Git General Training

Deep inspection of basic commands

Ilya Rokhkin
2017

Agenda:

1. Basic concepts and commands
   - Git Architecture, data model
   - DVCS, Repository, Commit, Parent Commit, Tree, Blob, Index.
   - Staged, Modified, Committed files.
   - Basic commands to work in Repository and outside.
   - Sharing work with peers.
   - Best practices

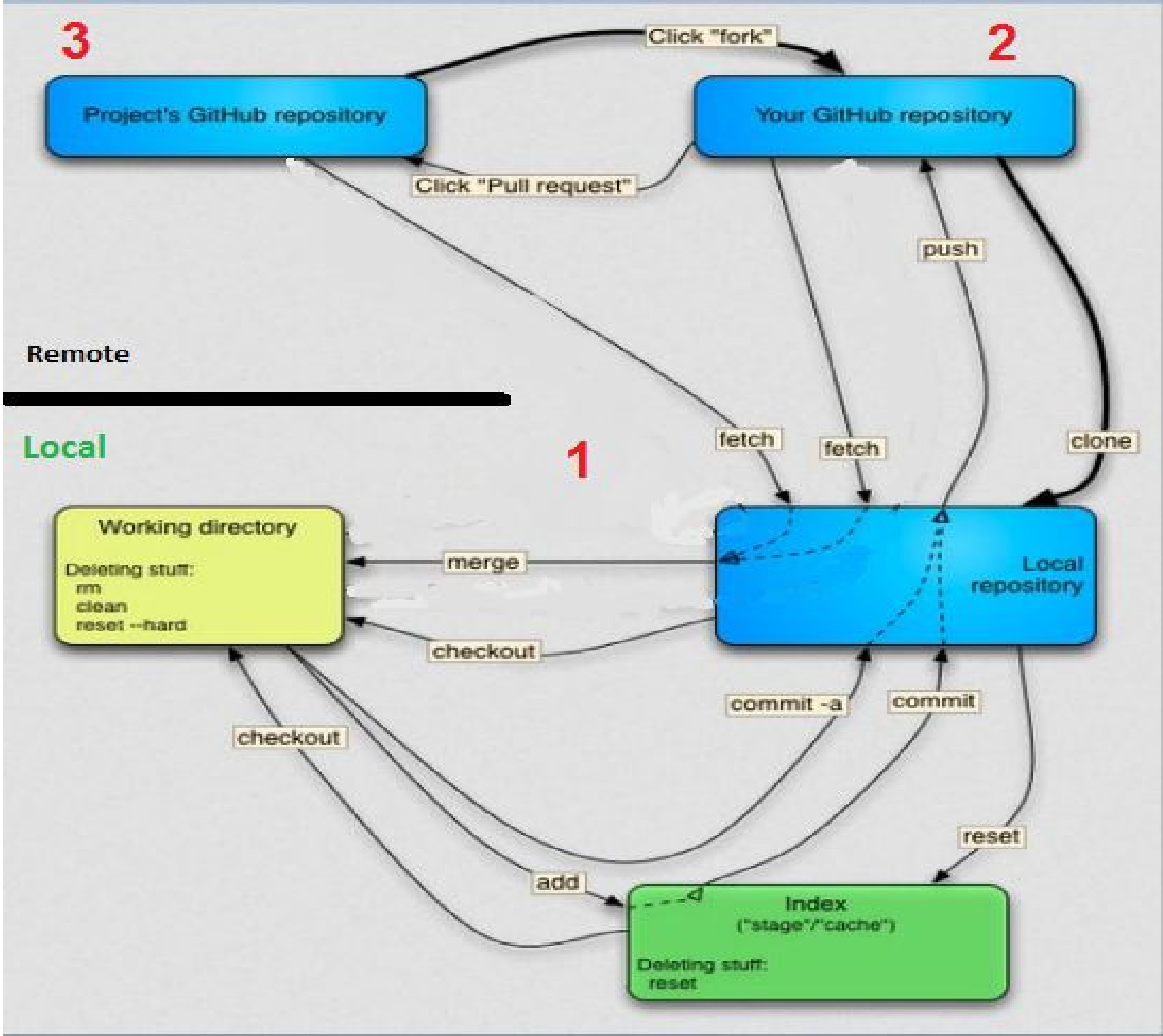
2. Practical part, lab work

GIT Overview

- Quick and efficient
- Expedite distributed development
- Atomic transactions, commit, cross repository
- Commits (Change) management
- A clear internal design
- Suited to handle everything from small to very large projects with speed and efficiency
- Support and encourage branched development

GIT Architecture

1 Local repo
2 Remote (Origin)
3 Common (Community) Remote Origin



Demo

Lab 1 - 2

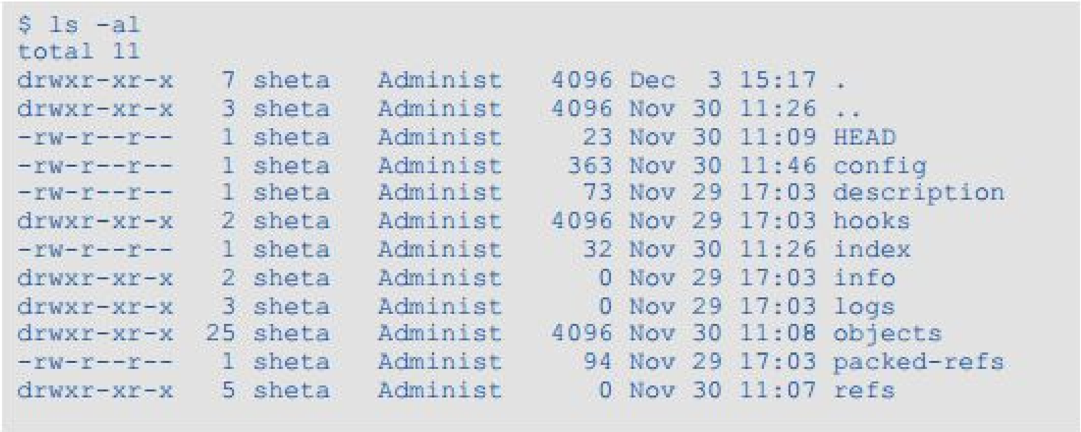
- We will configure your git user and e-mail

  Will create remote, bare repository in home dir

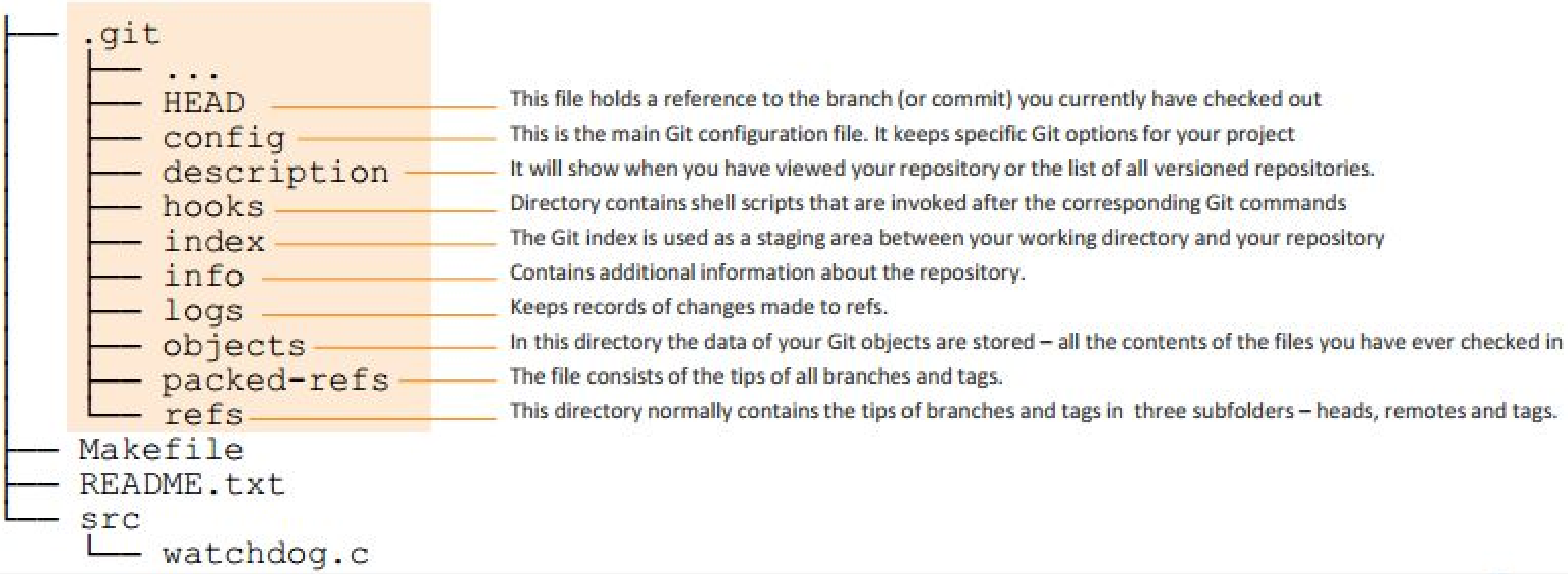
  Clone it to work repo1 in home dir also, add,commit and push

- Restructure files, add, commit and push

Git repository structure (.git)



```
$ ls -al
total 11
drwxr-xr-x   7 sheta   Administ   4096 Dec  3 15:17 .
drwxr-xr-x   3 sheta   Administ   4096 Nov 30 11:26 ..
-rw-r--r--   1 sheta   Administ     23 Nov 30 11:09 HEAD
-rw-r--r--   1 sheta   Administ    363 Nov 30 11:46 config
-rw-r--r--   1 sheta   Administ     73 Nov 29 17:03 description
drwxr-xr-x   2 sheta   Administ   4096 Nov 29 17:03 hooks
-rw-r--r--   1 sheta   Administ     32 Nov 30 11:26 index
drwxr-xr-x   2 sheta   Administ      0 Nov 29 17:03 info
drwxr-xr-x   3 sheta   Administ      0 Nov 29 17:03 logs
drwxr-xr-x  25 sheta   Administ   4096 Nov 30 11:08 objects
-rw-r--r--   1 sheta   Administ     94 Nov 29 17:03 packed-refs
drwxr-xr-x   5 sheta   Administ      0 Nov 30 11:07 refs
```



```
├── .git
│   ├── ...
│   ├── HEAD
│   ├── config
│   ├── description
│   ├── hooks
│   ├── index
│   ├── info
│   ├── logs
│   ├── objects
│   ├── packed-refs
│   └── refs
├── Makefile
├── README.txt
└── src
    └── watchdog.c
```

This file holds a reference to the branch (or commit) you currently have checked out

This is the main Git configuration file. It keeps specific Git options for your project

It will show when you have viewed your repository or the list of all versioned repositories.

Directory contains shell scripts that are invoked after the corresponding Git commands

The Git index is used as a staging area between your working directory and your repository

Contains additional information about the repository.

Keeps records of changes made to refs.

In this directory the data of your Git objects are stored – all the contents of the files you have ever checked in

The file consists of the tips of all branches and tags.

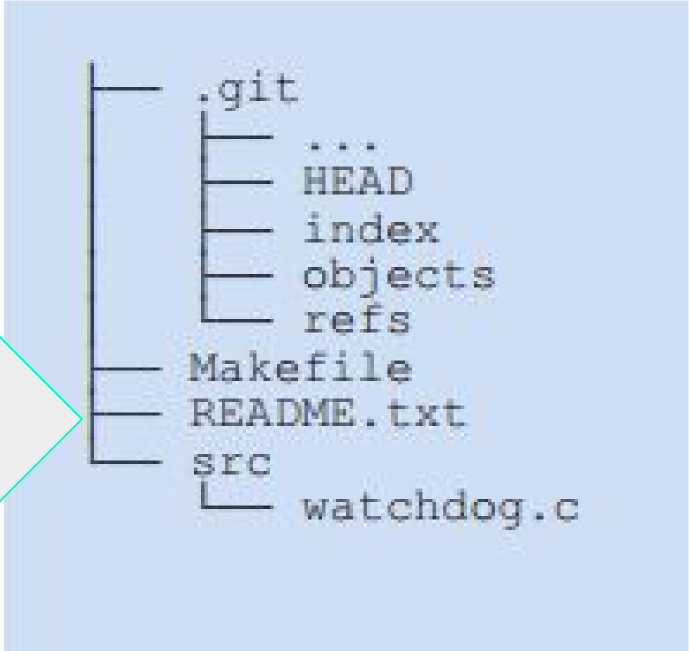
This directory normally contains the tips of branches and tags in three subfolders – heads, remotes and tags.

The Working Tree

- Working tree has all files and folders as found in your HEAD, plus the changes you made since your last commit

- There is only ONE main working tree per repository (and only 1 .git folder as well)

The Working Tree



```
├── .git
│   ├── ...
│   ├── HEAD
│   ├── index
│   ├── objects
│   └── refs
├── Makefile
├── README.txt
└── src
    └── watchdog.c
```

States of files in Working Tree

- **Untracked** – in the repository folder, git does not keep a version of it.
- **Modified** – tracked, modified since last stage or commit.
- **Staged** – a snapshot of the file, ready to be committed. Even if modified, git will still keep the snapshot.
- **Committed** – version of file saved in repository DB

Objects

Every object in GIT composed of those elements –

Type – “blob”, “tree”, “commit”, “tag/branch”.

A "blob" is basically like a file – it is used to store the content of a source file.

A "tree" is basically like a directory - it references a group of other trees (subdirectories) and/or blobs (files).

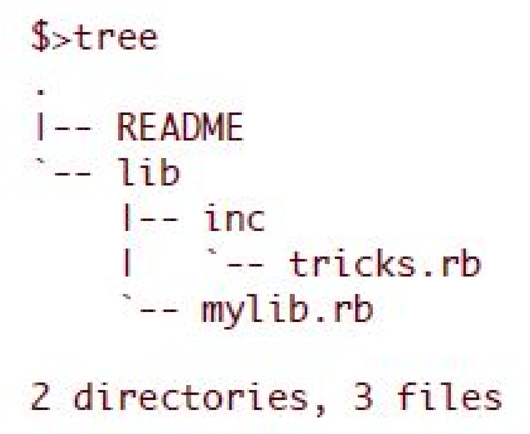
A "commit" points to a single tree, marking it as what the project looked like at a certain point in time. Keeps changed files since the last commit, author of the changes, a reference to the parent commit(s), etc.

A "tag/branch" is a way to mark a specific commit as special in some way. It is usually used to tag certain commits as specific releases or something along those lines.

Objects cont.

Almost all of GIT is built around manipulating this simple structure of four different object types. It is sort of it's own little file system that sits on top of your machine's file system.

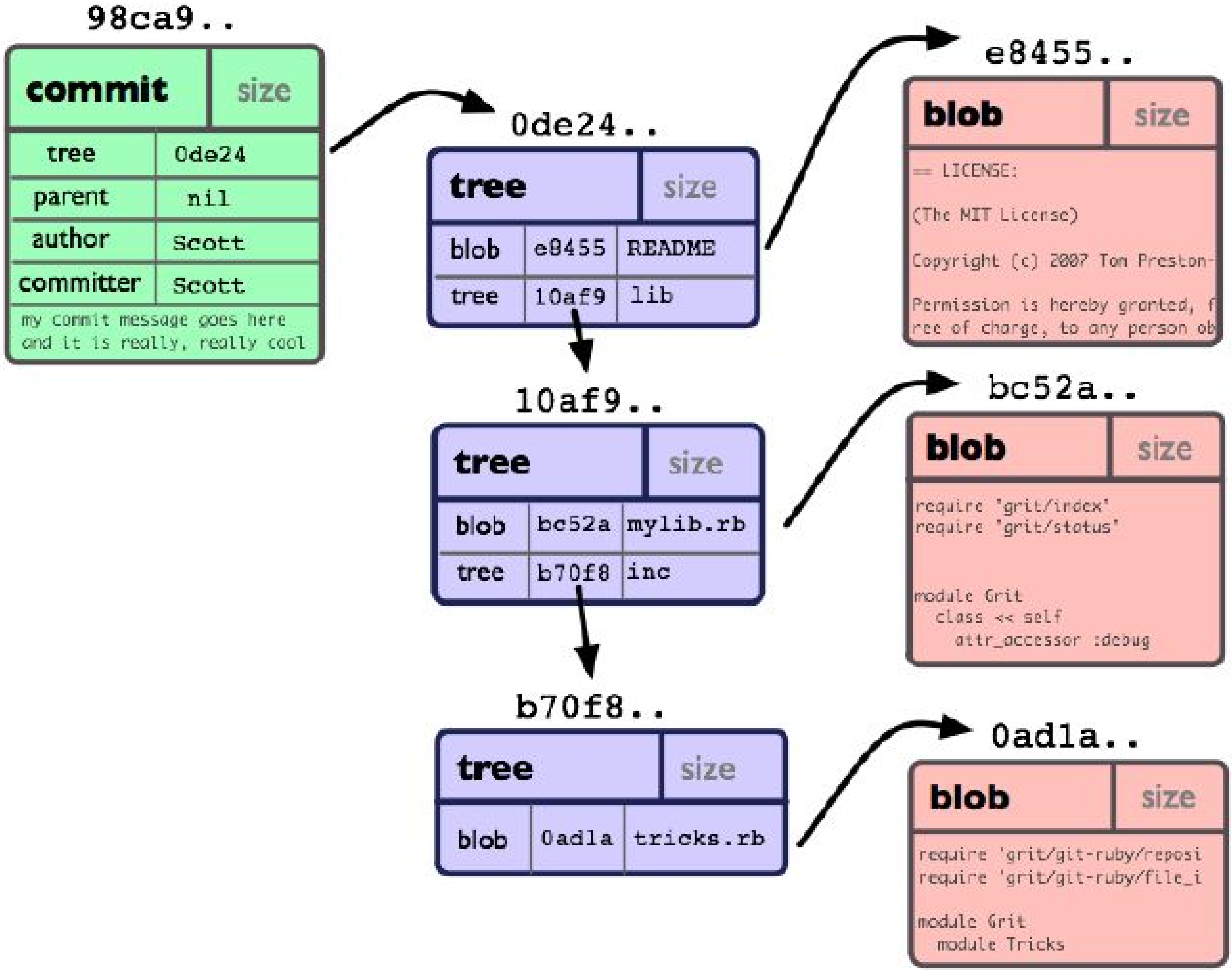
Let's say we have a small project that looks like this:



```
$>tree
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb

2 directories, 3 files
```

If we will commit this project to a GIT repository, it will be represented in GIT like this:

Commit Object



98ca9..

commit	size
tree	0de24
parent	nil
author	Scott
committer	Scott

my commit message goes here
and it is really, really cool

0de24..

tree		size
blob	e8455	README
tree	10af9	lib

e8455..

blob	size

```
== LICENSE:

(The MIT License)

Copyright (c) 2007 Tom Preston-

Permission is hereby granted, f
ree of charge, to any person ob
```

10af9..

tree		size
blob	bc52a	mylib.rb
tree	b70f8	inc

bc52a..

blob	size

```
require 'grit/index'
require 'grit/status'


module Grit
  class << self
    attr_accessor :debug
```

b70f8..

tree		size
blob	0ad1a	tricks.rb

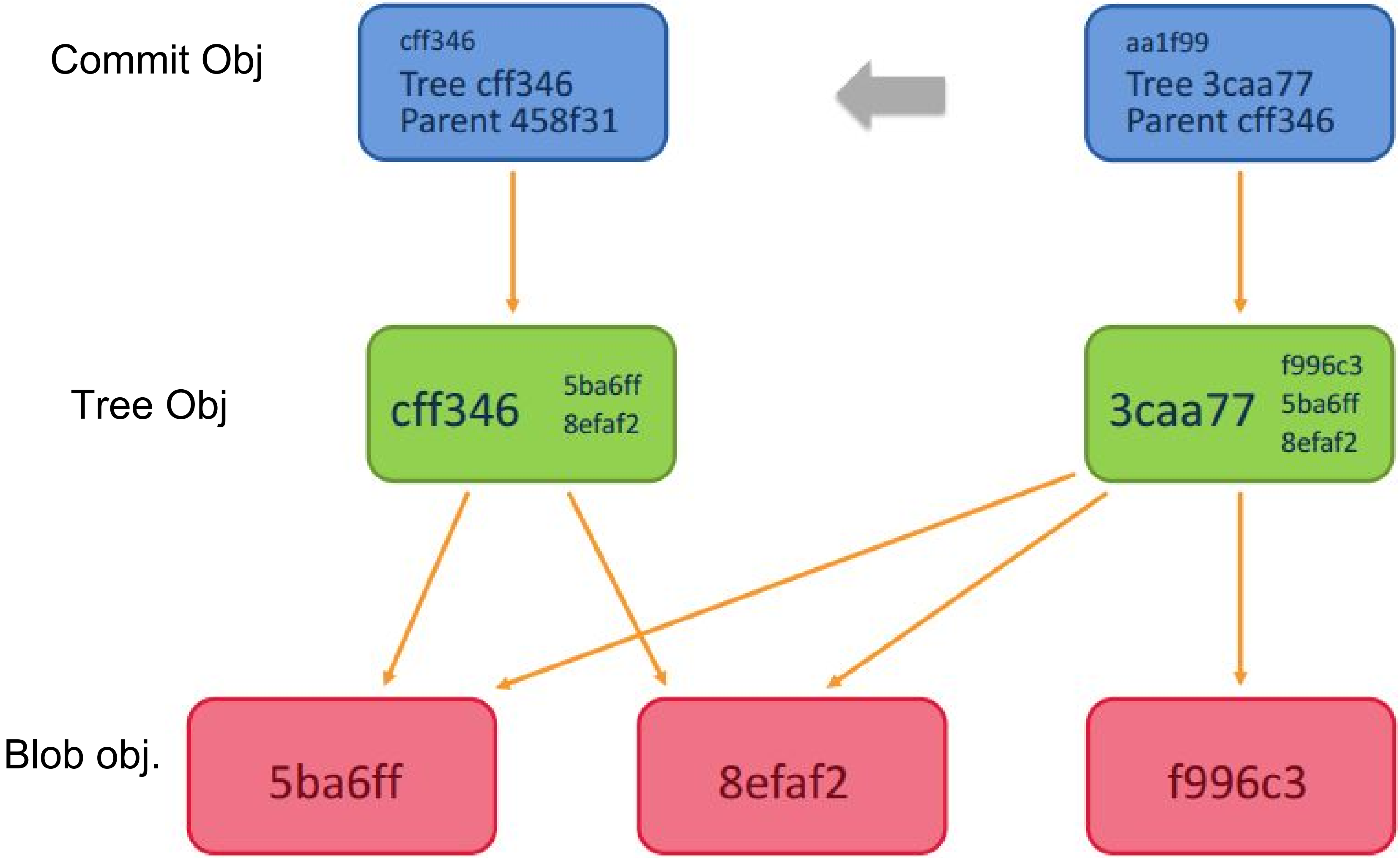
0ad1a..

blob	size

```
require 'grit/git-ruby/reposi
require 'grit/git-ruby/file_i

module Grit
  module Tricks
```

Commit object with its parent

- Links from tree object to common blobs



Commit Obj

cff346
Tree cff346
Parent 458f31

aa1f99
Tree 3caa77
Parent cff346

Tree Obj

cff346
5ba6ff
8efaf2

3caa77
f996c3
5ba6ff
8efaf2

Blob obj.

5ba6ff

8efaf2

f996c3

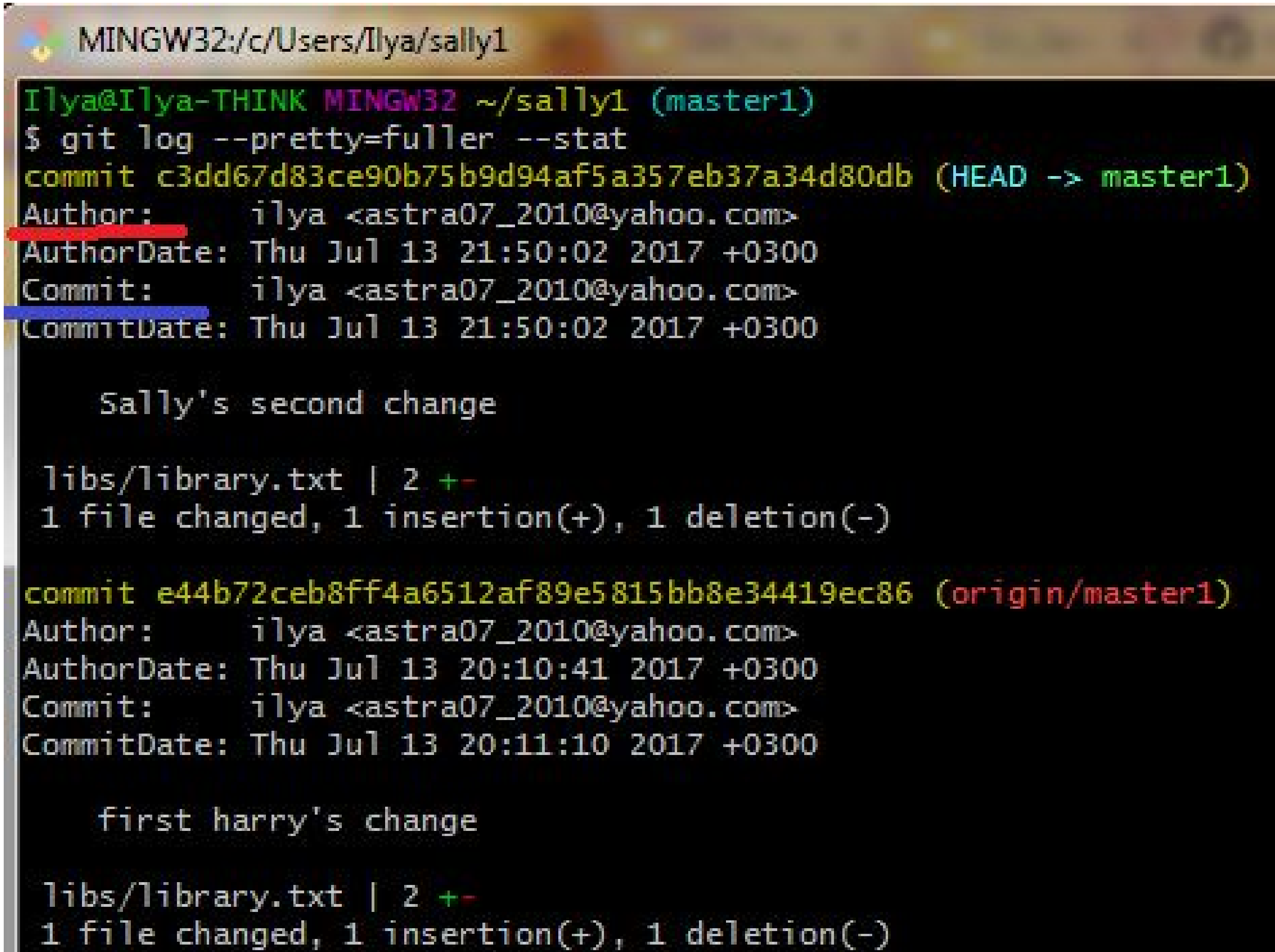
Secure Hash Algorithm – SHA1

- Each object in GIT is represented by a 40-digit string, that looks something like that: 7bf68ebf3d8cff042bd3cb87e7592ddda9caa665. This string is being calculated by taking the SHA1 hash of the contents of the object.

Each commit has Author and Committer

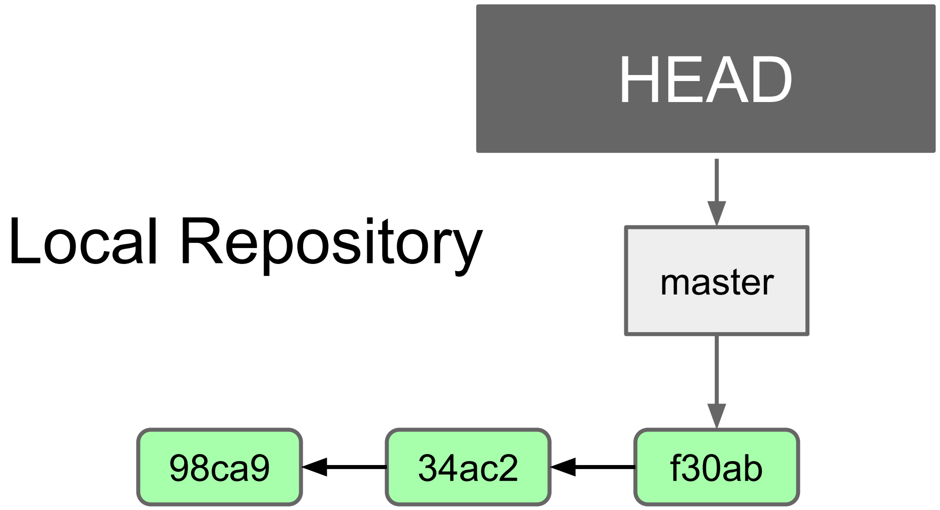
Author is who really wrote the code and committed

Committer, if not the same as Author, took Original commit and reused it in his branch



```
Ilya@Ilya-THINK MINGW32 ~/sally1 (master1)
$ git log --pretty=fuller --stat
commit c3dd67d83ce90b75b9d94af5a357eb37a34d80db (HEAD -> master1)
Author:     ilya <astra07_2010@yahoo.com>
AuthorDate: Thu Jul 13 21:50:02 2017 +0300
Commit:     ilya <astra07_2010@yahoo.com>
CommitDate: Thu Jul 13 21:50:02 2017 +0300

    Sally's second change

 libs/library.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit e44b72ceb8ff4a6512af89e5815bb8e34419ec86 (origin/master1)
Author:     ilya <astra07_2010@yahoo.com>
AuthorDate: Thu Jul 13 20:10:41 2017 +0300
Commit:     ilya <astra07_2010@yahoo.com>
CommitDate: Thu Jul 13 20:11:10 2017 +0300

    first harry's change

 libs/library.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The HEAD



- HEAD is a 'pointer' to the tip of the currently checked out branch
  - In a *detached HEAD* state, HEAD points directly to a commit

- Only one HEAD per repository

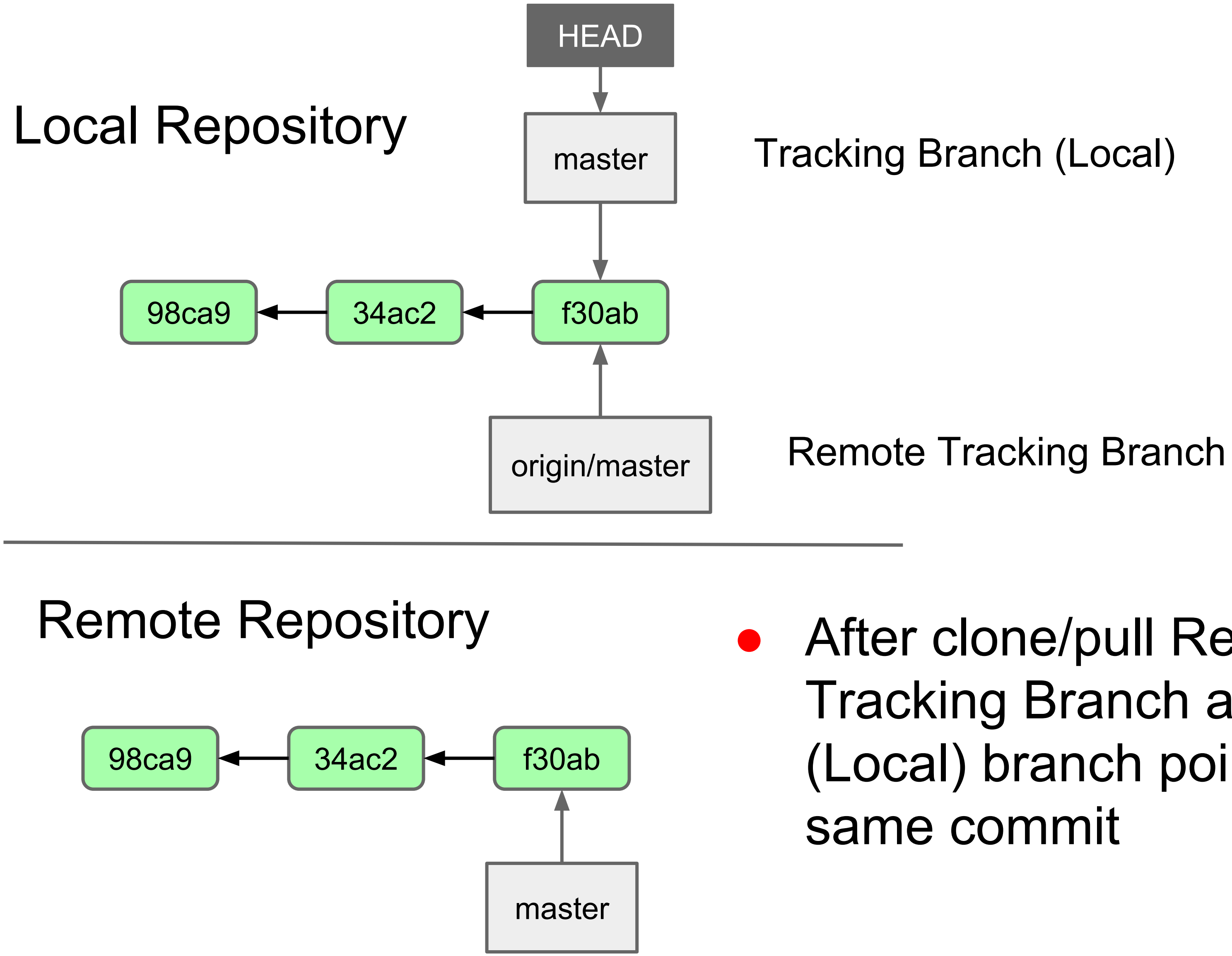
Demo

Lab 3

Teamwork, parallel work:

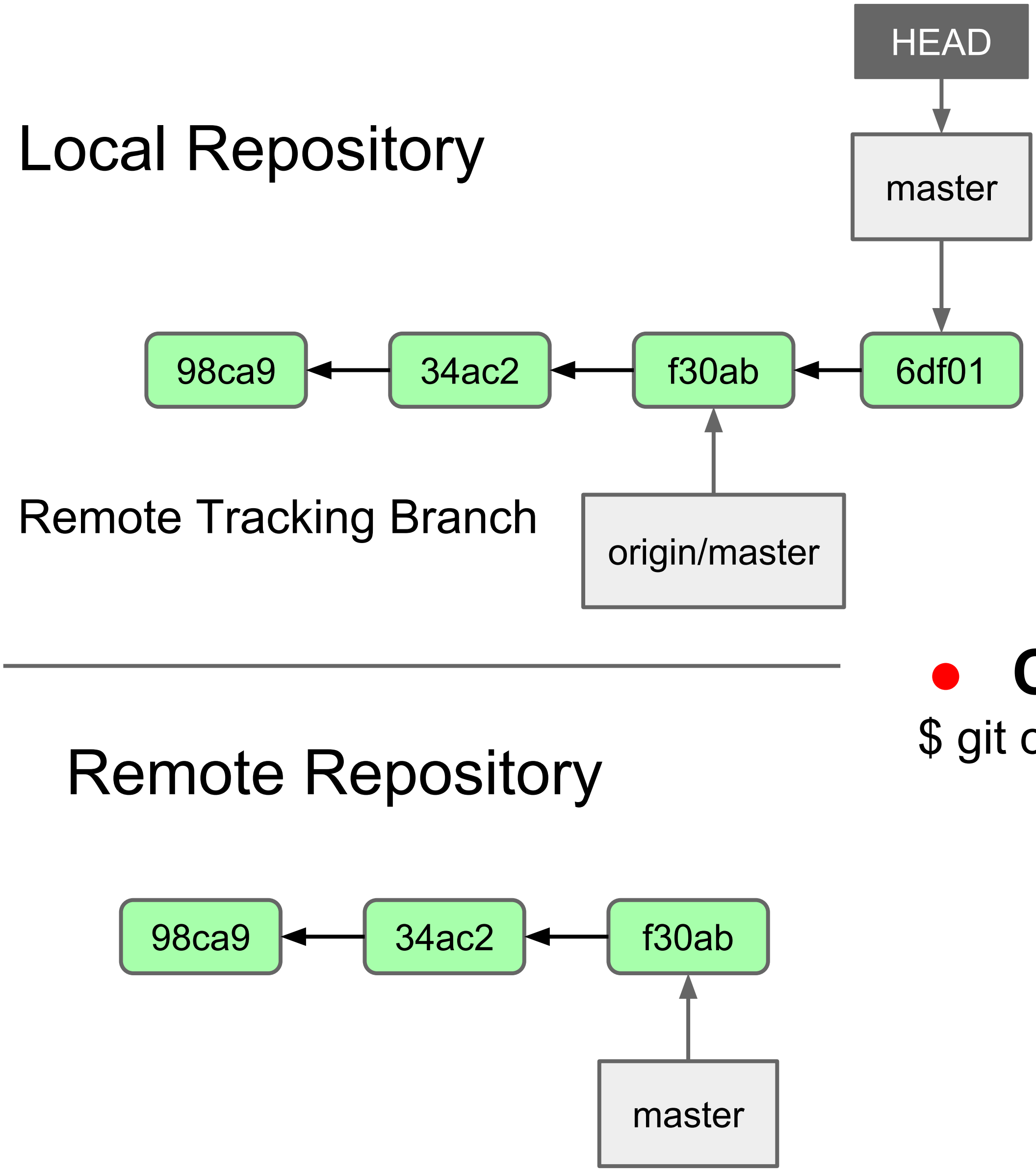
- We will clone second work repo2
- Will commit changes in both repositories
- Push and pull with silent rebase to apply the commit of one repo into another
- Overview results

Rebasing Remote Tracking Branch



- After clone/pull Remote Tracking Branch and Tracking (Local) branch pointing to the same commit

Rebasing Remote Tracking Branch

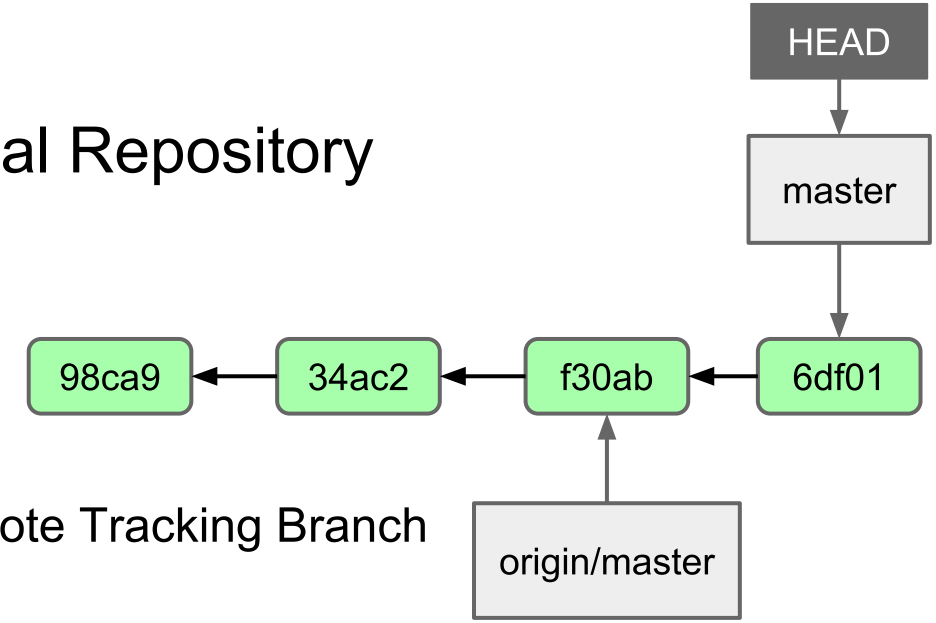


- **Commit on local repository**

```
$ git commit
```

Rebasing Remote Tracking Branch

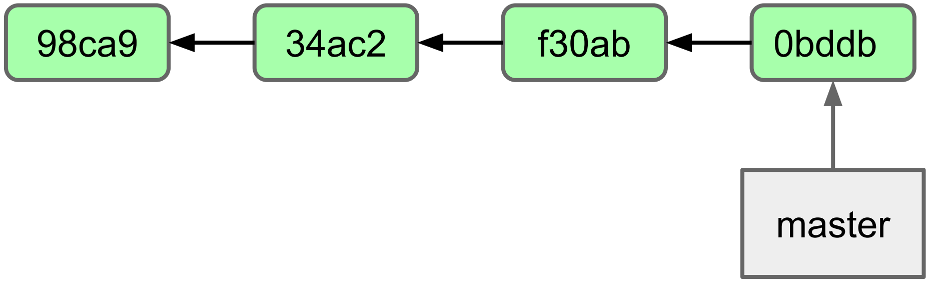
Local Repository



Tracking Branch (Local)

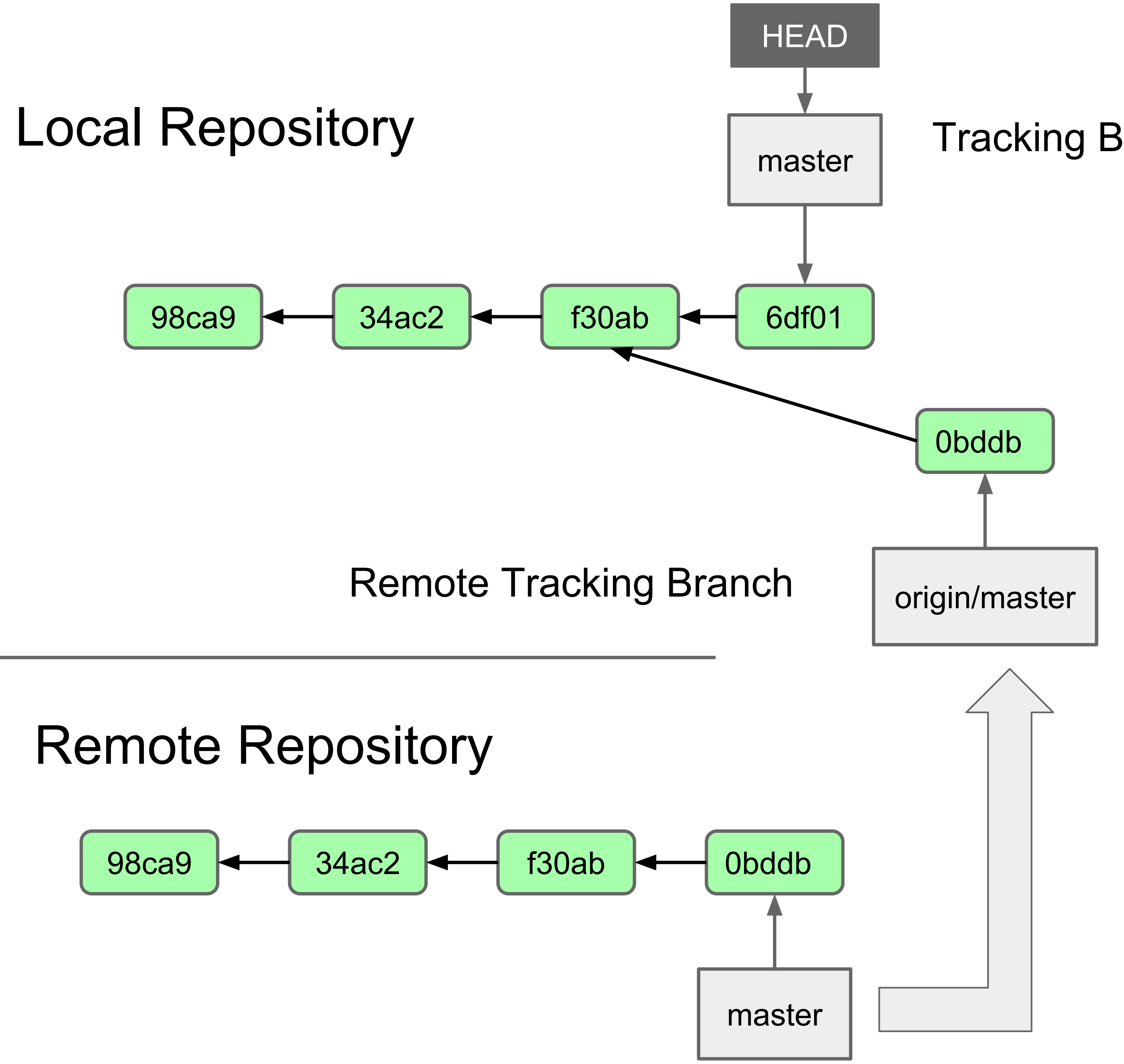
Remote Tracking Branch

Remote Repository



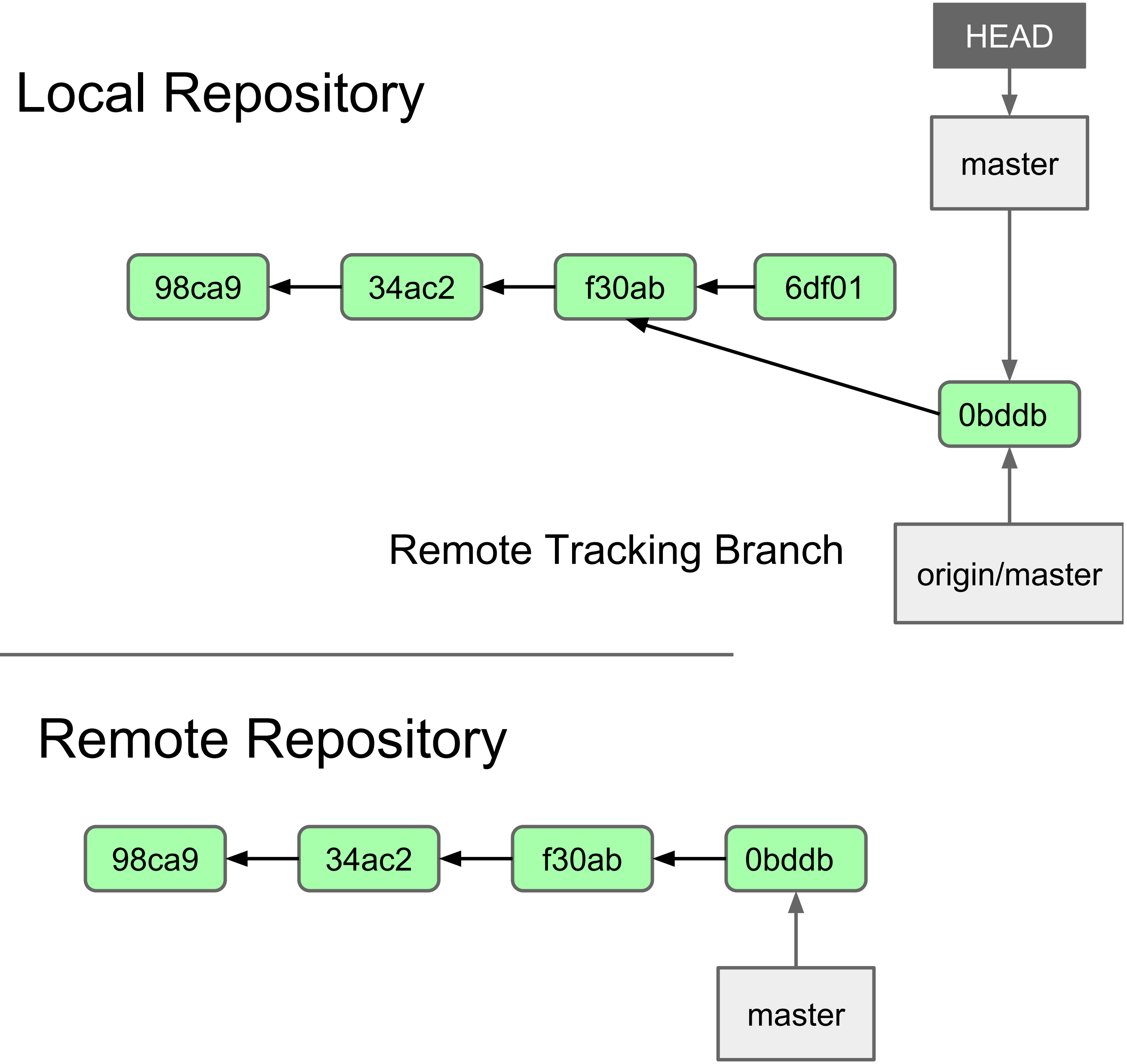
- **Push another Commit to remote repository from another repository/user**

Rebasing Remote Tracking Branch



- **Fetch/Pull the Commit from remote repository to Remote Tracking Branch**

Rebasing Remote Tracking Branch


Local Repository

Tracking Branch (Local)

Remote Tracking Branch

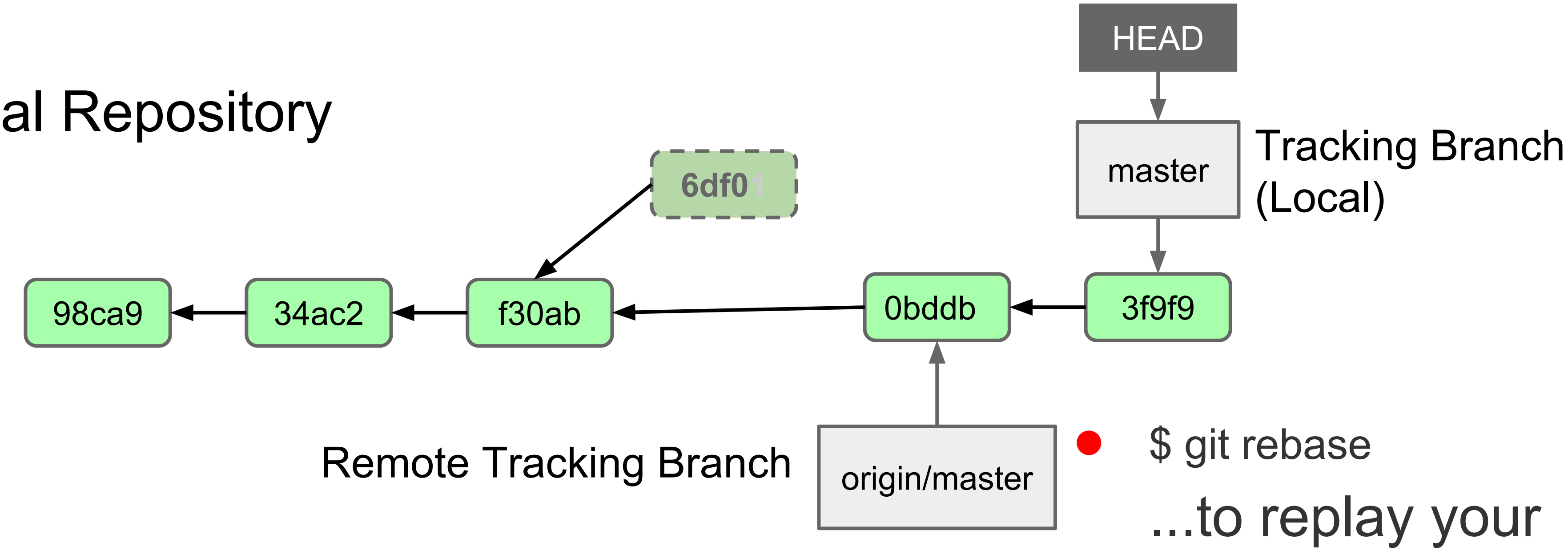
Remote Repository

- $ git rebase

First, rewinding head to replay your work on top of it...

Rebasing Remote Tracking Branch

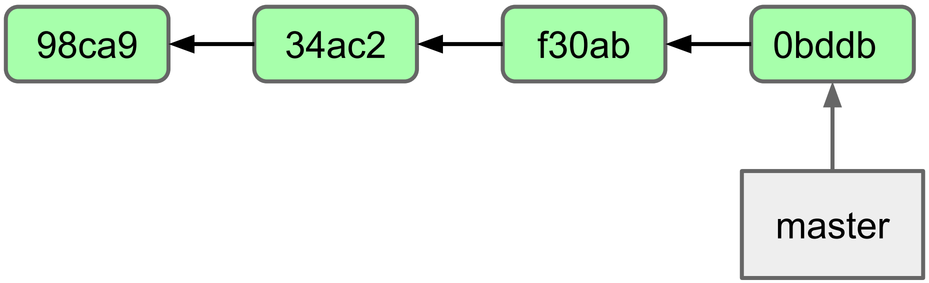
Local Repository



- $ git rebase

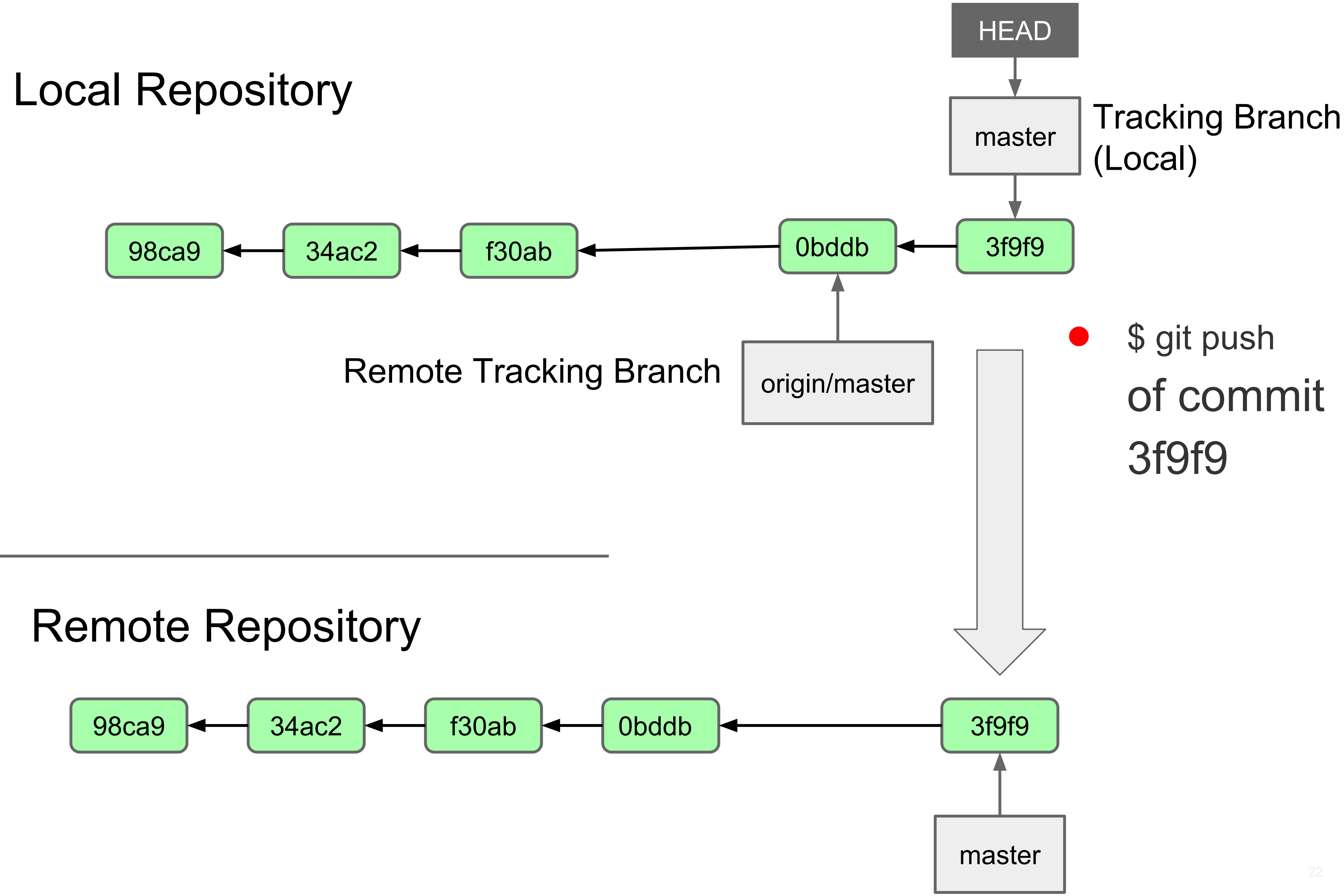
  ...to replay your work on top of it…

Remote Repository



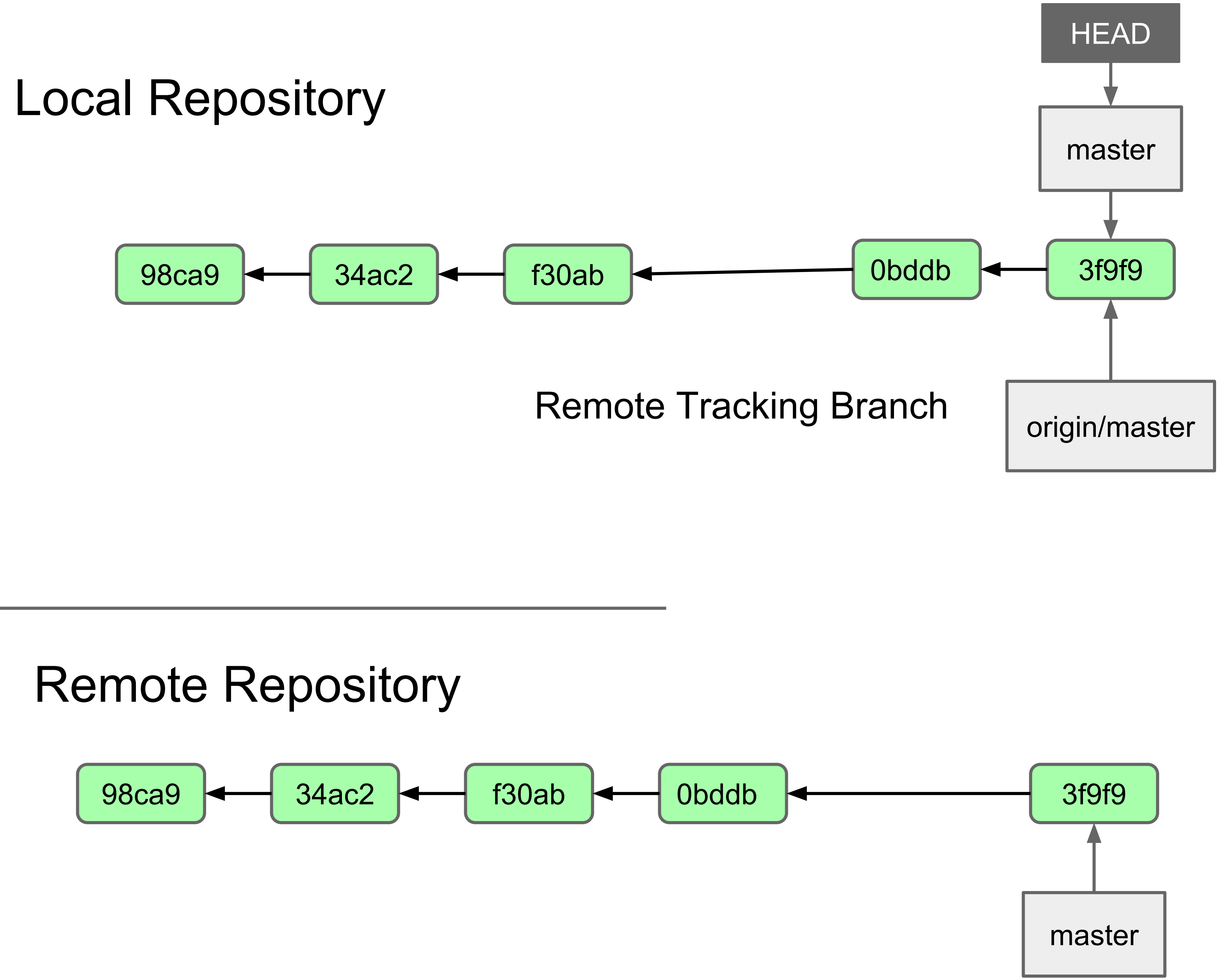
Applying: diff/patch of commit 6dfo into commit 3f9f9

Rebasing Remote Tracking Branch



- $ git push of commit 3f9f9

Rebasing Remote Tracking Branch



- After push Remote Tracking Branch and Tracking (Local) branch points to the same commit

Perils of rebase

- **Do not rebase commits that you have pushed to a public repository.**

If you follow that guideline, you’ll be fine. If you don’t, people will hate you, and you’ll be scorned by friends and family.

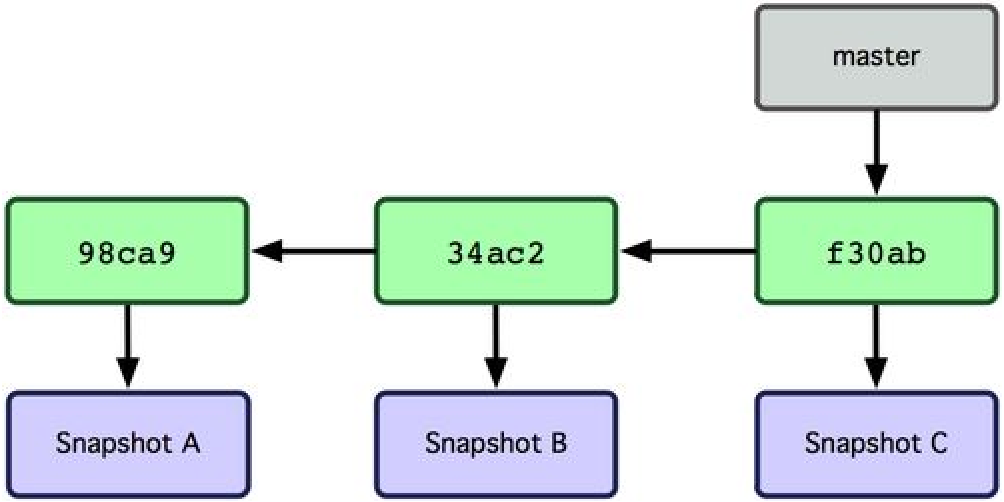
Demo

Lab 4

- Teamwork, parallel work with rebase and conflicts resolution:
- We will do commits in both repos, with change in the same line of the same file
- Pull with rebase, resolve conflicts, save the resolution file
- Add the file to the staging area - means resolve.
- Commit and push

Git branches

- Git branch is simply a movable pointer to a commit

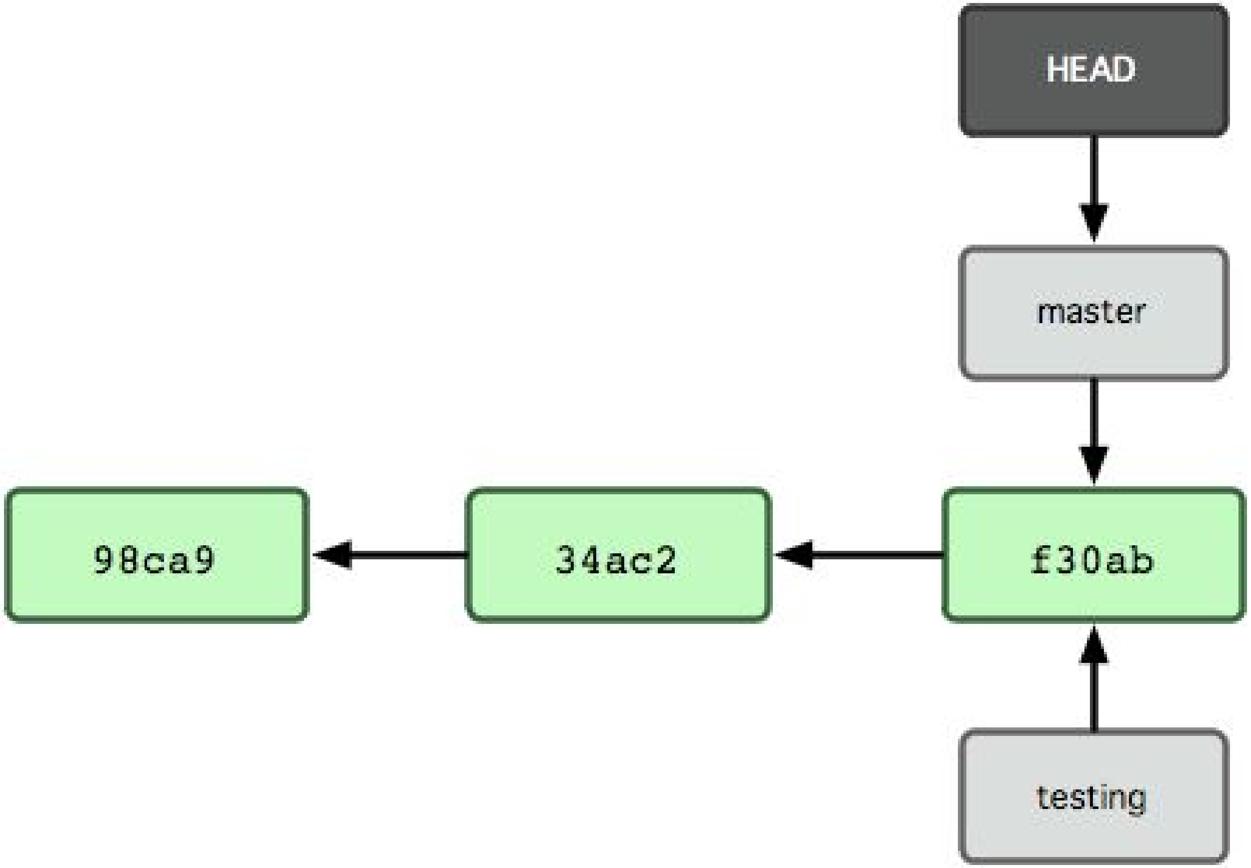


- Pointer moves forward automatically with each commit on a branch

Creating new branch

- New branch creates a new reference

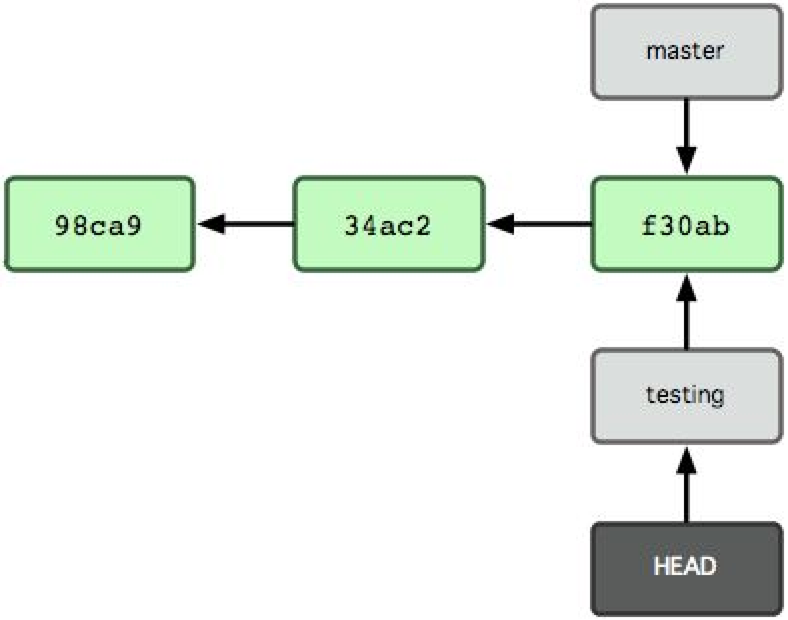
```
> git branch testing
```



Switching to a branch

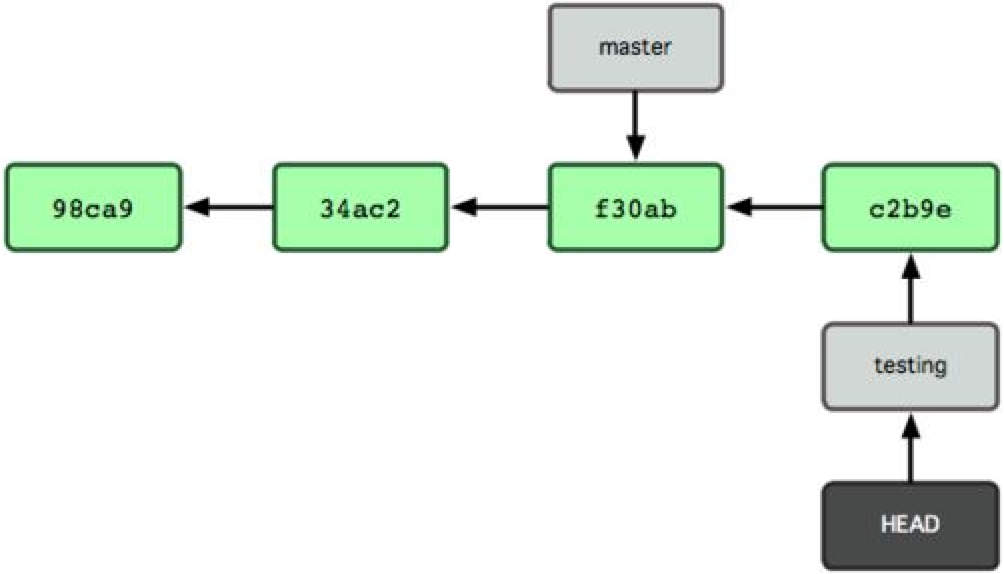
- Git checkout *branch-name* switches to an existing branch

```
> git checkout testing
```

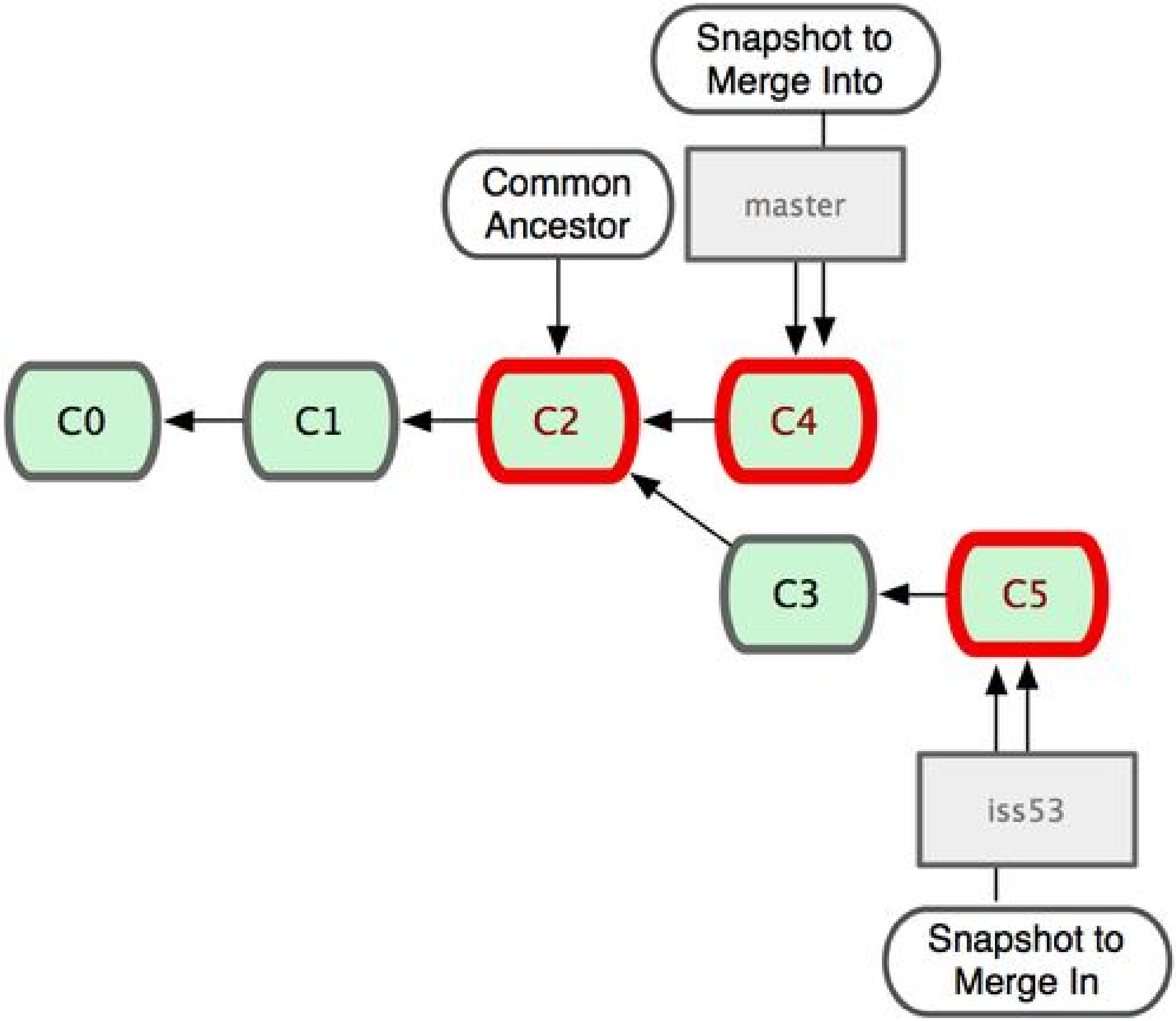


New commit moves current branch

```
> vi file04.txt
> git commit -a -m 'Commit message'
```

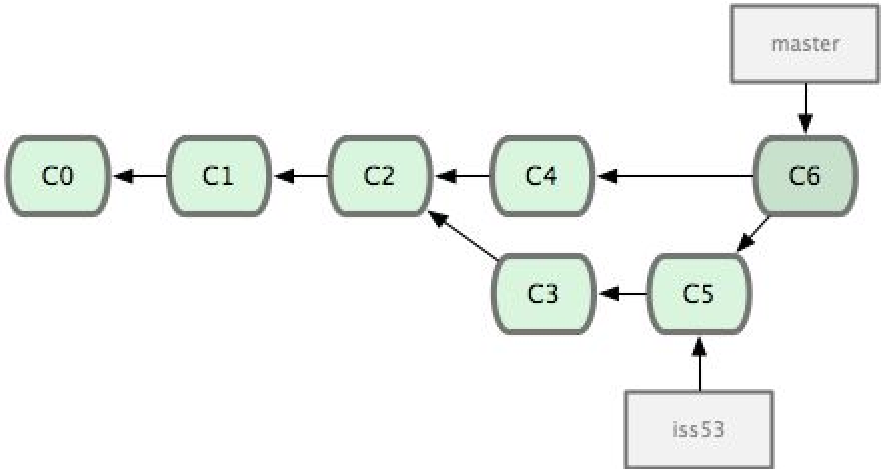


Merging branches



Merging branches (continued)

- As a result of merge Git creates a new commit, which has two parents:



Demo

Lab 5 - 7

- We will create tag
- Create branch bugfix from the tag

Will do bugfix commit in bugfix branch

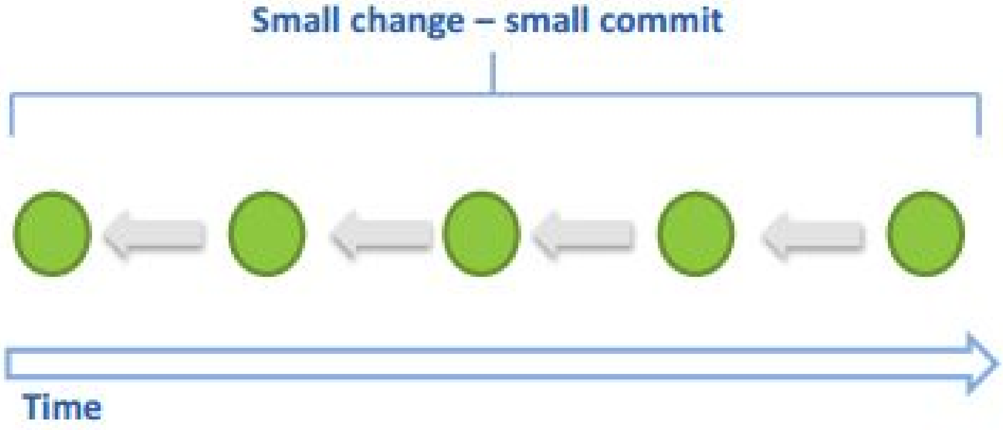
Check out master branch and commit new change

- Merge bugfix branch

Overview results

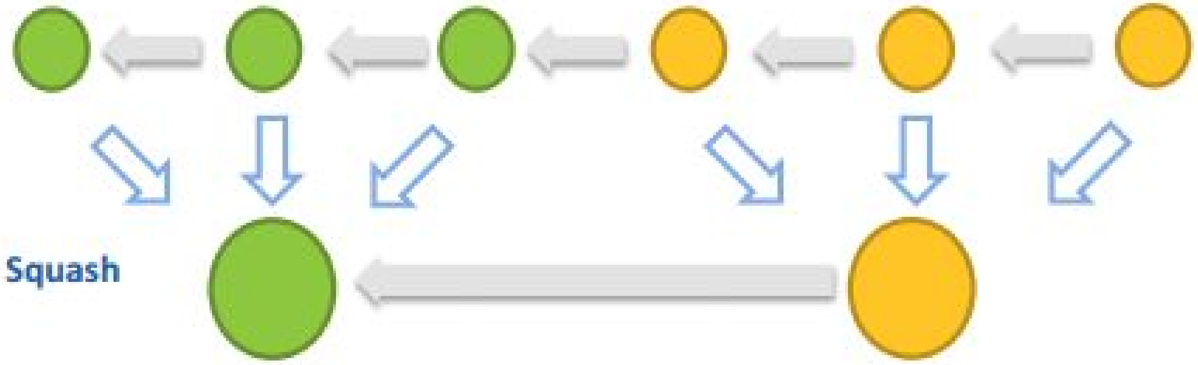
Best practices – commit

Keep changes small wherever possible and commit frequently



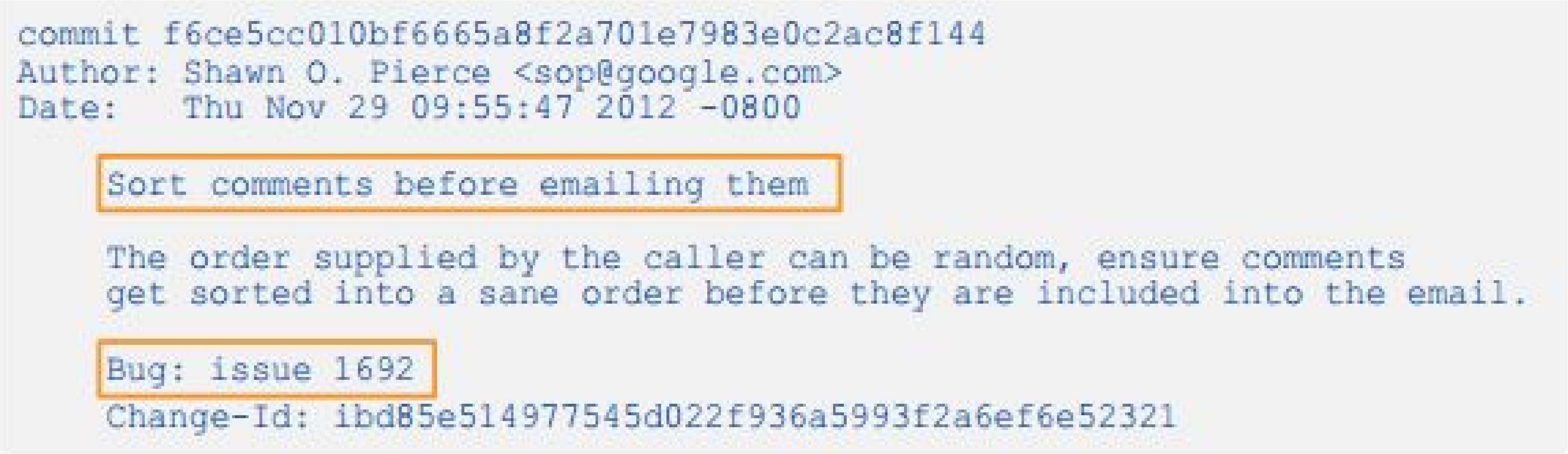
Best practices – squashing

Before pushing, squash related changes together to make for better understanding by others



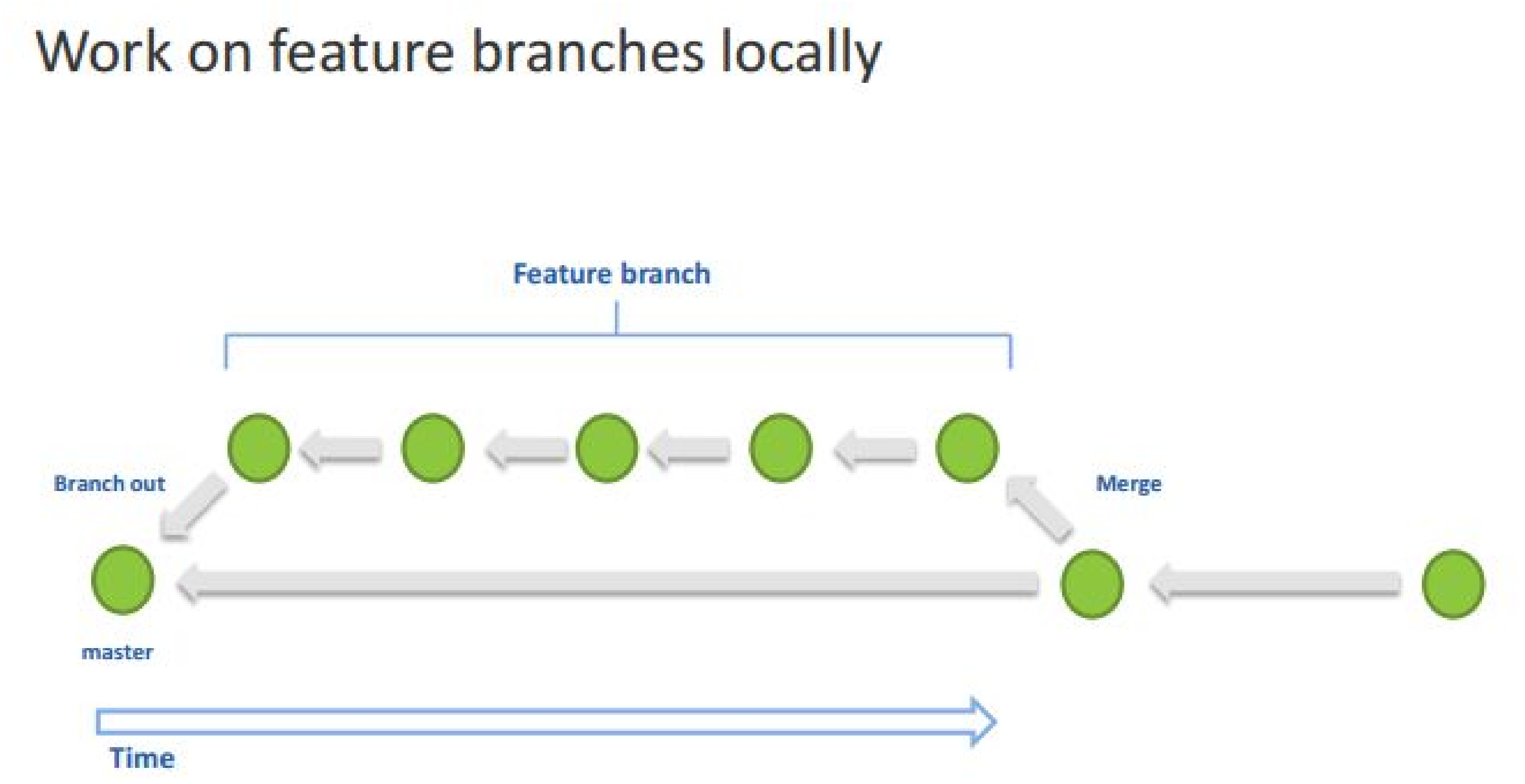
Best practices - concise commit messages

Limit commit message header to **60 characters** and add the meaningful details in the rest of the message



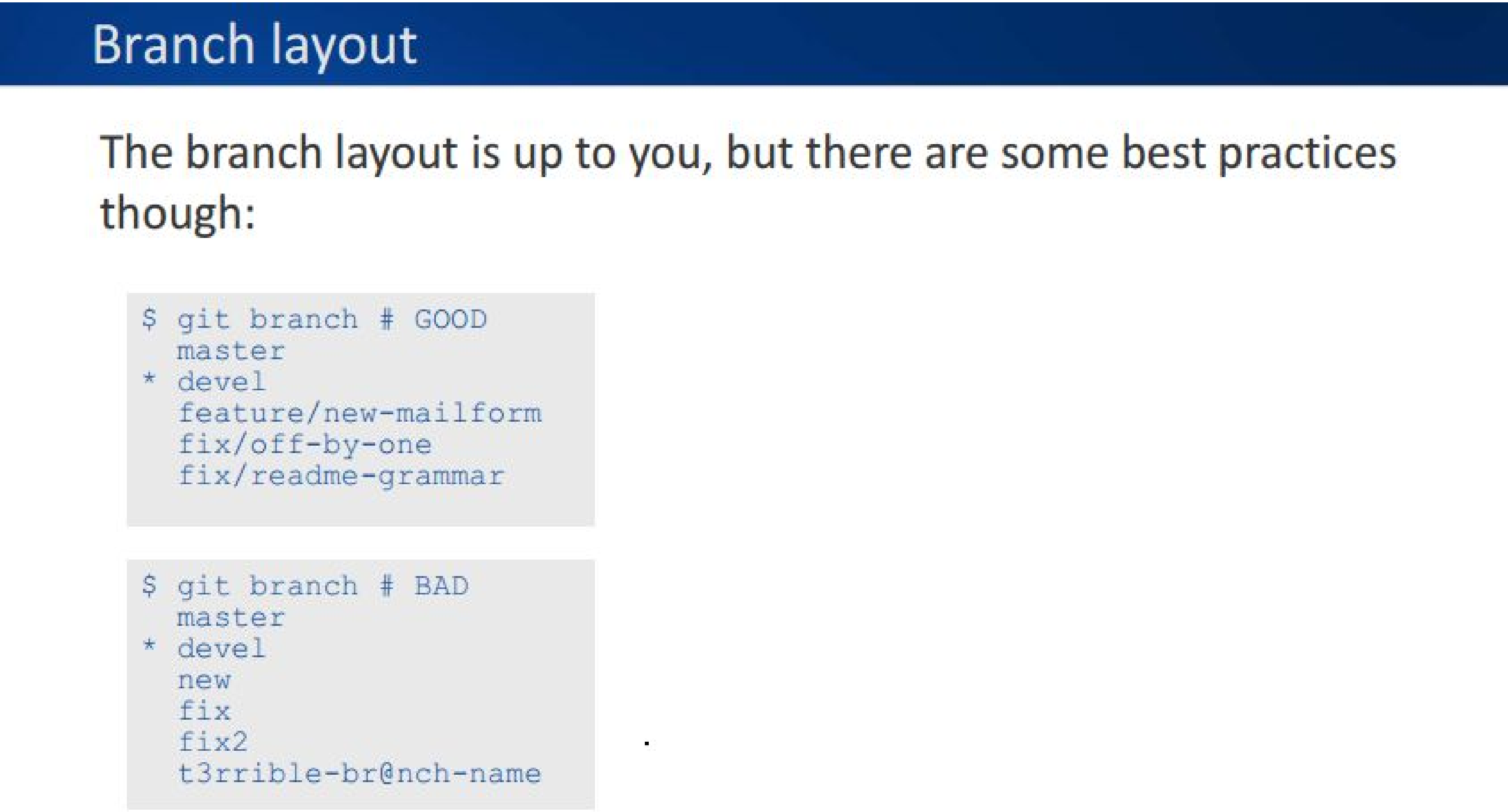
```
commit f6ce5cc010bf6665a8f2a701e7983e0c2ac8f144
Author: Shawn O. Pierce <sop@google.com>
Date:   Thu Nov 29 09:55:47 2012 -0800

    Sort comments before emailing them

    The order supplied by the caller can be random, ensure comments
    get sorted into a sane order before they are included into the email.

    Bug: issue 1692
    Change-Id: ibd85e514977545d022f936a5993f2a6ef6e52321
```

Best practices – local feature branches



Work on feature branches locally

Best practices – Branch Layout



Branch layout

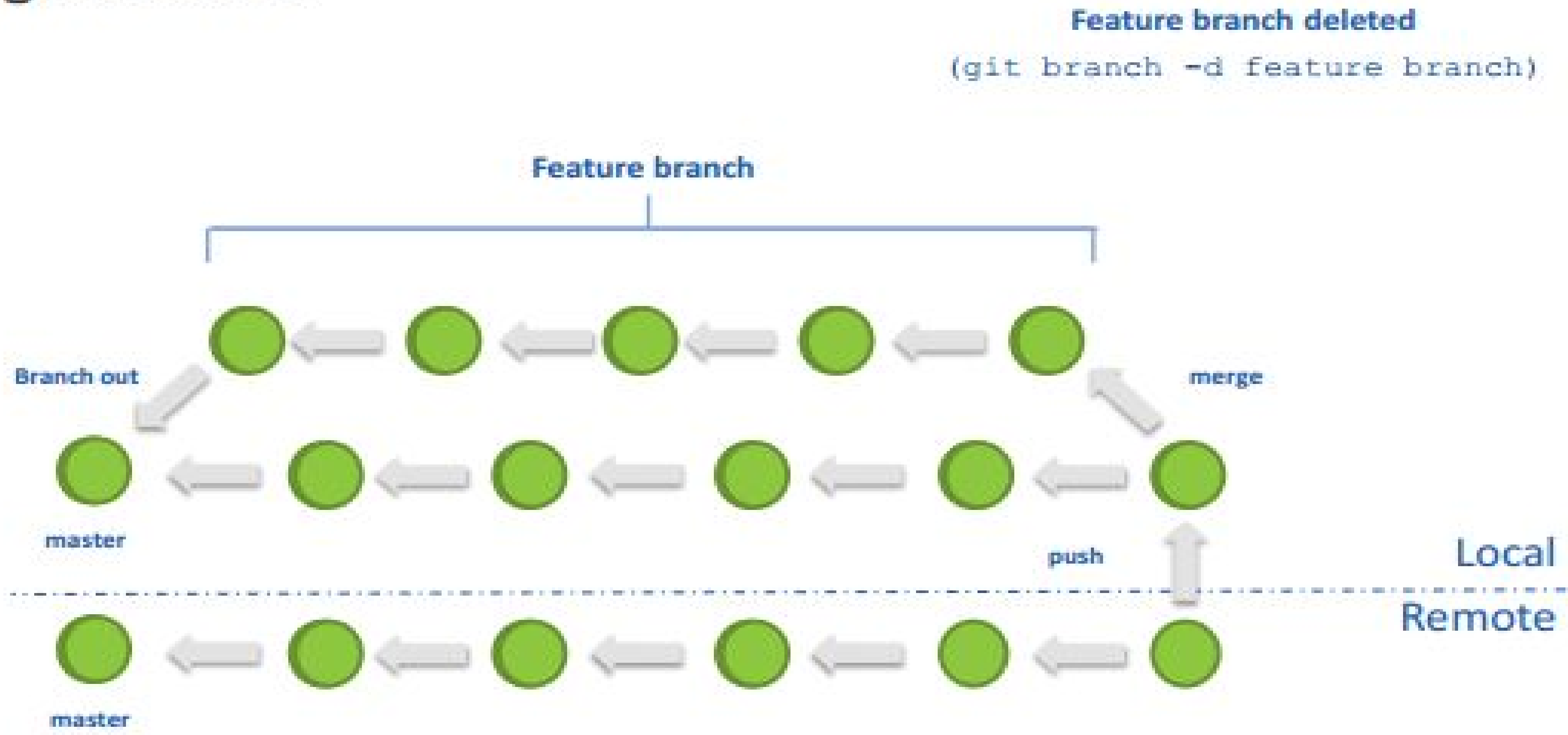
The branch layout is up to you, but there are some best practices though:

```
$ git branch # GOOD
  master
* devel
  feature/new-mailform
  fix/off-by-one
  fix/readme-grammar
```

```
$ git branch # BAD
  master
* devel
  new
  fix
  fix2
  t3rrible-br@nch-name
```

Best practices – clean up local branches

Clean up local branches once the code gets pushed to target branch



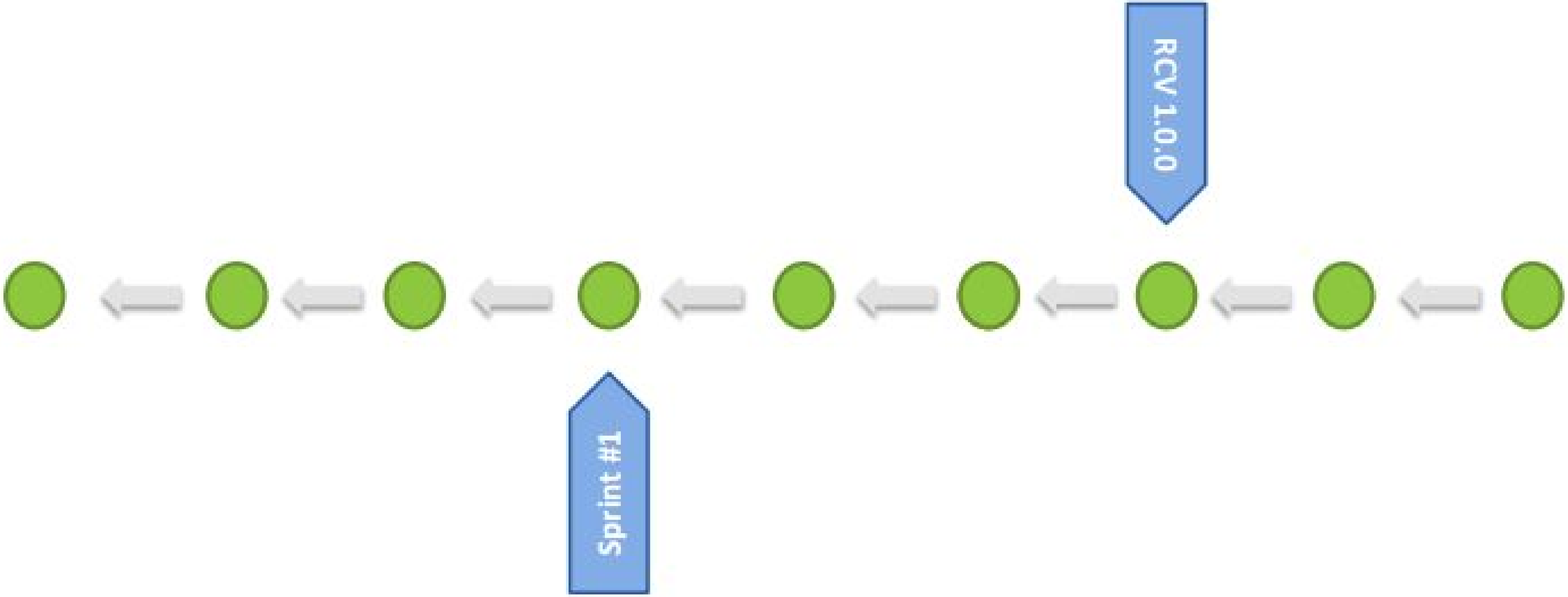
Feature branch deleted

(git branch -d feature branch)

"$ git branch --merged" will show all branch(es) which can be safely deleted

Best practices – tag milestones

Tag important milestones (for history and for accessibility)



Lab 8 - 10

- We will create branch bugfix2, from Release_01

  Cherry-pick 1 commit from master branch
- Undo modified file, undo staged file

  Undo latest local commit, revert pushed commit
- Stash meanwhile work aside, make commit, return work from stash

Lab 11 - 13

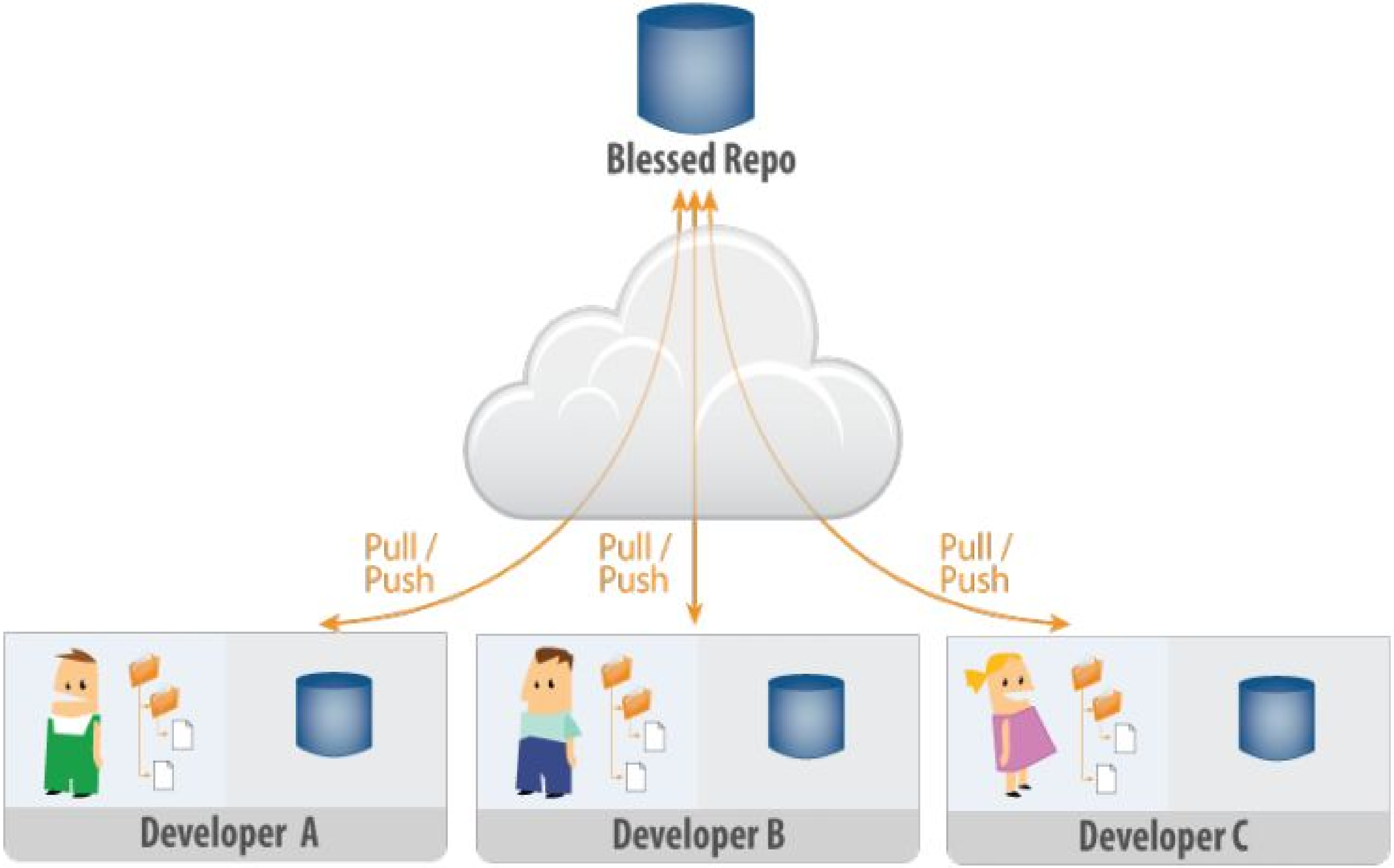
- Format patch in 1 repository, and apply it in another repository
- We will create 2 local commits and squash them to 1 commit by interactive rebase, and push only 1 commit to remote repository
- Create commit in 1 repository and pull it from another repository, without pushing to origin repository

Question?

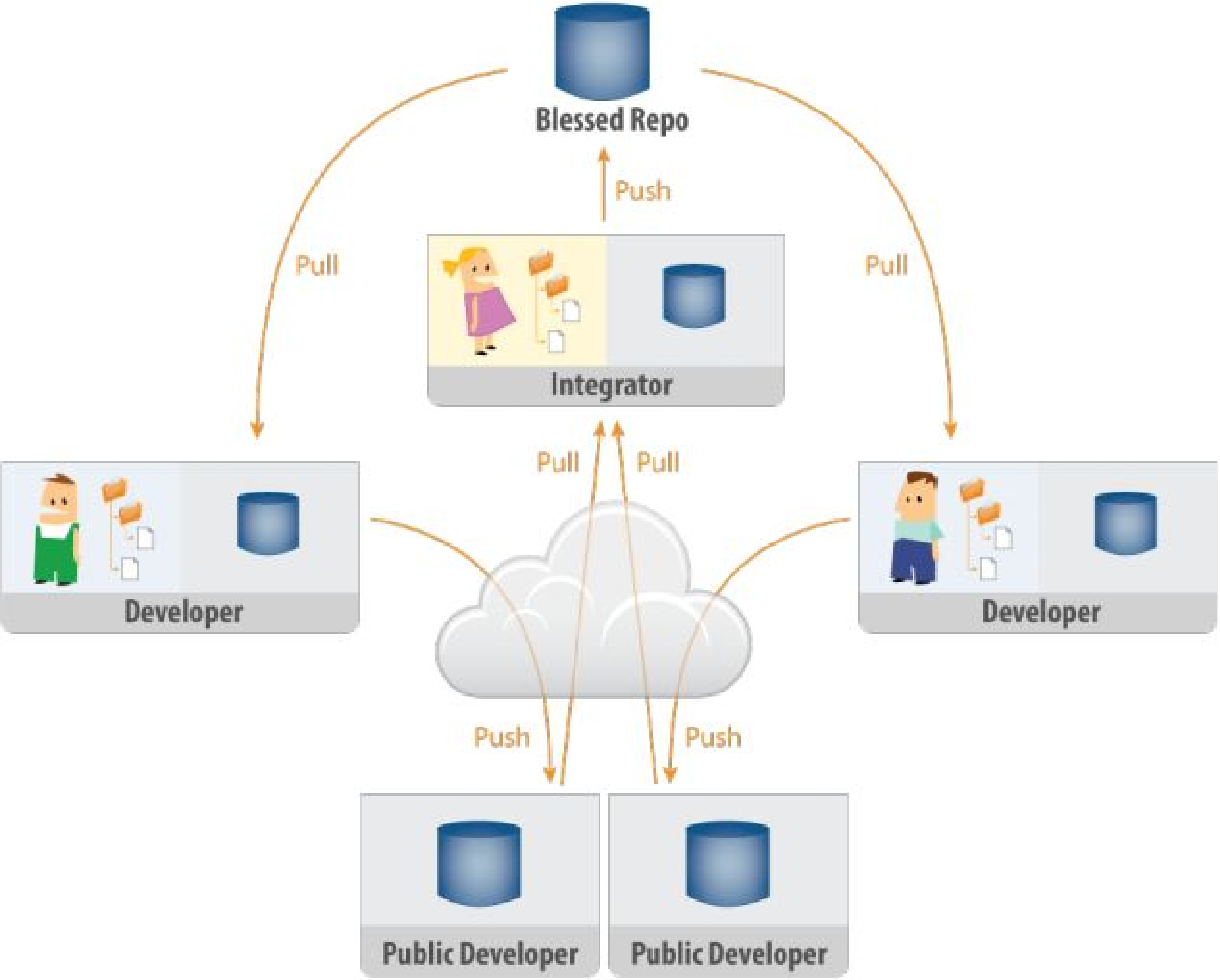
Thanks!

Backup slides

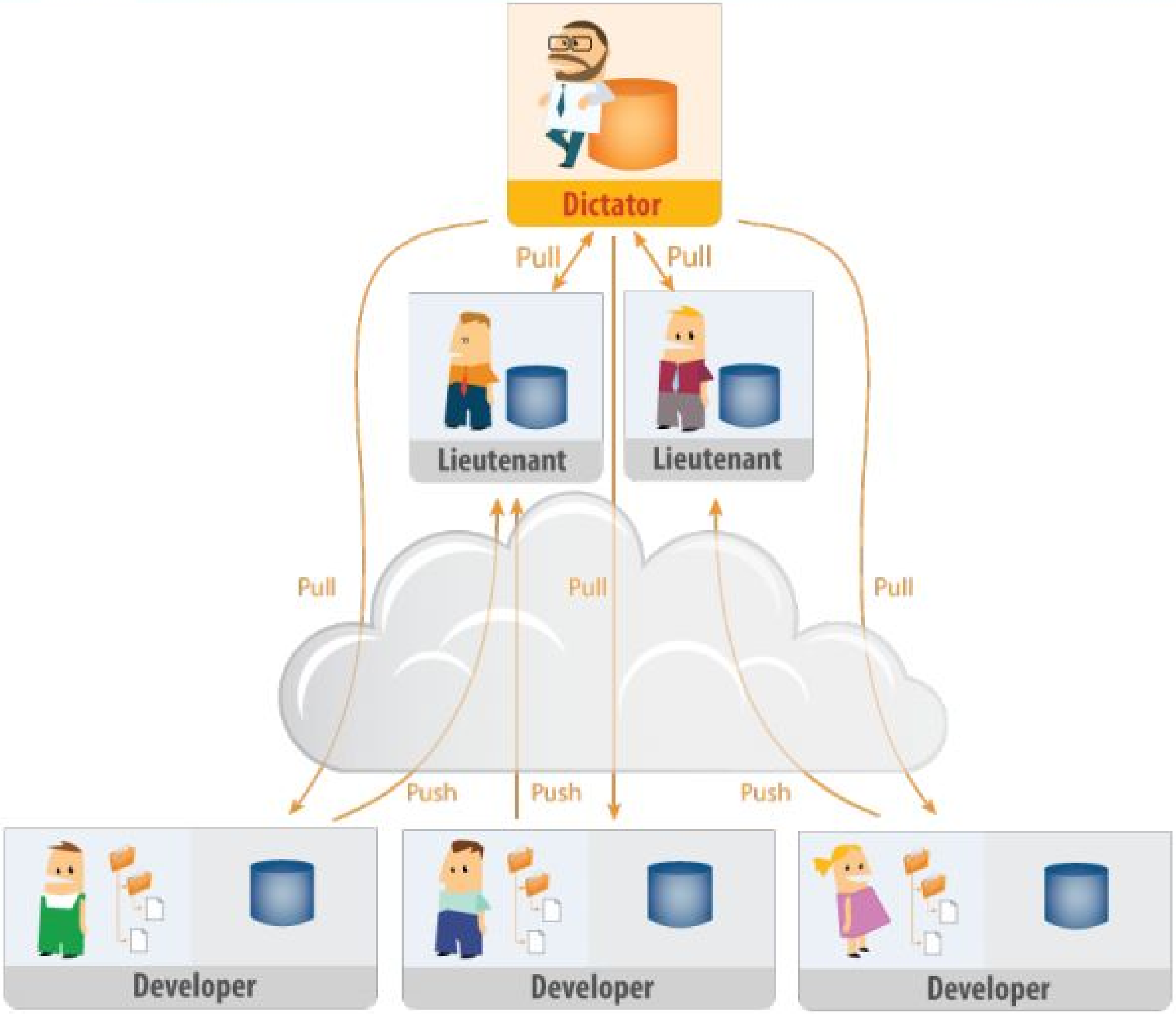
Centralized Workflow



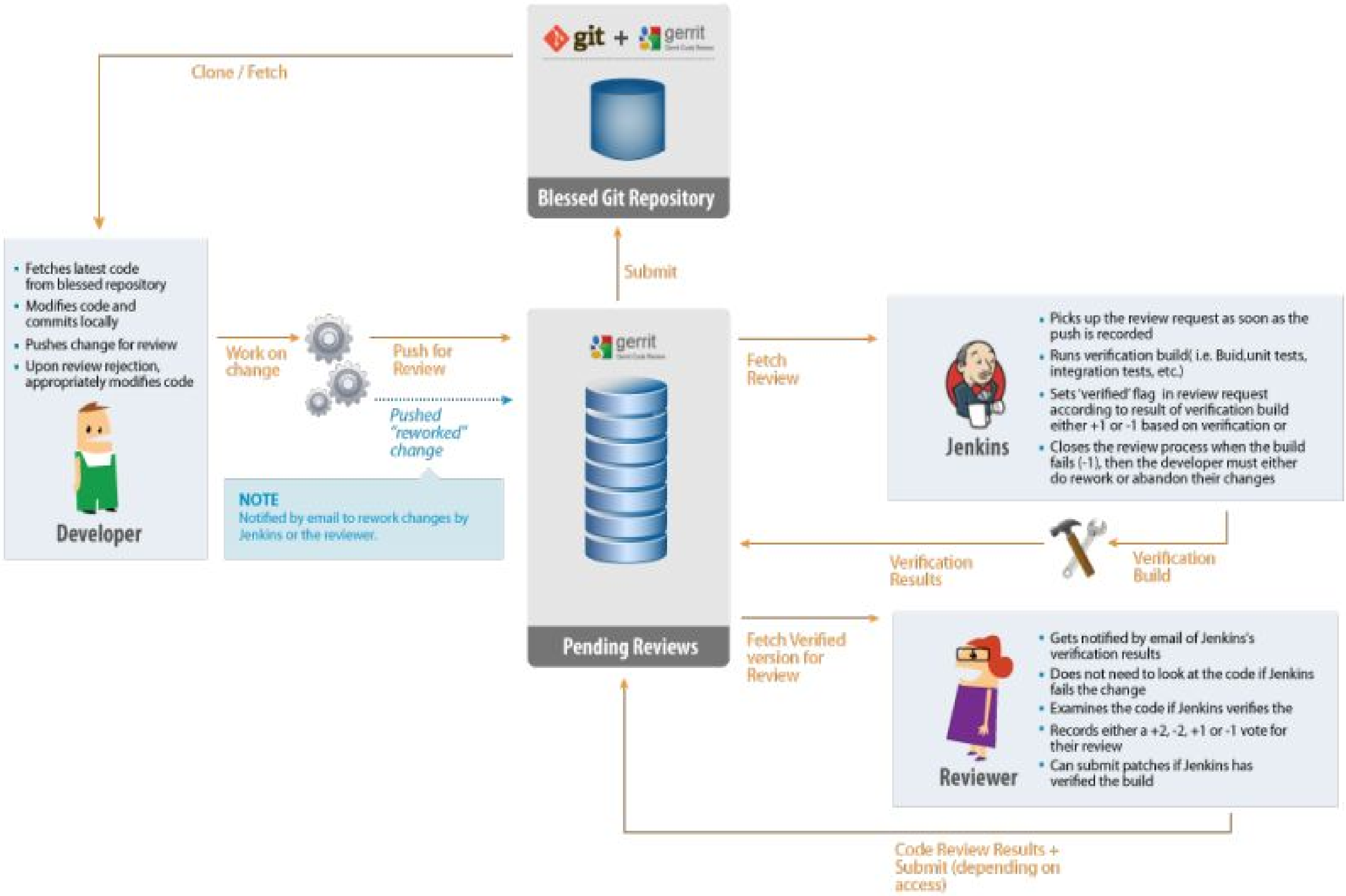
Integrators Workflow



Dictator / Lieutenants Workflow



Gerrit Code Review Workflow



git + gerrit

Blessed Git Repository

Clone / Fetch

Submit

- Fetches latest code from blessed repository
- Modifies code and commits locally
- Pushes change for review
- Upon review rejection, appropriately modifies code

Developer

Work on change

Push for Review

Pushed "reworked" change

NOTE
Notified by email to rework changes by Jenkins or the reviewer.

gerrit

Pending Reviews

Fetch Review

- Picks up the review request as soon as the push is recorded
- Runs verification build(i.e. Buid,unit tests, integration tests, etc.)
- Sets 'verified' flag in review request according to result of verification build either +1 or -1 based on verification or
- Closes the review process when the build fails (-1), then the developer must either do rework or abandon their changes

Jenkins

Verification Results

Verification Build

Fetch Verified version for Review

- Gets notified by email of Jenkins's verification results
- Does not need to look at the code if Jenkins fails the change
- Examines the code if Jenkins verifies the
- Records either a +2, -2, +1 or -1 vote for their review
- Can submit patches if Jenkins has verified the build

Reviewer

Code Review Results + Submit (depending on access)

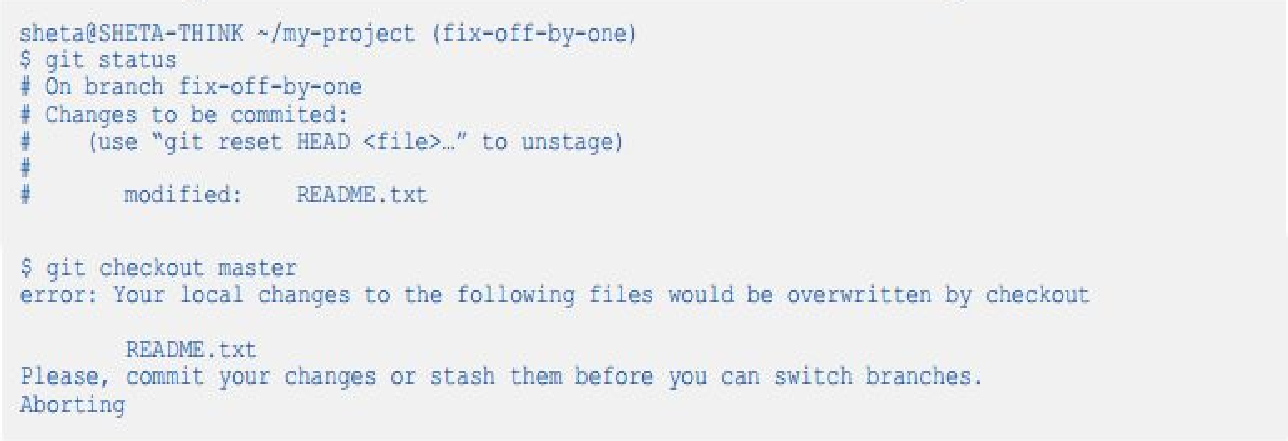
Detached HEAD

If you checkout any commit SHA1, tag, or remote-tracking branch then you will end up having a “detached HEAD“:

```
$ git checkout 494e2cb73ed6424b27f9766bf8a2cb29770a1e7e
Note: checking out '494e2cb73ed6424b27f9766bf8a2cb29770a1e7e'.

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.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 494e2cb... Added README file
```

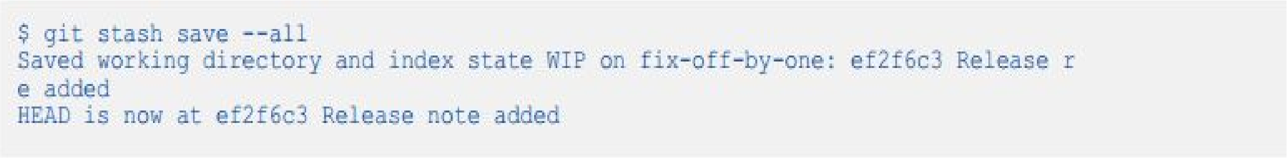
Git stash

You may be in a state where you have some changes that are not ready for committing, but you need to change branches in order to work on something else.



```
sheta@SHETA-THINK ~/my-project (fix-off-by-one)
$ git status
# On branch fix-off-by-one
# Changes to be commited:
#    (use "git reset HEAD <file>…" to unstage)
#
#       modified:    README.txt


$ git checkout master
error: Your local changes to the following files would be overwritten by checkout

        README.txt
Please, commit your changes or stash them before you can switch branches.
Aborting
```

`git stash` takes current state of your working directory (what is staged, modified, etc.) and saves it as a stack of unfinished changes in **`refs/stash`**.

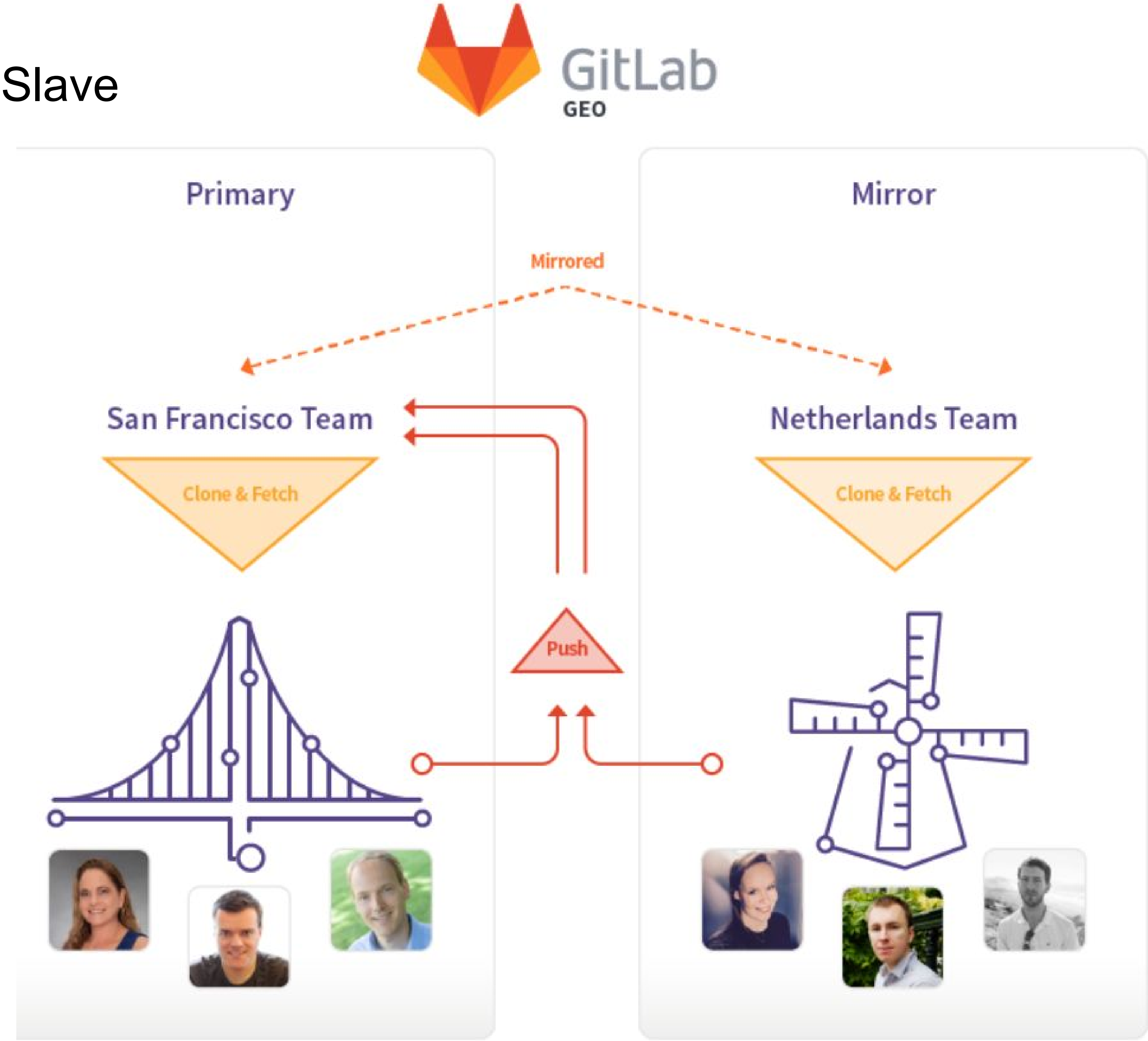


```
$ git stash save --all
Saved working directory and index state WIP on fix-off-by-one: ef2f6c3 Release r
e added
HEAD is now at ef2f6c3 Release note added
```

Later you can switch back to the previous branch and apply your saved changes to your working tree to have it exactly the way you had it prior to stashing your changes. You should

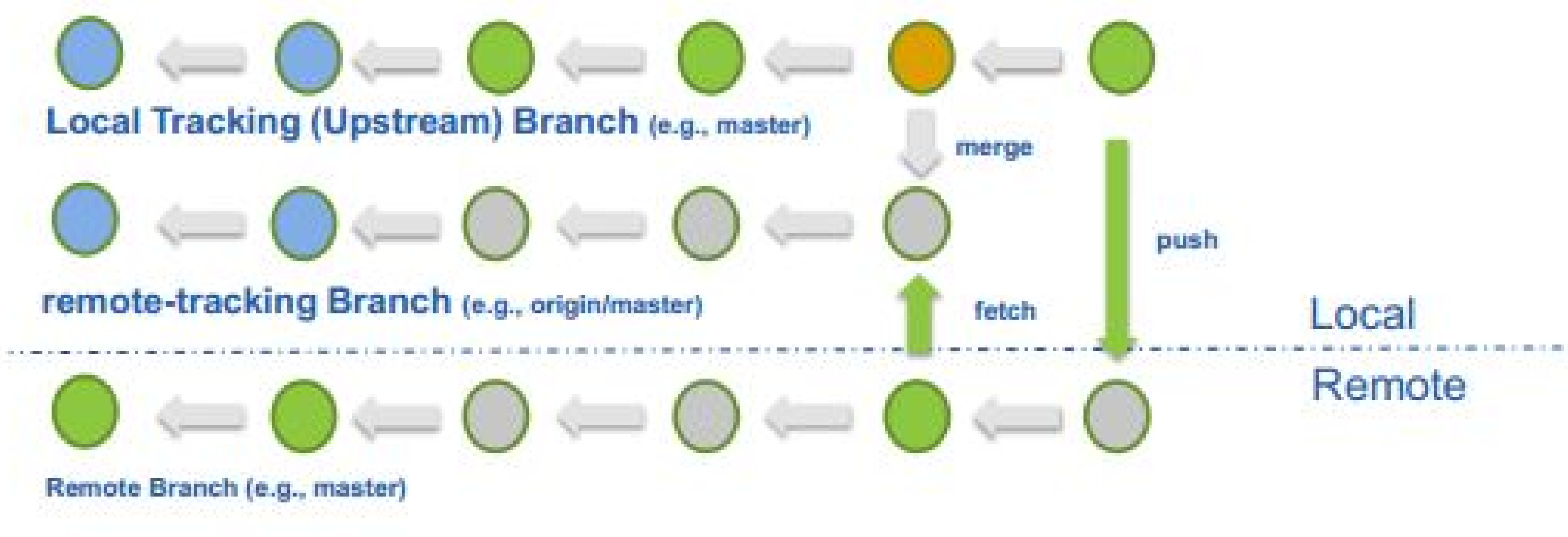
Git Master->Slave

Mirroring



What are tracking and remote-tracking branches?

- The combination of these branches defines a relationship between a local branch and one in the remote repository.
- When a repository is cloned, Git automatically creates **remote-tracking branches** (e.g., origin/master) for the remote branches and a **tracking branch** (e.g., master) to allow for local changes in relationship to the remote branch


Local Tracking (Upstream) Branch (e.g., master)
merge
push
remote-tracking Branch (e.g., origin/master)
fetch
Local
Remote
Remote Branch (e.g., master)