes!
  - It’ll help you be a more productive developer.
- Before pushing work to a shared repo:
  - Rebase to clean up the commit history

WARNING: NEVER REWRITE PUBLIC HISTORY!

- Rebase commits are copies
- If other people are working on the same repository they would be working on different commits.
- You could cause massive merge conflicts
- Even worse, you can cause people to lose their work!



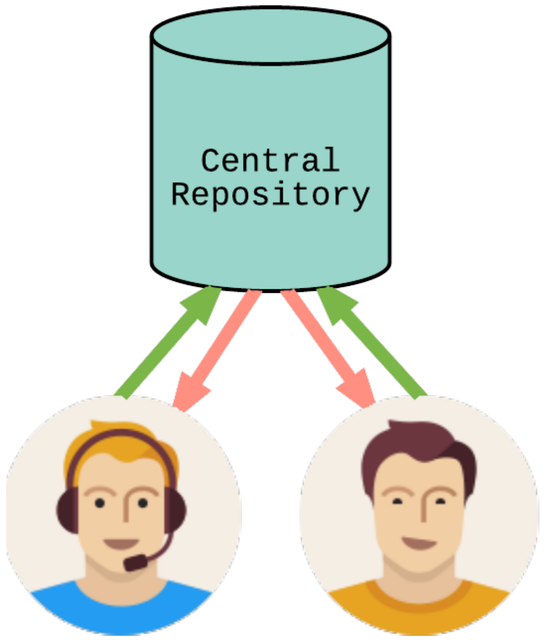
```
Warning: If you rewrite history and push to
    a public repo, monsters will eat you!
```

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise7-RebaseAndAmend.md

FORKS, REMOTE REPOS

DISTRIBUTED VERSION CONTROL



Centralized

Distributed

GITHUB VS GIT – THE KEY IS COLLABORATION

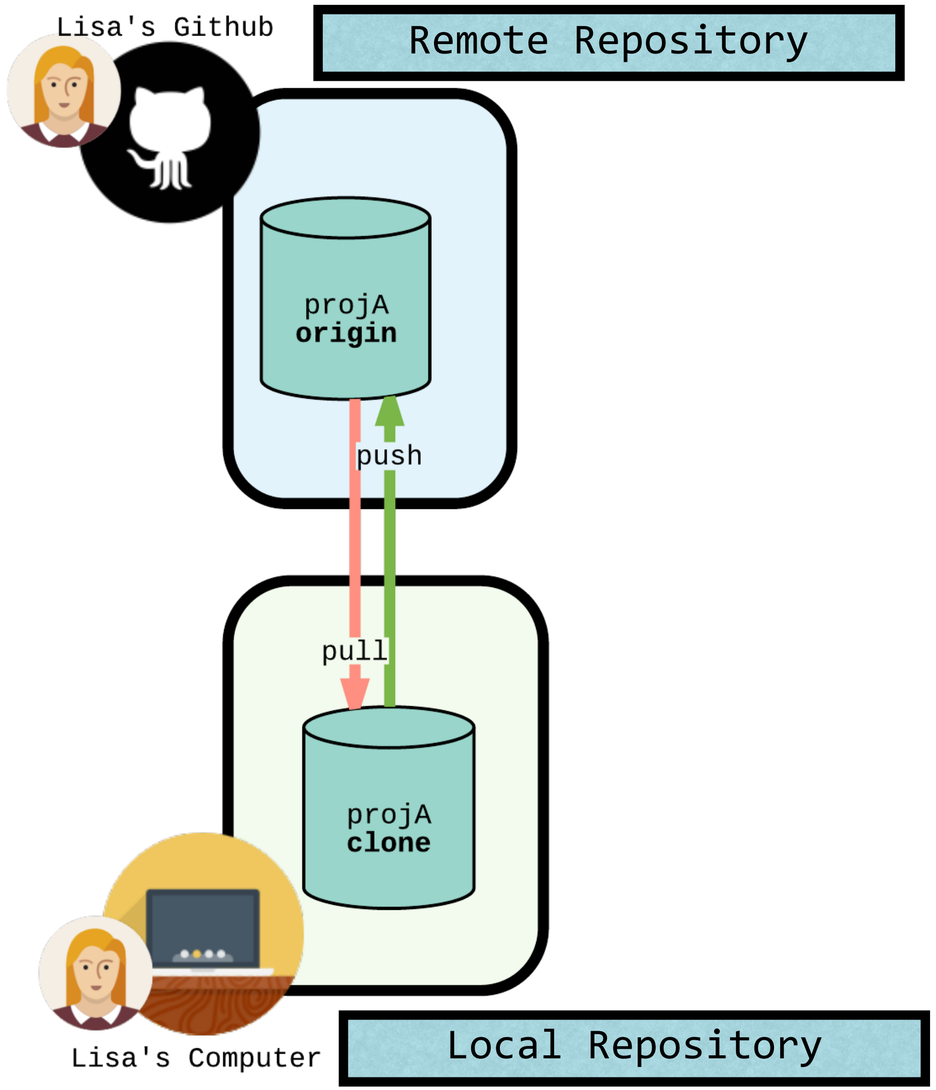
- Git:
  - Open source version control software
- Github:
  - Repository hosting
  - Browse code
  - Issues
  - Pull Requests
  - Forks

REMOTES

- A remote is a git repository stored elsewhere - on the web, in github, etc.
- ***origin*** is the default name git gives to the server you cloned from.
- Cloning a remote repository from a URL will fetch the whole repository, and make a local copy in your .git folder.
- You may have different privileges for a remote.
  - Read/Write for some, Read Only for others.

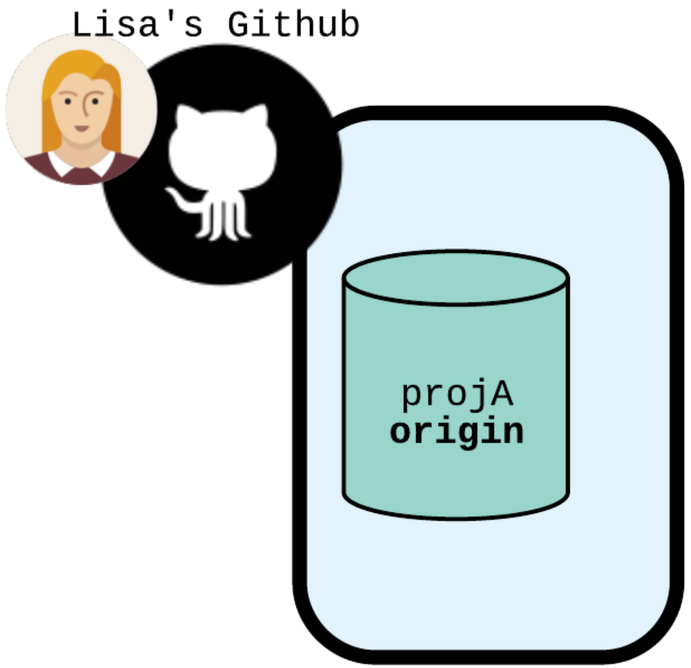
CLONE REPOSITORY

`git clone git@github.com:lisa/projA.git`



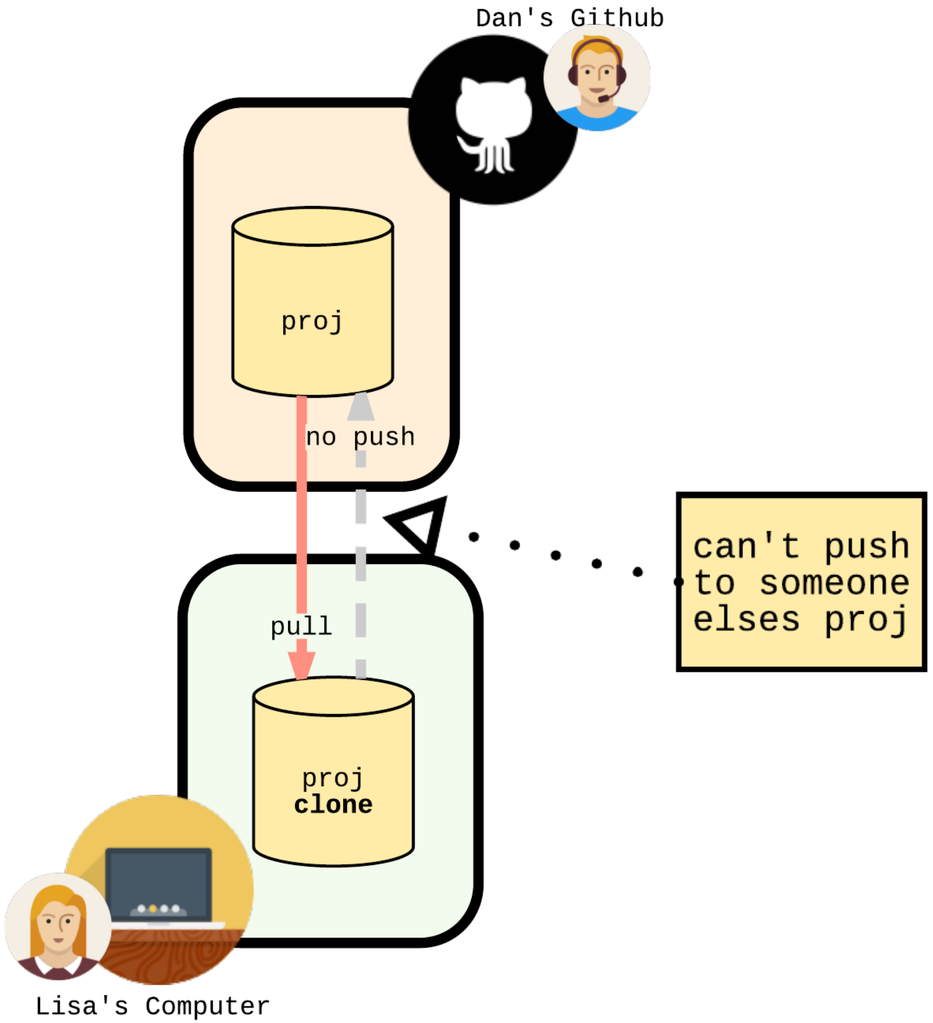
VIEWING REMOTES

```
❯ git remote -v
origin  git@github.com:lisa/projA.git (fetch)
origin  git@github.com:lisa/projA.git (push)
```



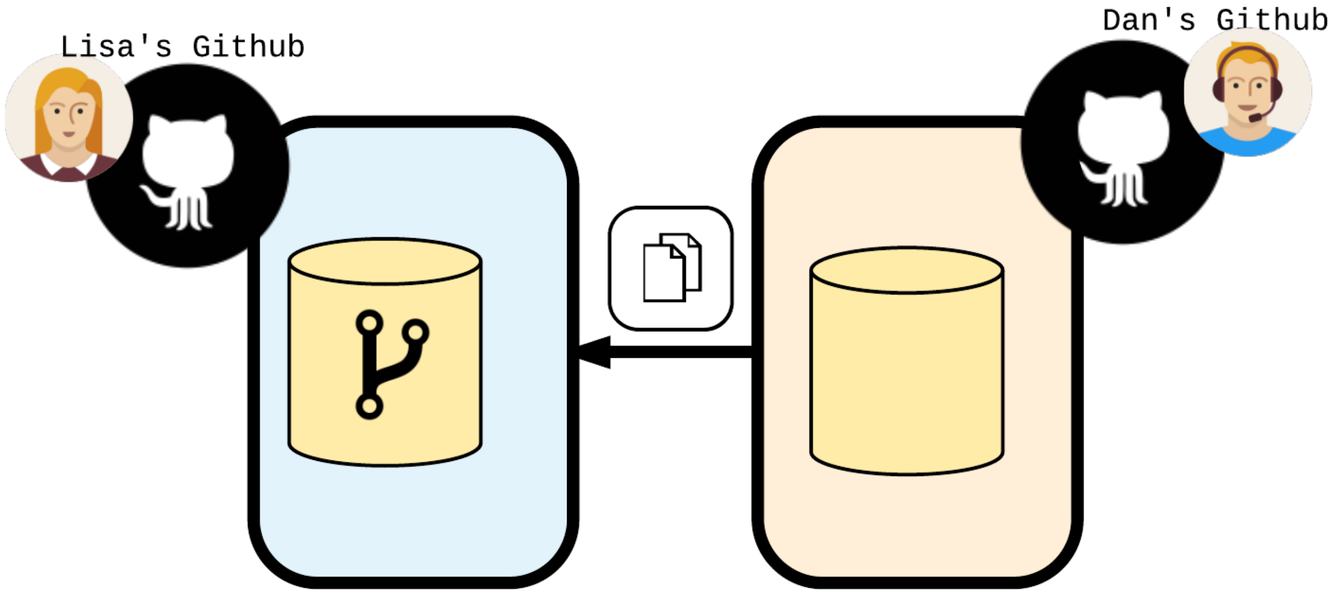
CLONED SOMEONE ELSE'S REPOSITORY

```
git clone git@github.com:dan/projB.git
```

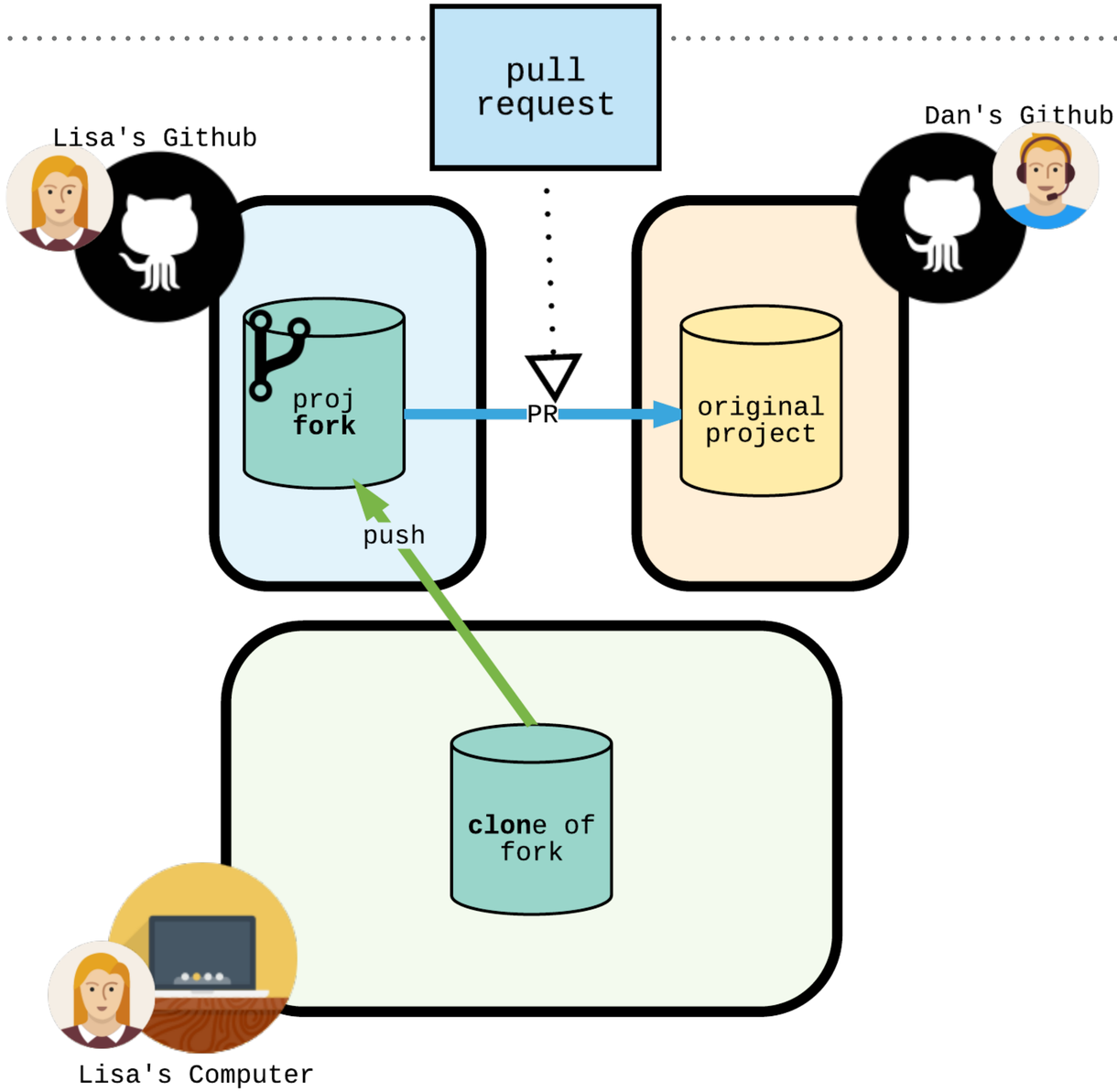


FORK

- A fork is a copy of a repository that's stored in your GitHub account.
- You can clone your fork to your local computer.

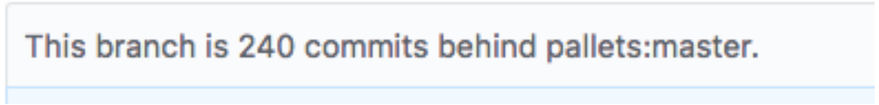


MERGING CHANGES TO ORIGINAL PROJECT FROM A FORK



STAYING UP TO DATE

- While you work on your fork, other changes are getting merged into the source repository.



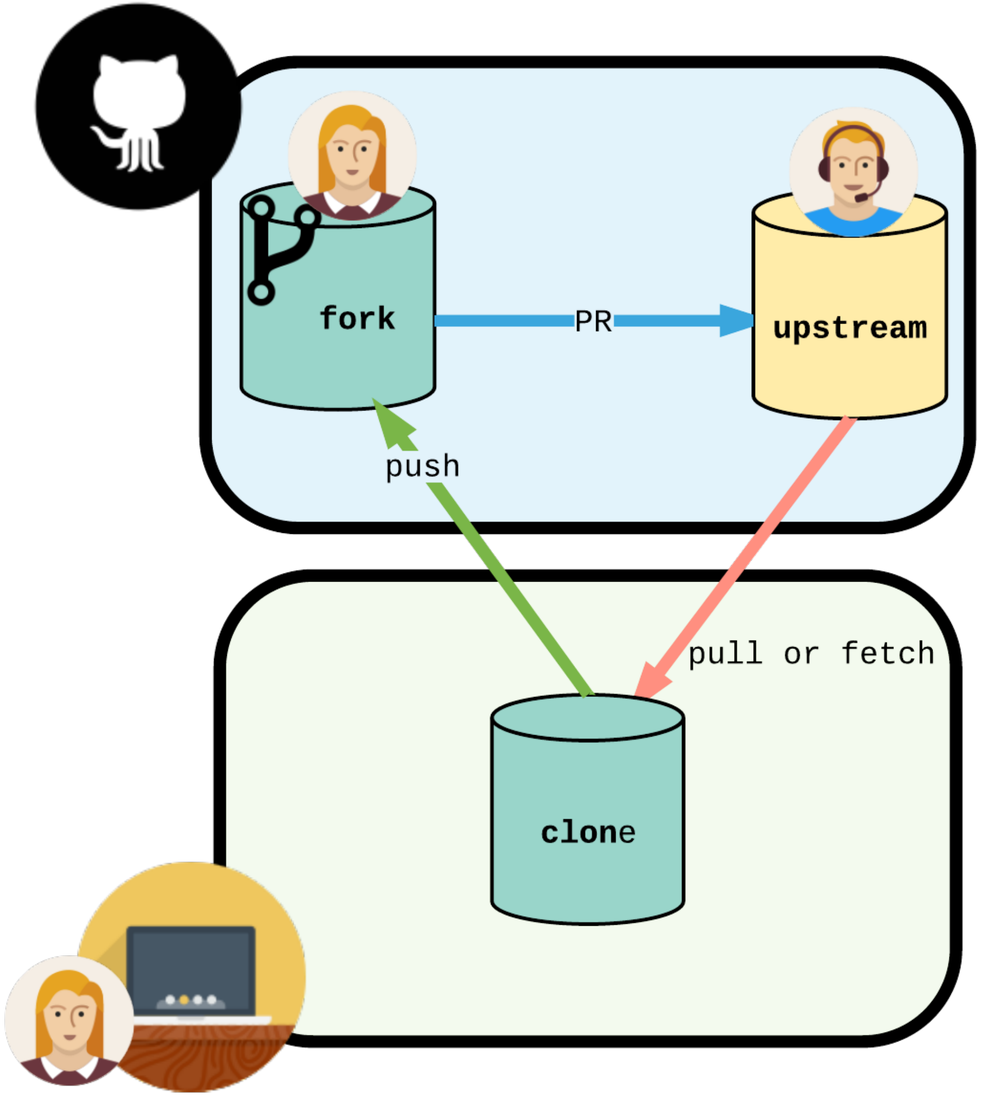
- In order to stay up to date, set up an upstream.

UPSTREAM

- The upstream repository is the base repository you created a fork from.
- This isn't set up by default, you need to set it up manually.
- By adding an upstream remote, you can pull down changes that have been added to the original repository after you forked it.

```
❯ git remote add upstream https://github.com/ORIG_OWNER/REPO.git
```

TRIANGULAR WORKFLOW



TRACKING BRANCHES

- Track a branch to tie it to an upstream branch.
  - Bonus: Use git push / pull with no arguments

- To checkout a remote branch, with tracking:
  - `git checkout -t origin/feature`
- Tell Git which branch to track the first time you push:
  - `git push -u origin feature`

TRACKING BRANCHES

```
❯ git branch
* master

❯ git fetch

❯ git branch -vv
* master bbeb1c32 [origin/master: behind 124] Merge branch 'master' of …
```

which upstream branch is being tracked

how many commits you're ahead or behind

FETCH

- Git fetch is important for keeping your local repository up to date with a remote.
- It pulls down all the changes that happened on the server
- But, it doesn’t change your local repository!

PULL

- Pulling will pull down the changes from the remote repository to your local repository, and merging them with a local branch.

- Under the hood:
  - `git pull = git fetch && git merge`

- If changes happened upstream, git will create a merge commit.
- Otherwise, it will fast-forward.

PUSH

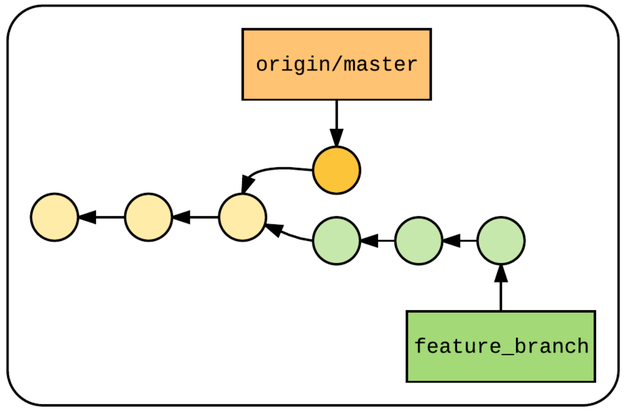
- Pushing sends your changes to the remote repository
- git only allows you to push if your changes won't cause a conflict

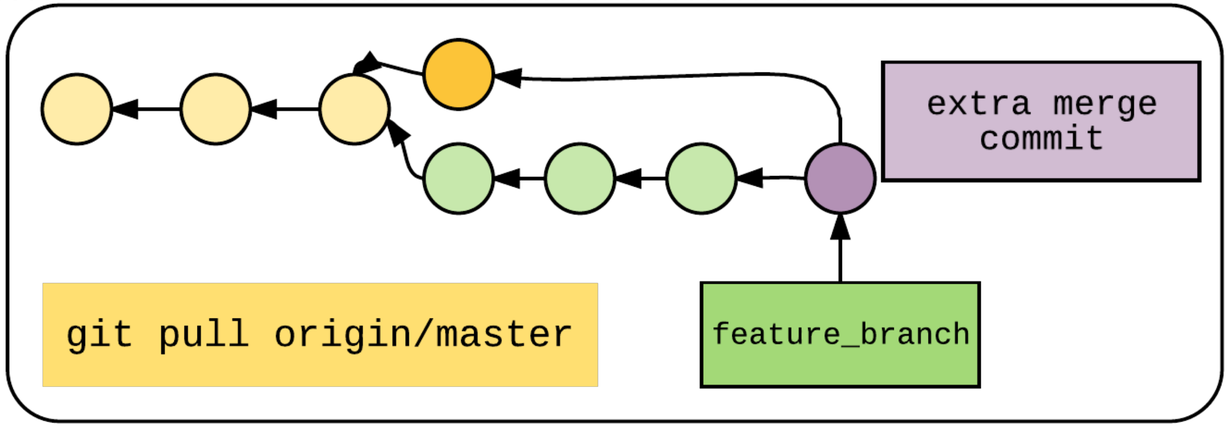
- Tip:
  - To see commits which haven't been pushed upstream yet, use:
    - `git cherry -v`

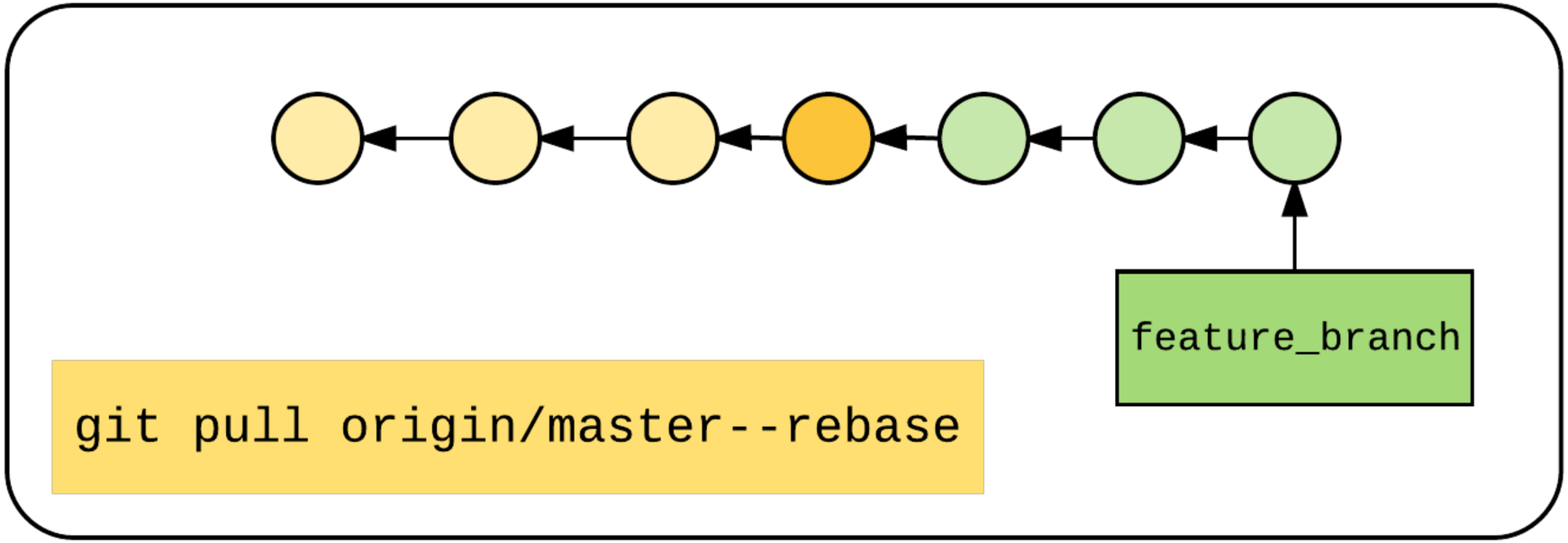
GIT PULL —REBASE

- Git pull —rebase will fetch, update your local branch to copy the upstream branch, then replay any commits you made via rebase.
- Bonus: When you open a PR, there will be no unsightly merge commits!

GIT PULL VS GIT PULL —REBASE







NOTE: TAGS

- Git doesn't automatically push local tags to a remote repository.
- To push tags:
  - `git push <tagname>`
  - `git push --tags`

CONTRIBUTING TO OPEN SOURCE PROJECTS – PULL REQUESTS

- Before opening a PR:
  - Keep commit history clean and neat. Rebase if needed.
  - Run projects tests on your code
  - Pull in Upstream changes (preferably via rebase to avoid merge commits)
  - check for a CONTRIBUTING (.md/.txt) in the project root

- After opening a PR:
  - Explain your changes throughly in the pull request
  - Link to any open issues that your pull request might fix
  - Check back for comments from the maintainers

ADVICE

- Encourage developers to work on their own forks of a repository.
- Mistakes are less likely to happen if no one is pushing directly to the “source of truth” for your codebase!
- You can rebase and force push freely to *your own origin*, as long as no one else is cloning your branch.

PUSHING/MERGING YOUR CHANGES BACK TO A REMOTE

- Rule of thumb:
  - Rebase commits on your local feature branch
  - Merge feature branches back to origin (or upstream)

- When accepting a pull request:
  - Squash and merge or rebase with care.
  - You'll lose context about the work on the feature branch!
  - It'll make it harder to debug when issues arise in the future

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise8-ForksAndRemoteRepos.md

DANGER ZONE

LOCAL DESTRUCTIVE OPERATIONS

- `git checkout -- <file>`
  - If the file is present in the staging area, it'll be overwritten.
- `git reset --hard`
  - Will overwrite changes that are staged and in the working area.

- Unless changes are stashed, there's ***no way*** of getting them back!
- Tip: use `git stash --include-untracked` to include working area changes in your stash

REMOTE DESTRUCTIVE OPERATIONS – REWRITING HISTORY

- There are many operations that can rewrite history:
  - rebase
  - amend
  - reset

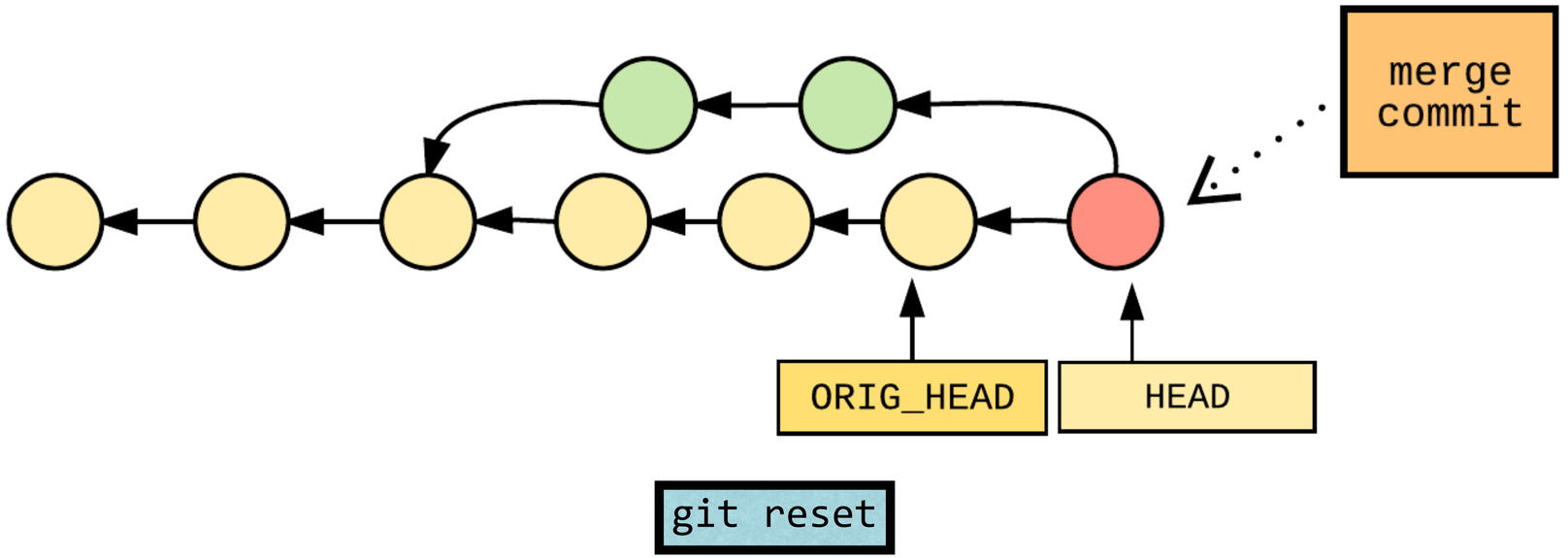
- If your code is hosted or shared:
  - **Never run `git push -f`**

RECOVER LOST WORK

- Use `ORIG_HEAD:`
  - The commit `HEAD` was pointing to before a:
    - reset
    - merge
- Check for repository copies:
  - github
  - coworker

ORIG_HEAD TO UNDO A MERGE

- use `ORIG_HEAD` to undo merges
- `git reset --merge ORIG_HEAD`
- use `--merge flag` to preserve any uncommitted changes



USING GIT REFLOG AND '@' SYNTAX

- By default, git keeps commits around for about 2 weeks.
- If you need to go back in time, and find a commit that's no longer referenced, you can look in the reflog.
- The syntax of reflog is different.
  - `HEAD@{2}` means "the value of HEAD 2 moves ago"

```
❯ git reflog
e7cd68b (HEAD -> dont_overwrite_vary_header, origin/dont_overwrite_vary_header) HEAD@{0}: reset: m
94c49a4 HEAD@{1}: commit: A
e7cd68b (HEAD -> dont_overwrite_vary_header, origin/dont_overwrite_vary_header) HEAD@{2}: commit (
overwrite Vary header when setting for cookie access #2317
a54419a HEAD@{3}: rebase finished: returning to refs/heads/dont_overwrite_vary_header
a54419a HEAD@{4}: rebase: Don't overwrite Vary header when setting for cookie
```

ADVANCED TOOLS

GIT GREP – SEARCH YOUR CODE – IT'S BLAZING FAST!

- Only searches files in your repository by default:
  - `git grep -e <regular expression>`
- Search in a file or a path:
  - `git grep -e <expression> -- <file or path>`
- Searches in your staging area with a flag:
  - `git grep --cached -e <regular expression>`

- Example: Find all instances of 'json' in requests directory:
  - `git grep -e 'json' -- requests/`

GIT GREP OUTPUT – IT'S HARD TO READ

- Use flags to print out the line number, group matches in a single file together, and print a new line between file groups:
  - `git grep --line-number --heading --break -e <expr>`

SUBMODULES

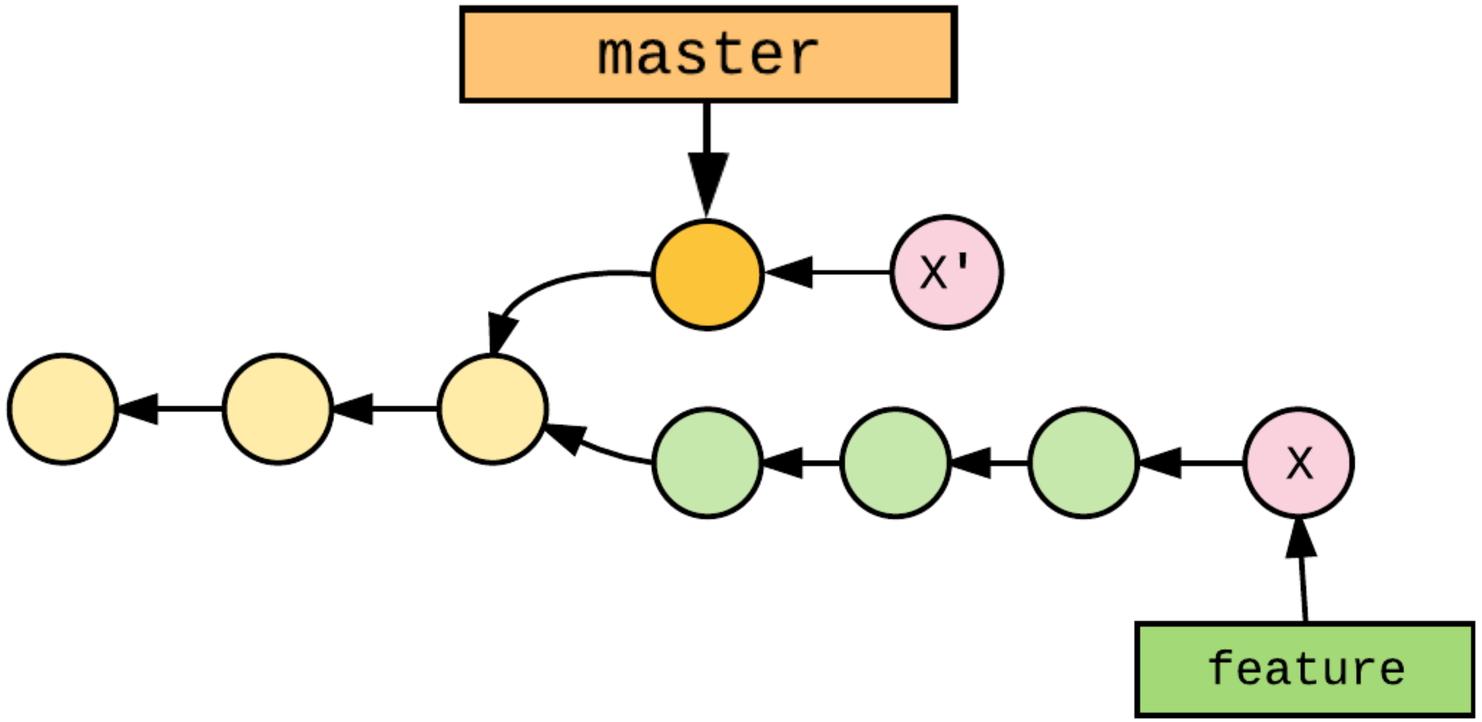
- Git merges allow you to *embed* the contents of another git repository into your repository.
- Git then keeps track of which commit in the other repository your repository points to.
- This may be useful for something like an internal library.
- Avoid them if possible! They can be hard to work with.
- Use a package manager for your programming language instead (i.e. npm for javascript, PyPi for python)

GIT SUBTREE: SUBMODULE ALTERNATIVE

- Git Subtree:
  - add other git projects as dependencies
  - collaborators don’t have to install or updating submodules
  - easy to keep dependencies up to date

CHERRY PICKING

```
❯ git checkout master
❯ git cherry-pick <SHA>
```



GIT BLAME

- The git blame command is used to find out about changes to a file. It can tell you when and by who.
- Useful for looking up commits to figure out *why* a particular change was made.
- The standard format is:
  - git blame <filename>

GIT BLAME WHEN EXAMINING REFACTORING

- Git blame can point out authors who moved code around or changed formatting while refactoring.

- The better format to use is:
  - git blame -w -M -C
    - -w: ignore whitespace
    - -M: detect moved or copied lines within a file
    - -C detect code moved or copied from other files modified in the commit

GIT BLAME A DELETED FILE

- use the diff-filter flag on log to see all commits where the file was deleted:
  - `git log –diff-filter=D -- <deleted_file>`
- git blame using the parent commit
  - `git blame <commit>^ -- <deleted_file>`

GIT BLAME BY LINES OR FUNCTION

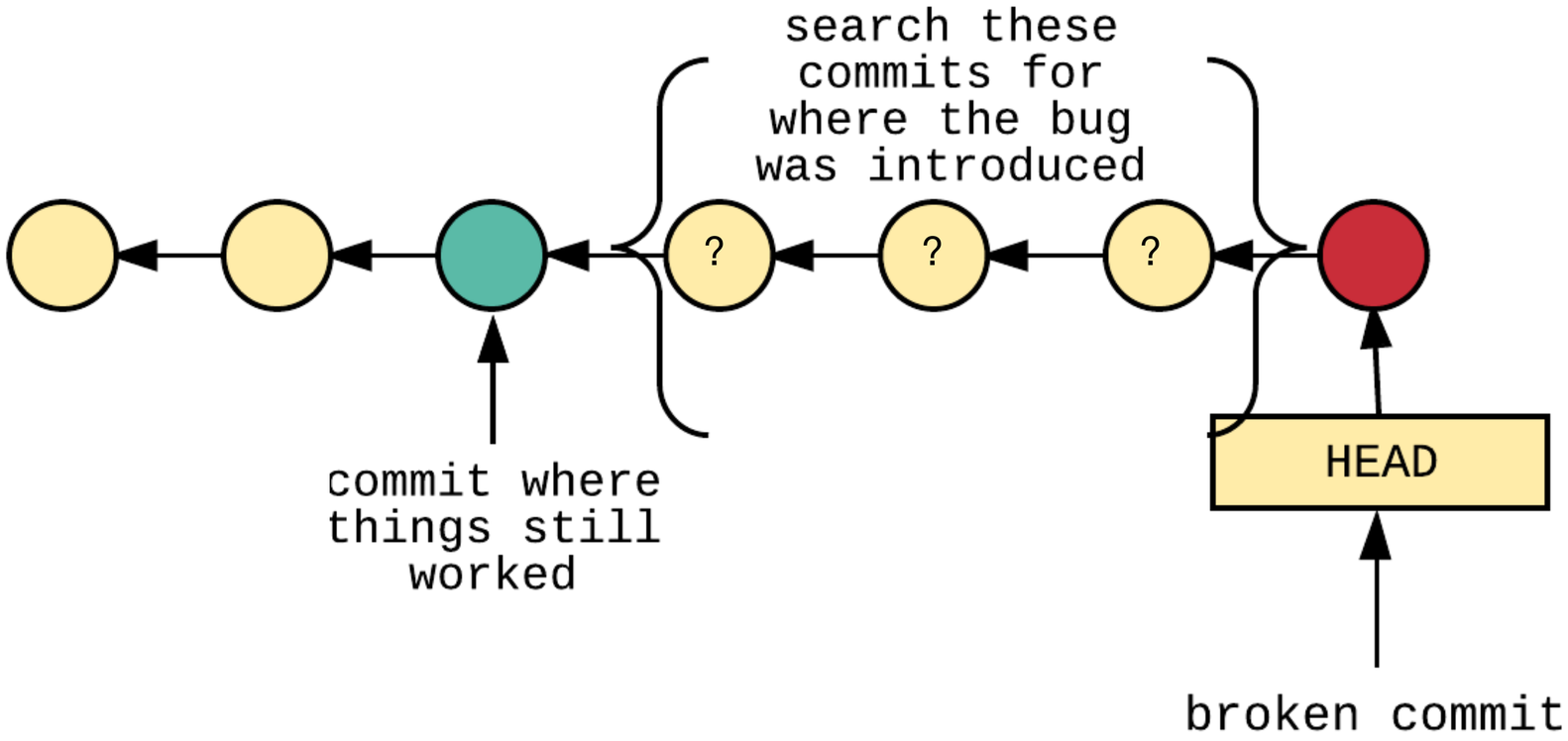
- Use git blame -L!
- It can be used with line numbers *or* a regular expression
- Line numbers:
  - `git blame -L1,5 -- <file>`
  - this example will blame lines 1 through 5
- Regular expression:
  - `git blame -L'/^def get/,/^def/' <file>`
  - this example will blame the python function that starts with ‘def get’ and will search until the next function definition ‘def’

GIT BISECT

- Given a starting point and an end point, search through a commit history to find which commit introduced a bug.
- Git does this via binary search - so it doesn't have to test each commit.
- It's much faster than trying to identify the bad commit on your own.

GIT BISECT

```
› git bisect start <BAD_SHA> <GOOD_SHA>
```



GIT BISECT – TESTING COMMITS

- Git will check out an arbitrary commit between the **GOOD** and **BAD** SHAs.
- You then have two options:
  1. Perform a manual test.
     - Test Passes = run **`git bisect good`**
     - Test Failed = run **`git bisect bad`**
     - Git will check out the next commit
  2. Run a script that returns `SUCESS` (exit code 0) or `FAIL`
     - **`git bisect run`** `<command> <arguments>`

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise9-AdvancedTools.md

CUSTOMIZING GIT

CONFIGURATION: LOCAL VS. GLOBAL

- Local settings are per-repository
  - Stored in:
    - `.git/config`
  - To set: `git config <setting> <value>`
- Global settings apply to all repositories
  - Stored in:
    - `~/.gitconfig`
  - To set: `git config --global  <setting> <value>`

LOCAL VS GLOBAL .GITIGNORE

- Local: `.git/.gitignore`

- Global:
  - Excludes files in *all* git repositories.
  - OSX tip: Ignore `.DS_Store` files!
- Create a file. Example: `~/.gitignore`
- Write a `core.excludesfile` setting:
  - `git config --global core.excludesfile ~/.gitignore`

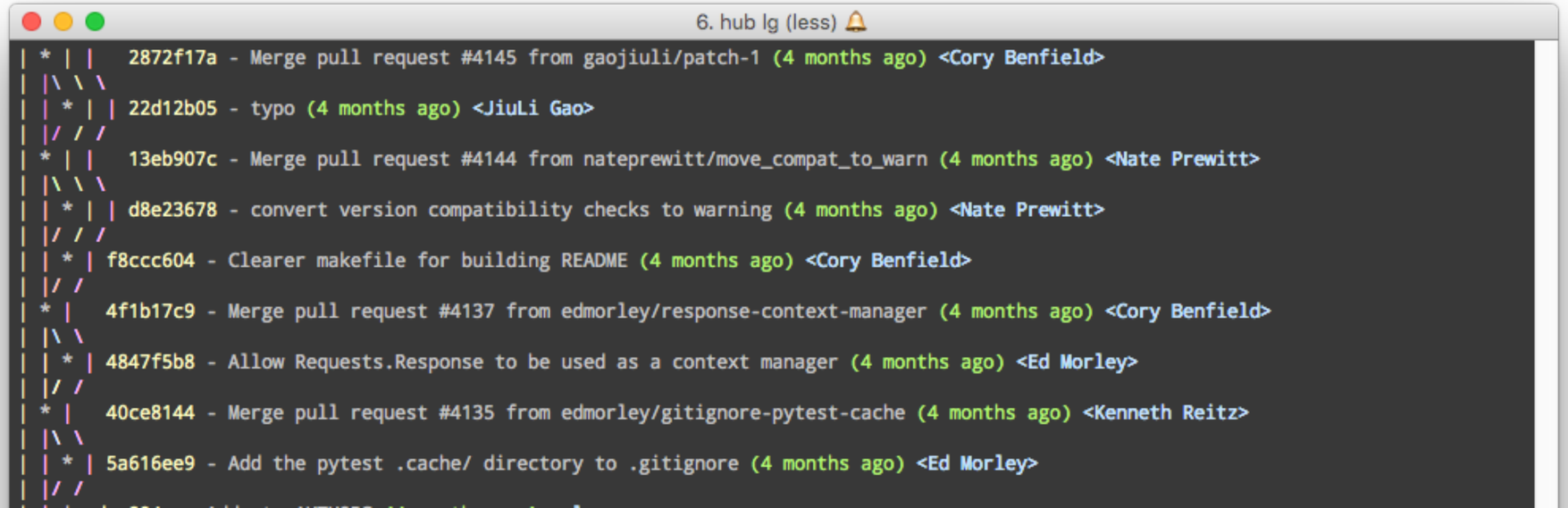
ADDITIONAL USEFUL CONFIG

- Always rebase a branch instead of merging.
  - `git config branch.<branch name>.rebase true`
  - i.e. `git config branch.master.rebase true`
- git rerere (reuse recorded resolution)
  - `git config --global rerere.enabled true`
- Always auto-correct typos
  - `git config --global help.autocorrect 10`

```
› git checkxout master
WARNING: You called a Git command named 'checkxout', which does not exist.
Continuing in 1.0 seconds, assuming that you meant 'checkout'.
Switched to branch 'master'
Your branch is up-to-date with 'upstream/master'.
```

ALIASES & GIT LOG, BEAUTIFIED

- Create aliases:
  - git config --global alias.last "log -1 HEAD"
  - git last <- will show the last commit
- Set up an alias for pretty log:
  - git.io/pretty-logs



GIT HOOKS

- Scripts that run when a git event occurs.
- Located in the .git/hooks directory
- Many types, common ones are:

file	purpose
pre-commit	run before the commit. if script exits with a non-0 status, prevent the commit
post-merge	invoked by git merge if no conflicts occurred
post-checkout	run after git checkout

SAMPLE HOOKS

- Hooks can be any executable file.
  - Shell, python, ruby, etc.
- When you create a new git repository, you'll see some sample hooks in .git/hooks
  - They're confusing and not very helpful!
  - To try one out, remove .sample from the file name.

HOOK EXAMPLE - PRE-COMMIT CHECK FOR SYNTAX ERRORS

- This hook will prevent python files with syntax errors from being commited. http://git.io/hook-example
- To use it, name it .git/hooks/pre-commit and make it executable with chmod +x

1
```
#!/usr/bin/env python

import sys
import subprocess
```
2
```
diff_command = 'git diff --staged --name-only --diff-filter=AM *.py'

output = subprocess.check_output(diff_command.split())
files = output.split('\n') if output else []
```
3
```
for file in files:
    linter_cmd = 'pylint --errors-only %s' % file
    output = subprocess.call(linter_cmd.split())
```
4
```
    if output:
        print 'Error in file %s. \nAborting commit!' % file
        sys.exit(1)  # return 1 (failure)
    else:
        sys.exit(0)  # 0 is success status code
```

HOOK EXAMPLE: POST-MERGE, CHECK FOR NEW DEPENDENCIES

- This hook will warn if requirements.txt changed. http://git.io/merge-hook-example
- To use it, name it .git/hooks/post-merge and make it executable with chmod +x

1
2
3

```
#!/usr/bin/env python
import sys
import subprocess

diff_requirements = 'git diff ORIG_HEAD HEAD --exit-code -- requirements.txt'

exit_code = subprocess.call(diff_requirements.split())
if exit_code == 1:
    print 'The requirements file has changed! Remember to install new
dependencies.'
else:
    print 'No new dependencies.'
```

GIT HOOK RESOURCES

- Github generated .gitignore files for many languages
  - https://github.com/github/gitignore
- For a good resource of pre-written git hooks, and a framework for writing git hooks in languages other than shell script:
  - https://github.com/pre-commit/pre-commit
- For javascript specific git hooks, check out husky:
  - https://github.com/typicode/husky

GIT TEMPLATES

- Git templates allow you to specify files that will be copied into your .git folder after git init is run.

- You can specify which template dir to always use in your config file.
- If you work on projects with multiple programming languages, you can specify which template directory to use with a flag.

GIT TEMPLATE - PYTHON EXAMPLE

For a new python project in git, I'd like the following behaviors:

- Always set up a python style `.gitignore`
- Add a pre-commit hook that lints my files for syntax errors
- Warn me if the dependencies in the requirements file changed after merging

USING A GIT TEMPLATE

```
❯ tree ~/.git-templates/python
/Users/nina/.git-templates/python
└── hooks
    ├── post-merge
    └── pre-commit

1 directory, 2 files

❯ cd ~/code/new_proj

❯ git init --template=/Users/nina/.git-templates/python

❯ ls -l .git/hooks
total 16
-rwxr-xr-x  1 nina  staff  329 Oct  1 18:35 post-merge
-rwxr-xr-x  1 nina  staff  518 Oct  1 18:35 pre-commit
```

GIT TEMPLATE RESOURCES

- Github generated .gitignore files for many languages
  - https://github.com/github/gitignore

GIT LINT

- Git lint is an open source tool that lets you set up linting for different source files easily.
- Works for CSS, Javascript, Python, PHP, and many many others!
- https://github.com/sk-/git-lint

TIME SAVING ALIASES – FOR UNIX AND OS X

- When you use git day in and day out, having aliases helps!
- Add these to your .bashrc file, then run `source ~/.bashrc`
- https://git.io/git-alias

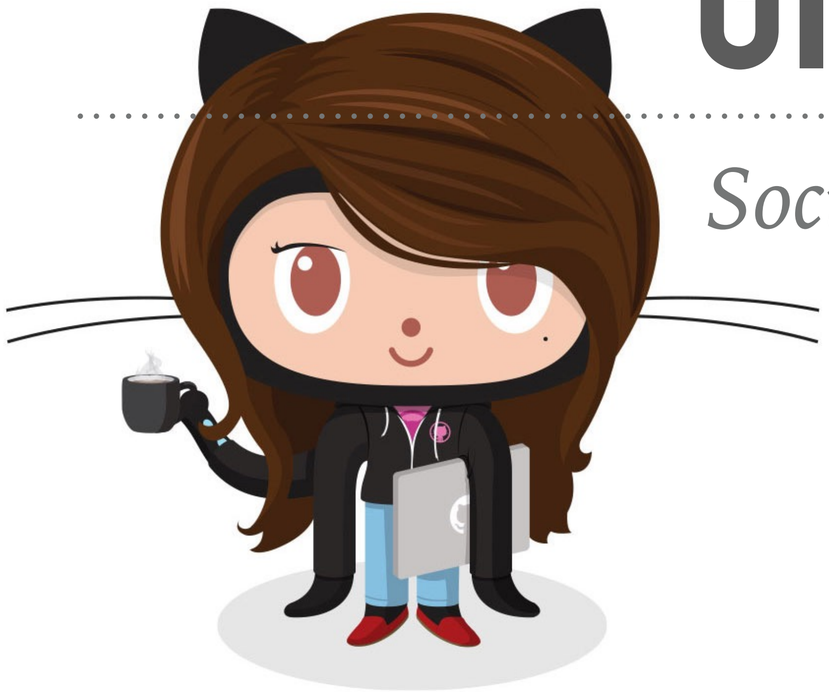
```
alias ga='git add'
alias gp='git push'
alias gpu='git pull'

alias gs='git status'
alias gd='git diff'
alias gds='git diff --staged'

alias gm='git commit -m'
alias gc='git checkout'
```

EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise10-Hooks.md

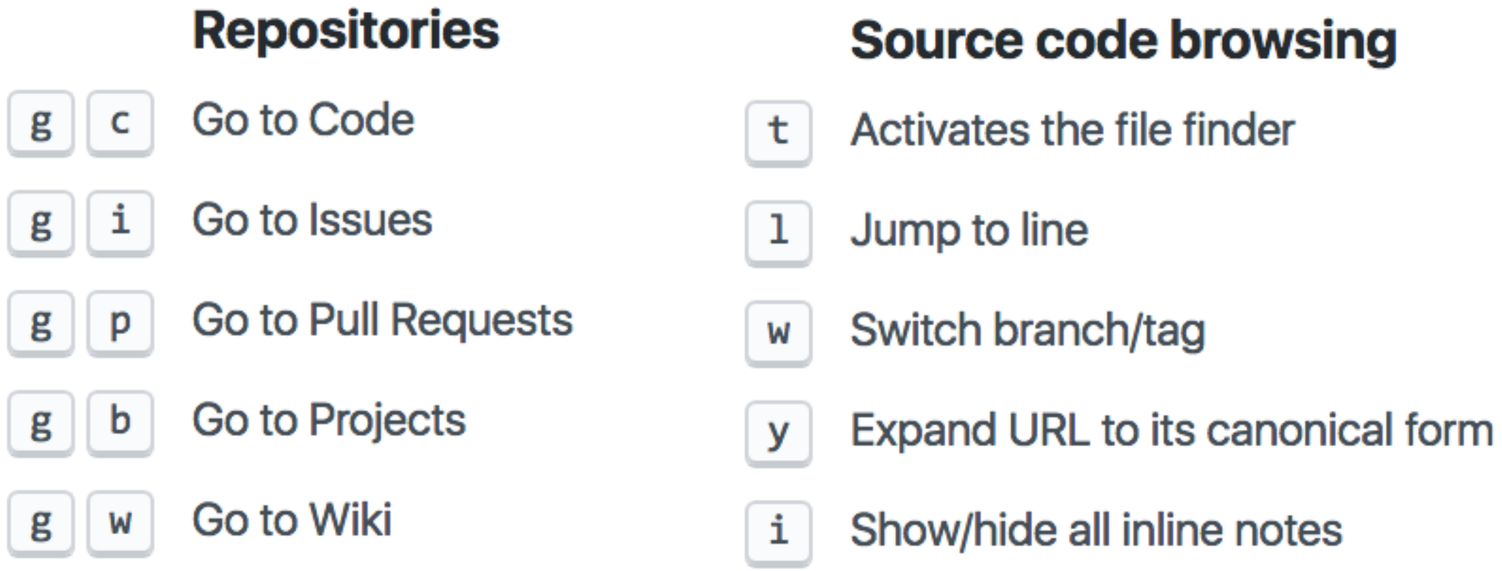
GITHUB

Social Coding



NAVIGATE LIKE A PRO

- Press '?' on any github.com page for a list of shortcuts.
- Then, hit 'Show All'



Repositories

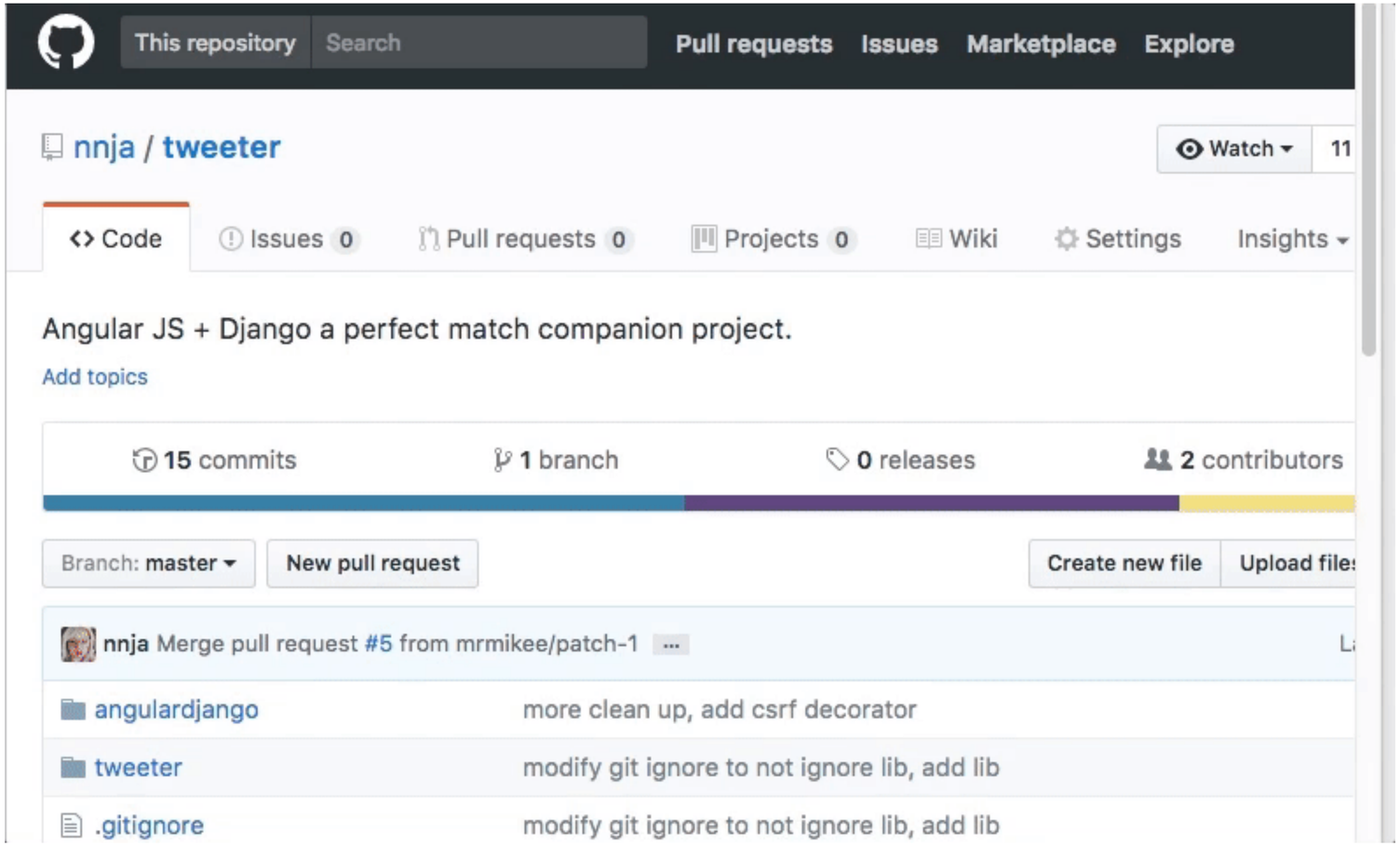
g	c	Go to Code
g	i	Go to Issues
g	p	Go to Pull Requests
g	b	Go to Projects
g	w	Go to Wiki

Source code browsing

t	Activates the file finder
l	Jump to line
w	Switch branch/tag
y	Expand URL to its canonical form
i	Show/hide all inline notes

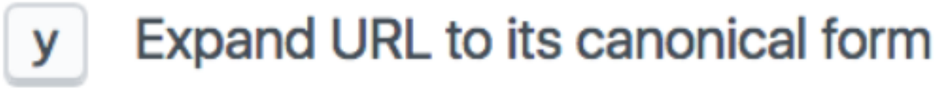
NAVIGATE – GITHUB.COM FILE FINDER

t Activates the file finder



GITHUB - PERMALINK TO CODE

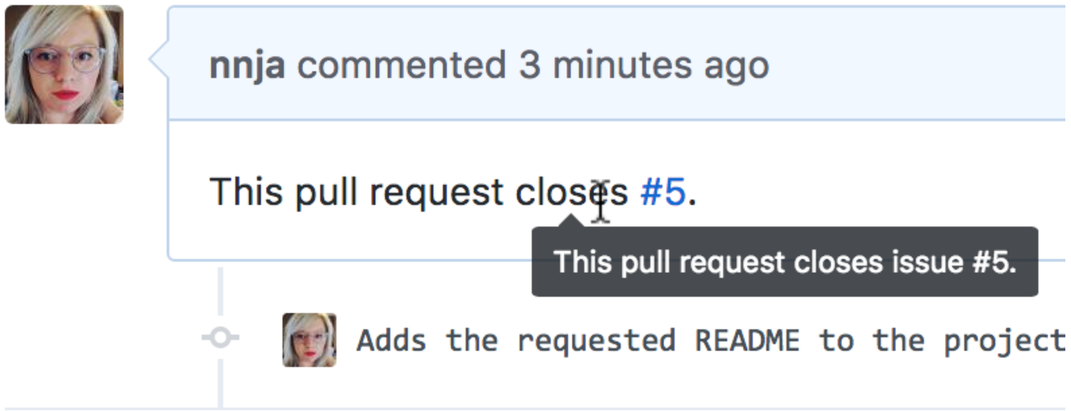
- When viewing code on github, you're looking at latest version on a particular branch
  - i.e. github.com/nnja/tweeter/blob/master/
- If you want to send someone a link to a particular set of changes, and guarantee that they'll see the same code you were looking at, hit 'y' when visiting the file on github.



- This will link to the ***commit instead*** of the *branch*!

AUTOMATICALLY CLOSE ISSUES FROM PR OR COMMIT

- In the pull request, add to the description:
  - fixes #<issue number> or closes#<issue number>
- In a commit, add closes#<issue number>to to message.

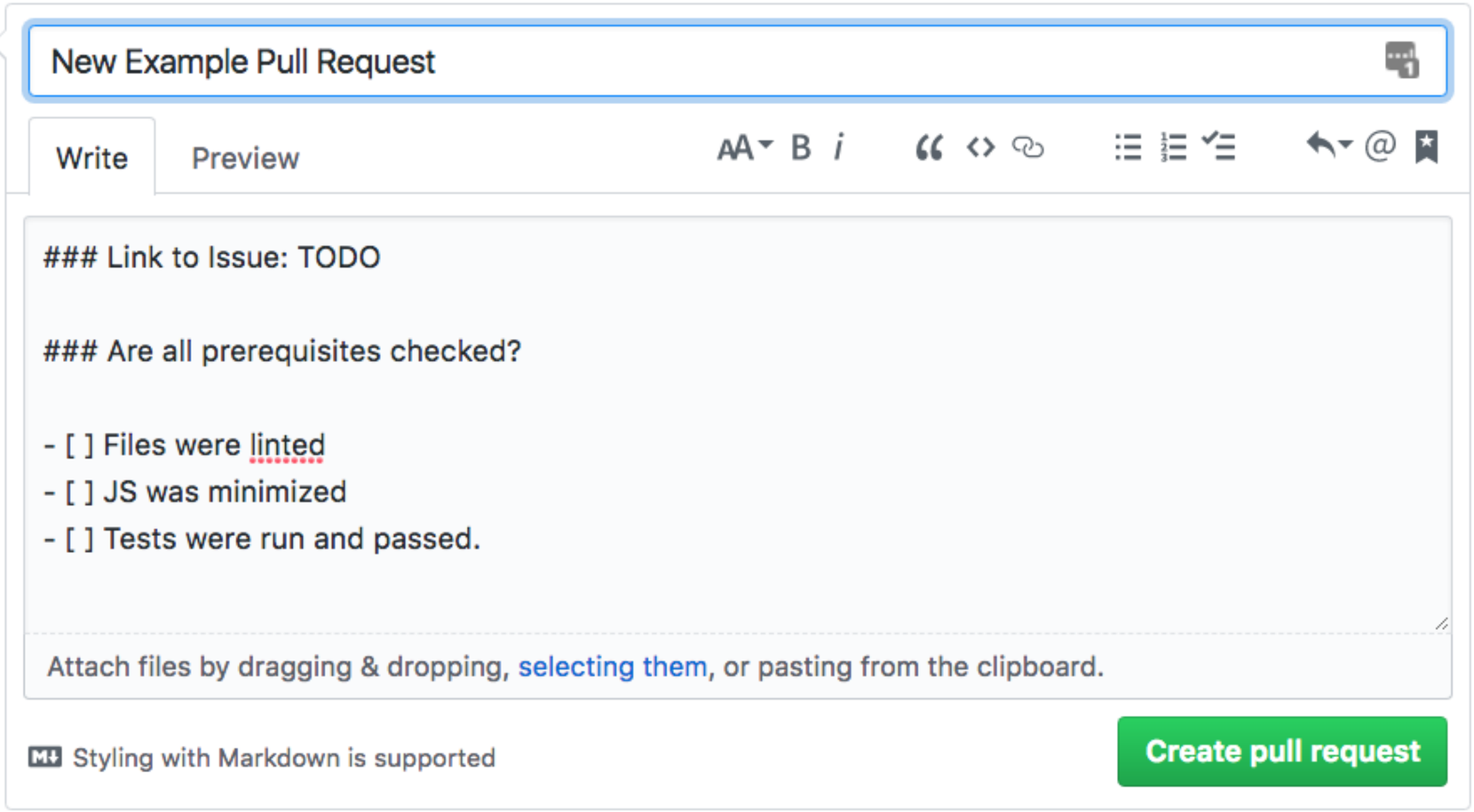


- When the PR or commit is merged, the issue will close.

.GITHUB – TEMPLATES FOR ISSUES, PULL REQUESTS

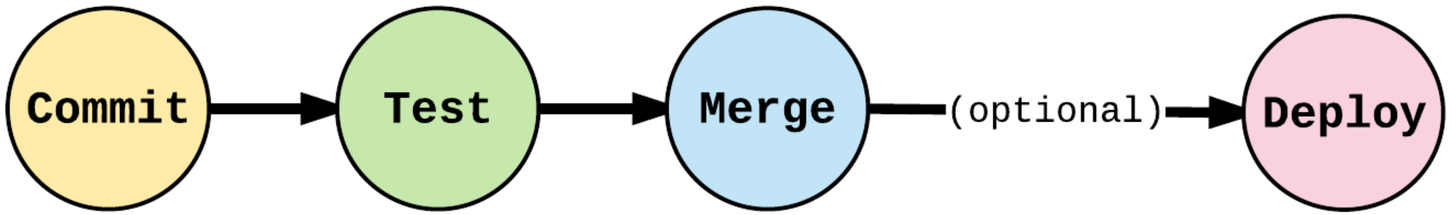
- You can add a .github/ folder for your project to specify templates for pull requests, issues, and contributing guidelines.
- These live in the following files:
  - `CONTRIBUTING.md`
  - `ISSUE_TEMPLATE.md`
  - `PULL_REQUEST_TEMPLATE.md`
- These files will automatically populate the description text when creating a new issue or PR.

CREATING PR FROM .GITHUB TEMPLATE



CONTINUOUS INTEGRATION

- Merging smaller commits frequently, instead of waiting until a project is “done” and doing one big merge.
- This means that features can be released quicker!
- CI only works well when there are tests that ensure that new commits didn’t “break the build”
- It’s even possible to perform a deployment at the end of a CI build!



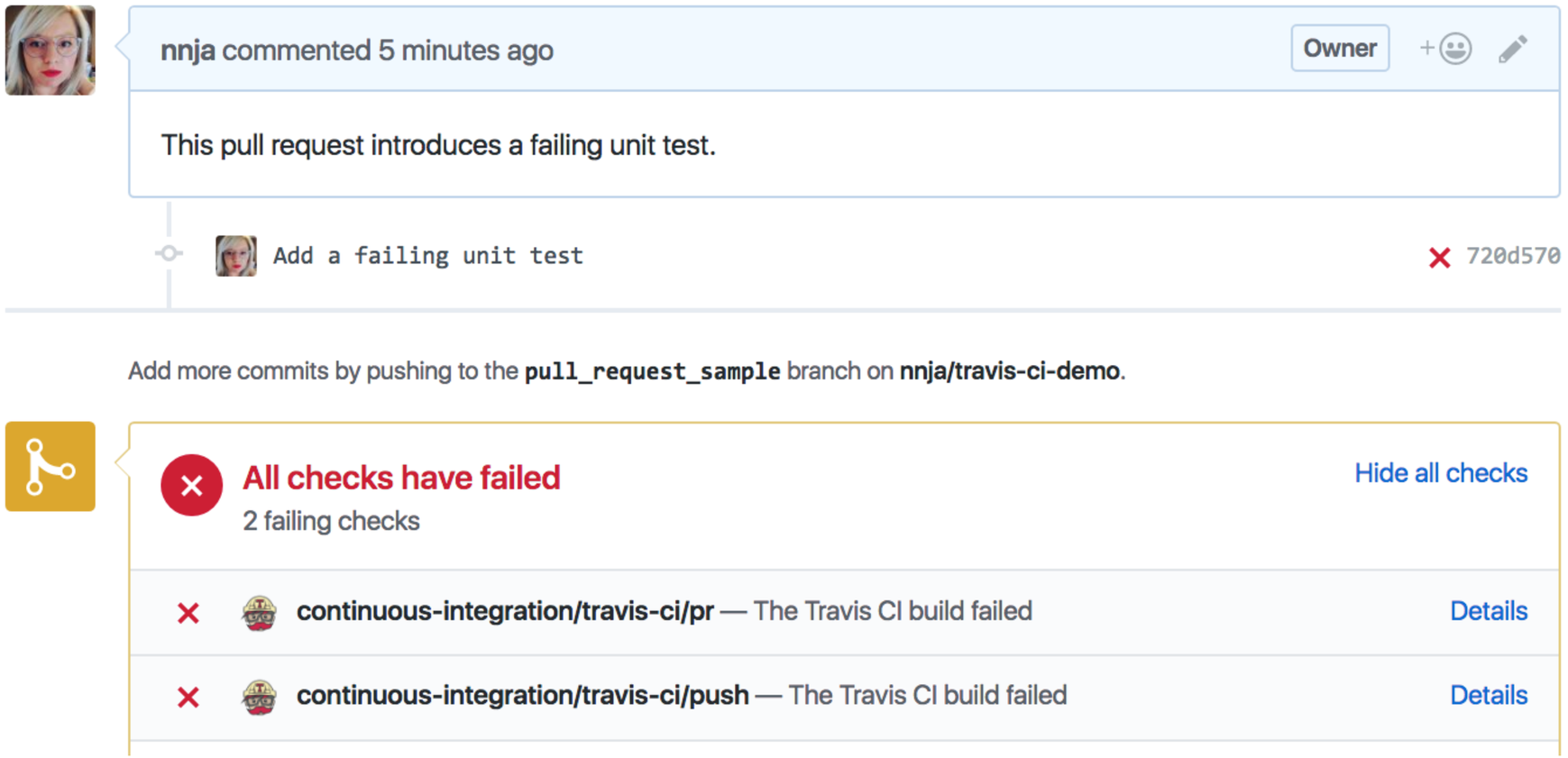
Travis CI INTEGRATES WITH GITHUB

- Travis CI is free for open source projects!
- It's easy to specify what commands you need to run to run tests
- It's also easy to test against multiple versions of a language (python2 vs python3) and even multiple versions of libraries
- Tests run automatically on branches and pull requests

- Getting set up is easy:
  - go to `travis-ci.org`, log in with your github account
  - add a travis.yml configuration file
  - push to trigger builds

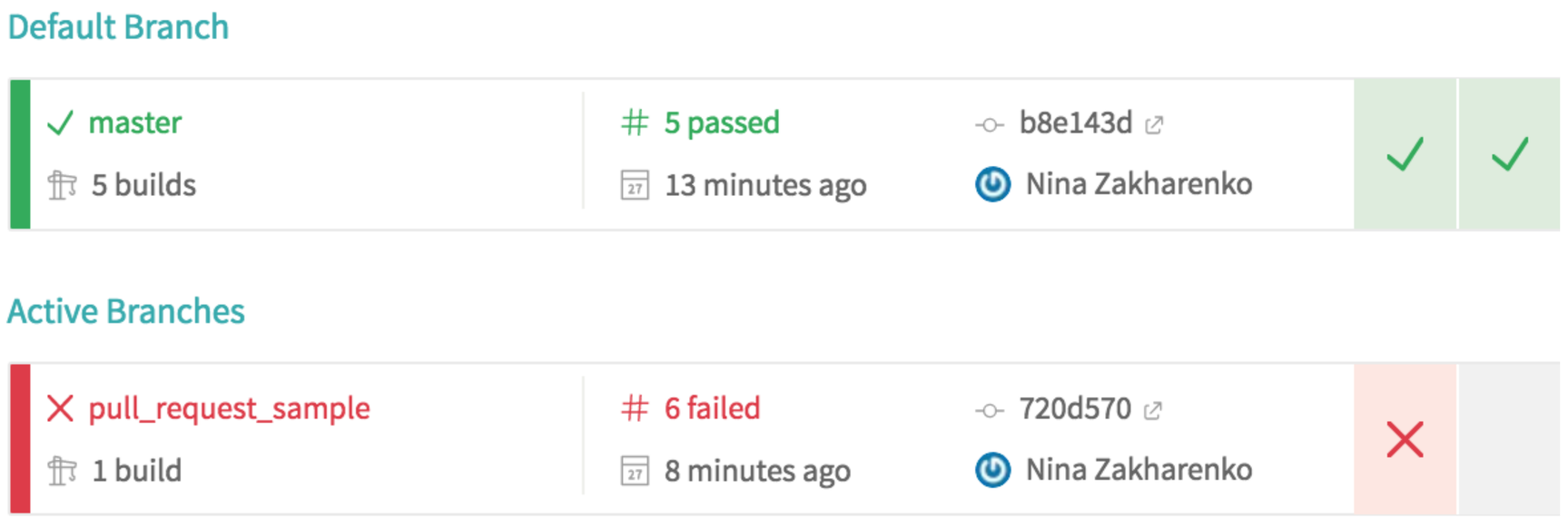
RUN TESTS BEFORE MERGING OR ACCEPTING PR

- Sample pull request: `git.io/travis-ci-demo`



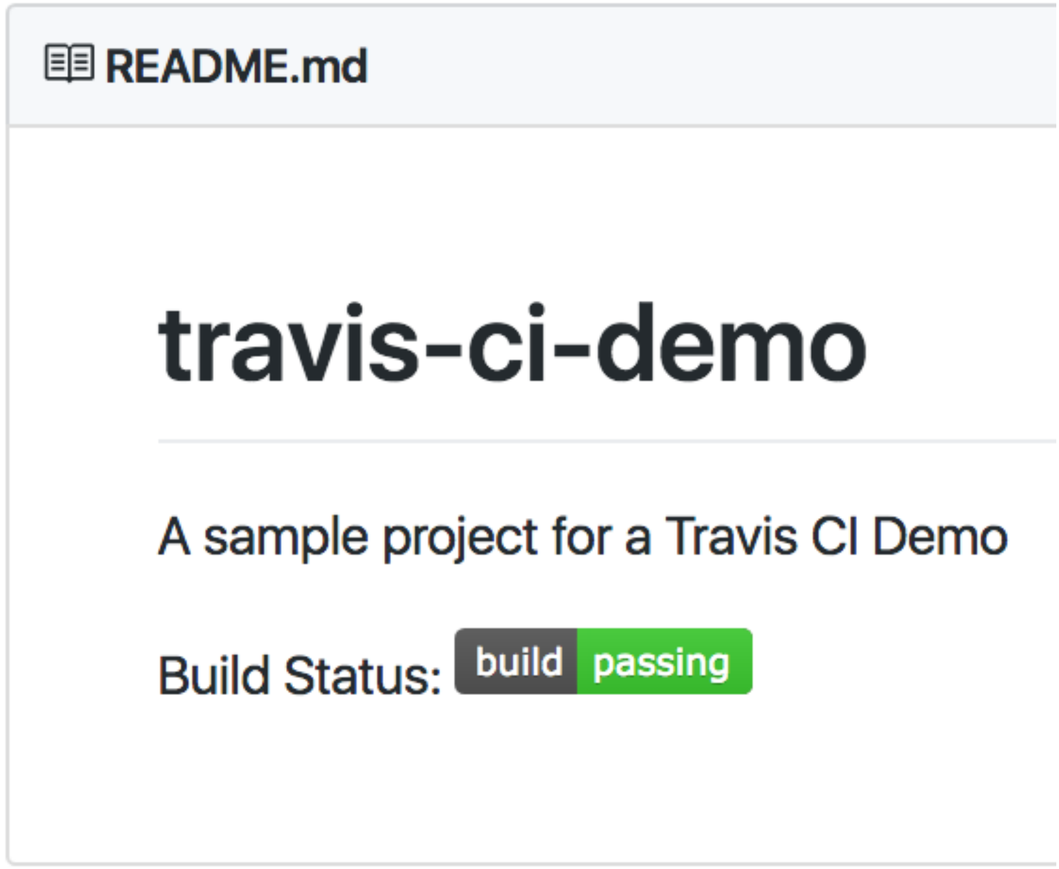
LOOK AT BUILD RESULTS

- visit `travis-ci.org/<username>/<project>`



DISPLAY BUILD STATUS OF YOUR PROJECT

- Add an image to your README to display the build status.
- Instructions: `bit.ly/travis-status`



HUB COMMAND LINE TOOL – HUB.GITHUB.COM

- To install hub on osx:

```
❯ brew install hub
❯ alias git=hub
# Add the line above to your .bashrc, then source .bashrc
```

- Open a pull request from the command line:

```
❯ git pull-request -h hub_example
https://github.com/nnja/travis-ci-demo/pull/2
```

- Open a browser page to the project's issues:

```
❯ git browse -- issues
```

- Even make a fork from a cloned repo!

EXERCISE

- Fork `github.com/nnja/travis-ci-demo`
- Navigate to your fork. Hit 't'. Use the browser to look at `travis.yml`
- Visit `travis-ci.org/profile/` and log in with your GitHub account
- Find your fork of the demo, enable travis on it by flipping the switch.

- Click on the repository name, then start the build.

ADVANCED GITHUB

Using the API

GITHUB API

- GitHub has an incredibly powerful RESTful API.
- Currently on Version 3

OCTOCAT COMES IN MANY FLAVORS

- Api reference: developer.github.com/v3/libraries/

Octokit comes in many flavors

Use the official Octokit library, or choose between any of the available third party libraries.

Ruby	octokit.rb
Obj-C	octokit.objc
.NET	octokit.net

…and many more!

MAKING REQUESTS

- Un-authenticated
  - Rate limited. 60 requests per hour
- Personal token
  - Useful for testing, personal projects
  - Requests authenticated as user who owns the token
- OAuth
  - When your application acts on behalf of a user.
  - The user will log-in via the OAuth flow in your project

CREATE AND UPDATE VIA API

- GitHub allows you to create via the API as well.
- It's possible to create and update:
  - Issues
  - Pull Requests
  - New Repositories
  - Gists

STATUSES

- GitHub allows you to list statuses for a specific REF
- Potential statuses:
  - Did the build pass?
  - Is the PR approved by a reviewer?
- Using the status endpoint, integrate with:
  - Slack
  - Custom emails
  - PagerDuty
- `GET /repos/:owner/:repo/commits/:ref/statuses`

LISTENING TO EVENTS – WEBHOOKS

- Create an app that subscribes to events
- Subscribed URLs will receive a POST request with a payload when the event occurs

- Examples:
  - when a user is added as a collaborator to a repo
  - after a push to a repository

- Getting started: developer.github.com/webhooks

FINAL EXERCISE

- https://github.com/nnja/advanced-git/blob/master/exercises/Exercise11-GitHubAPI.md

THE END!

git.io/advanced-git

nina@nnja.io
@nnja

ADDITIONAL RESOURCES

- Git docs online
  - https://www.kernel.org/pub/software/scm/git/docs/
- GitHub API React Examples
  - https://react.rocks/tag/GitHub_API
- Bup: A backup system based on git Pacifies
  - https://bup.github.io/