

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

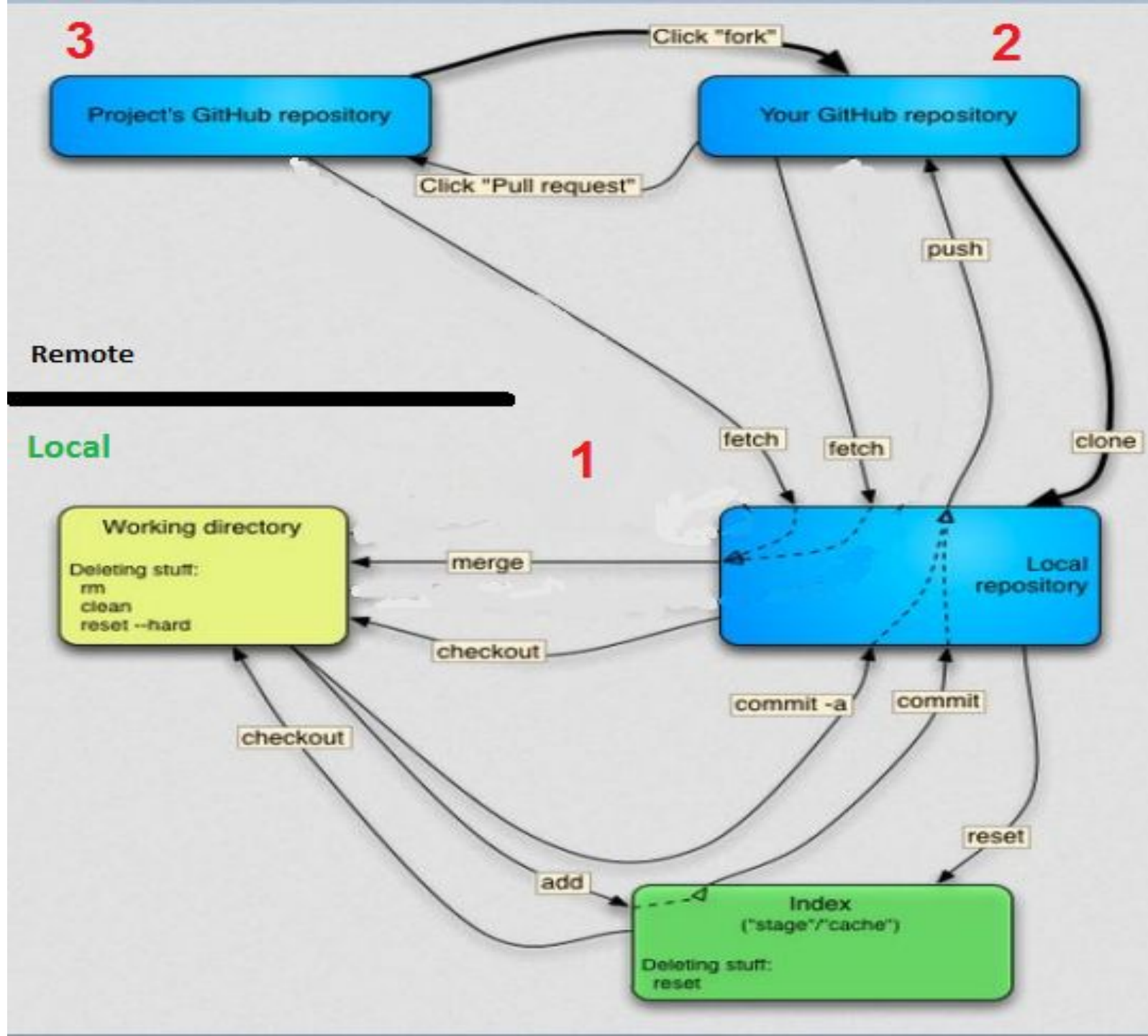
2. Practical part, lab work

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

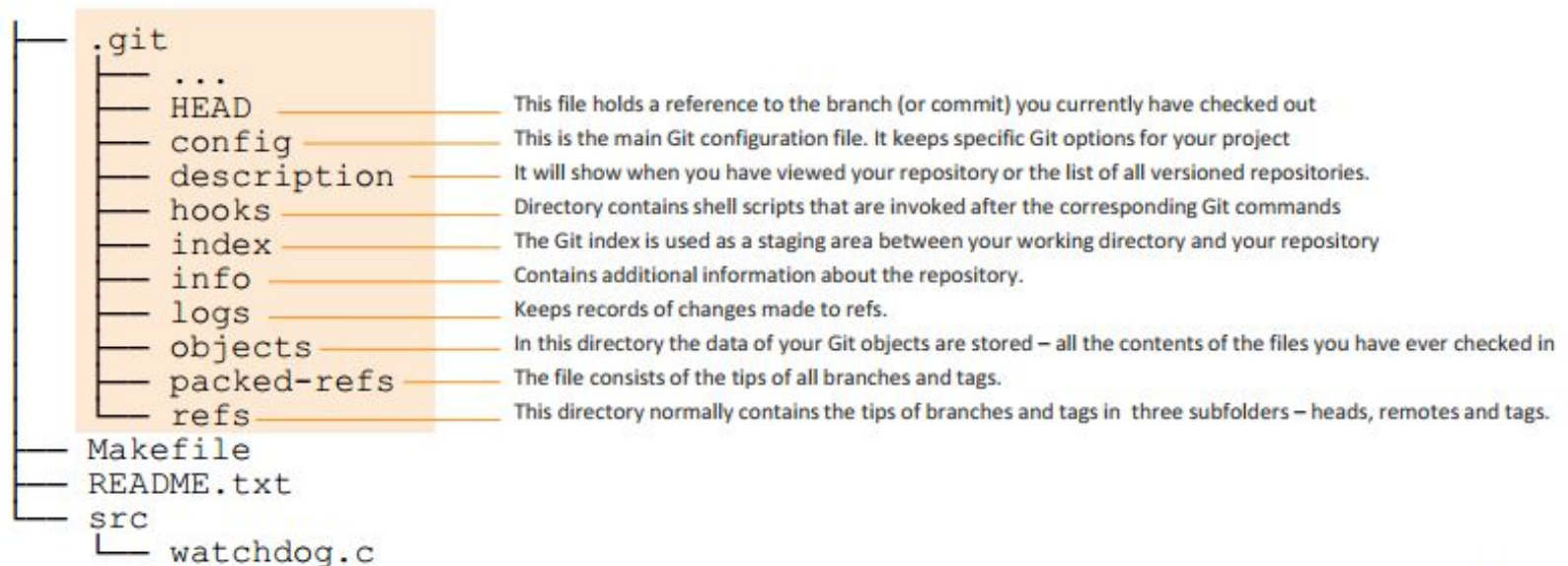
Clone it to work repo1 in home dir also,

add,commit and push

- Restructure files, add, commit and push

Git repository structure (.git)

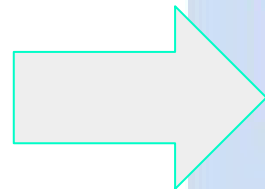
```
$ ls -al
total 11
drwxr-xr-x  7 sheta  Administ 4096 Dec  3 15:17 .
drwxr-xr-x  3 sheta  Administ 4096 Nov 30 11:26 ..
-rw-r--r--  1 sheta  Administ  23 Nov 30 11:09 HEAD
-rw-r--r--  1 sheta  Administ 363 Nov 30 11:46 config
-rw-r--r--  1 sheta  Administ  73 Nov 29 17:03 description
drwxr-xr-x  2 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
```



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:

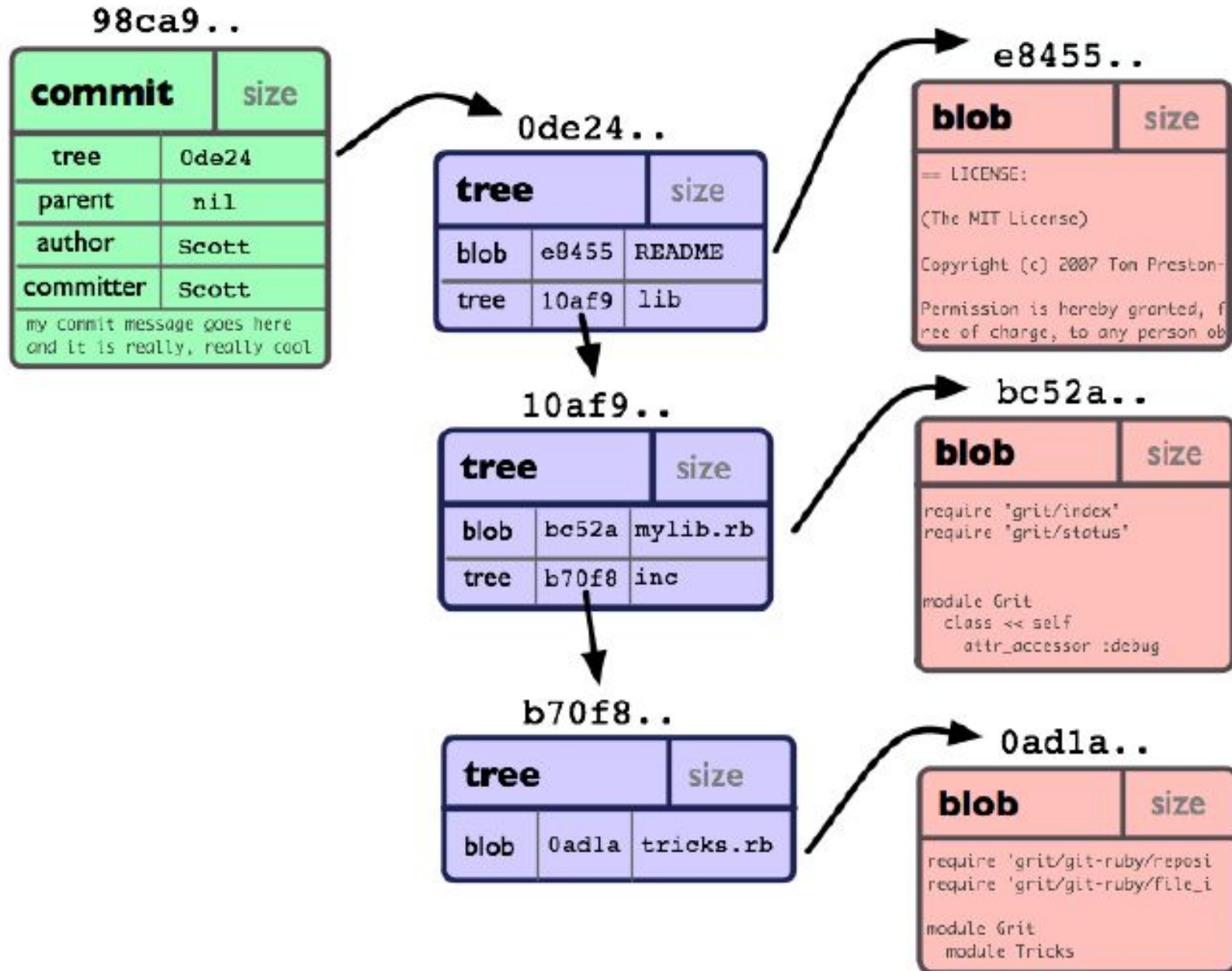
```
$>tree
```

```
.  
|-- README  
`-- lib  
    |-- inc  
    |   |-- tricks.rb  
    |-- mylib.rb
```

```
2 directories, 3 files
```

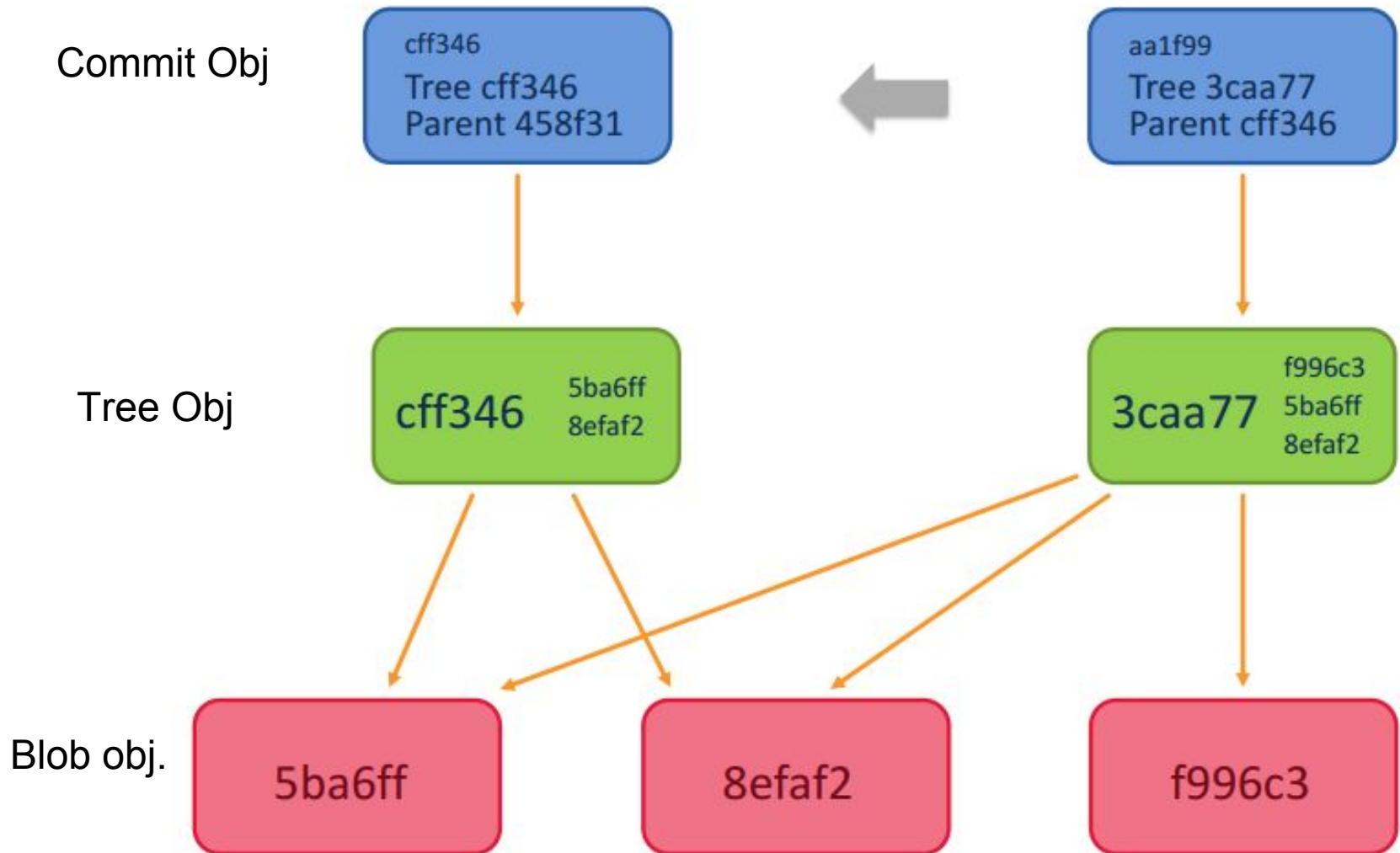
If we will commit this project to a GIT repository, it will be represented in GIT like this:

Commit Object



Commit object with its parent

- Links from tree object to common blobs



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

```
MINGW32:/c/Users/Ilya/sally1
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

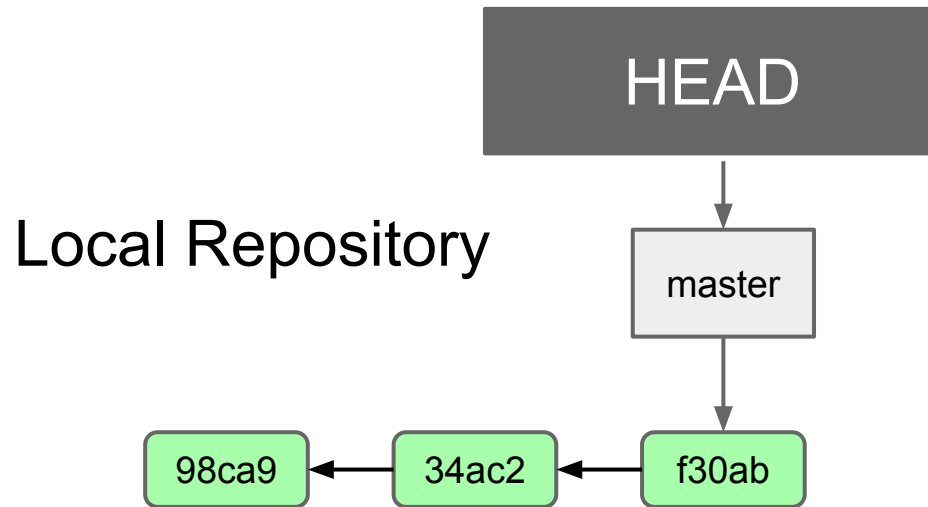
    libs/library.txt | 2 +-
    1 file changed, 1 insertion(+), 1 deletion(-)

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

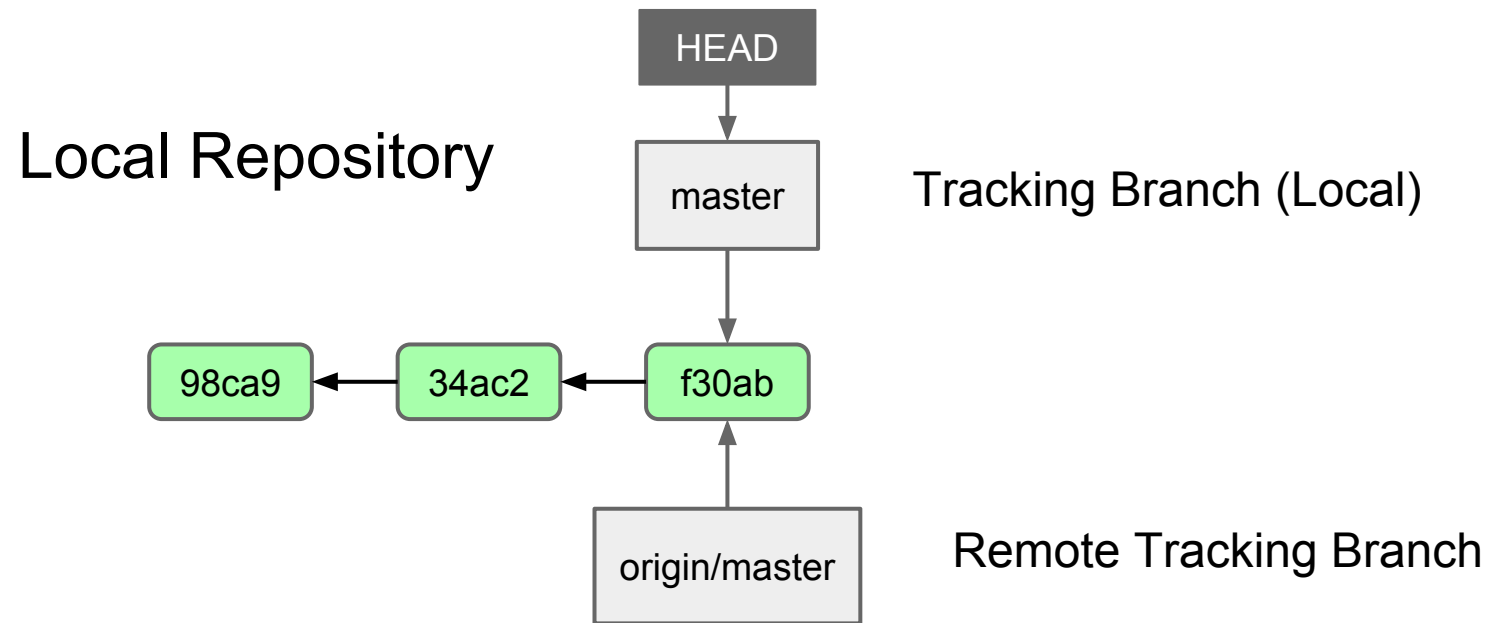
Demo

Lab 3

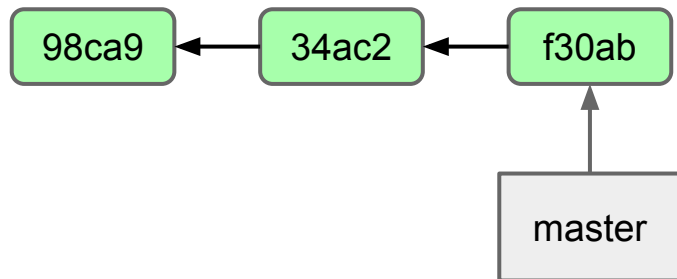
Teamwork, parallel work:

- 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

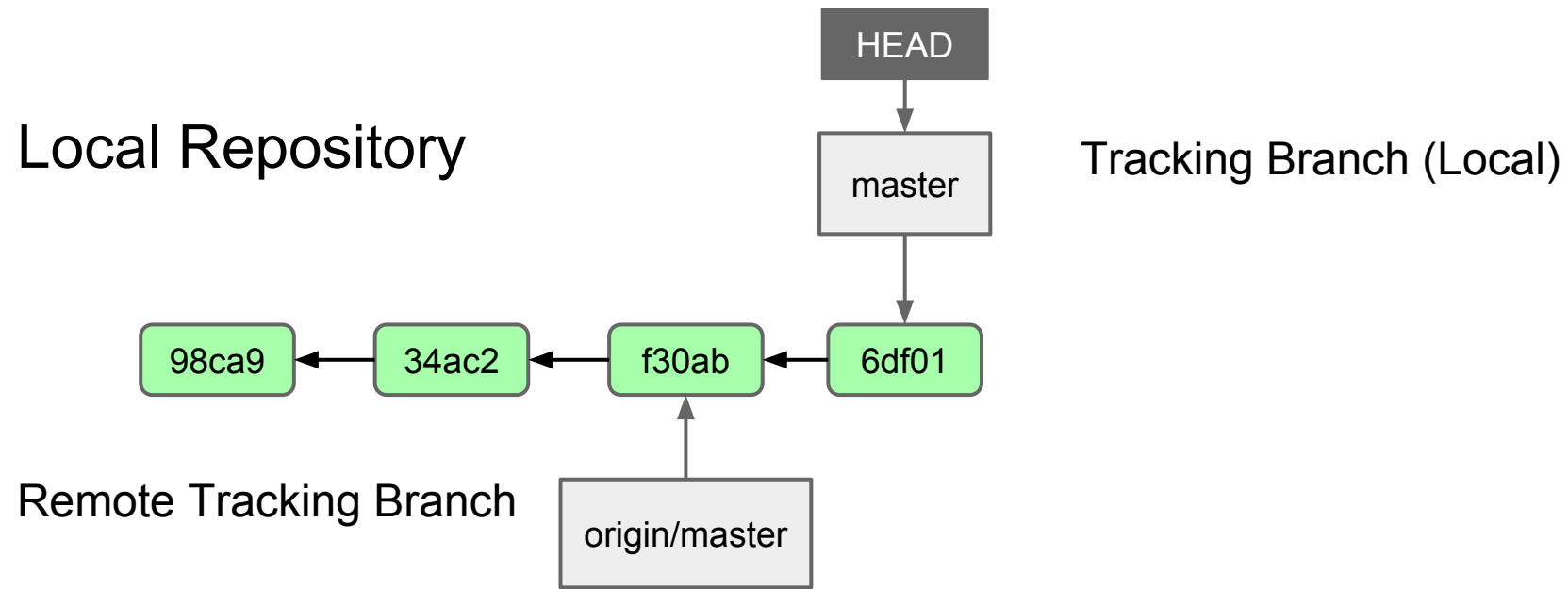


Remote Repository



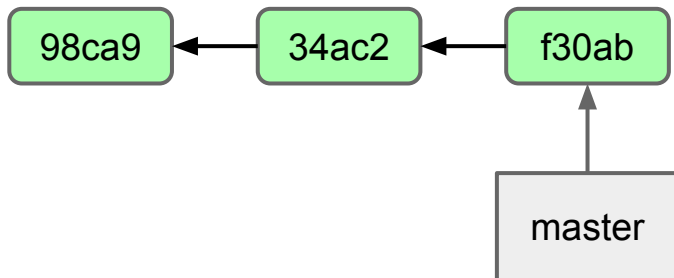
- After clone/pull Remote Tracking Branch and Tracking (Local) branch pointing to the same commit

Rebasing Remote Tracking Branch

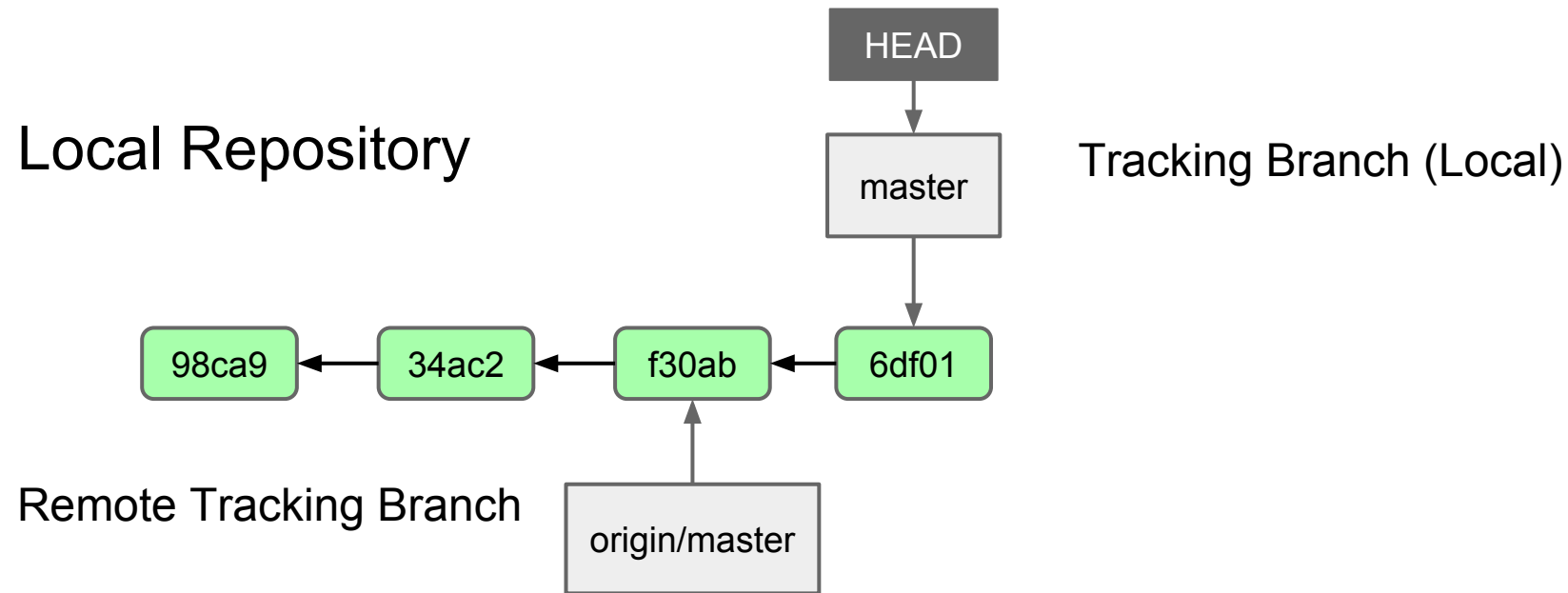


- **Commit on local repository**
\$ git commit

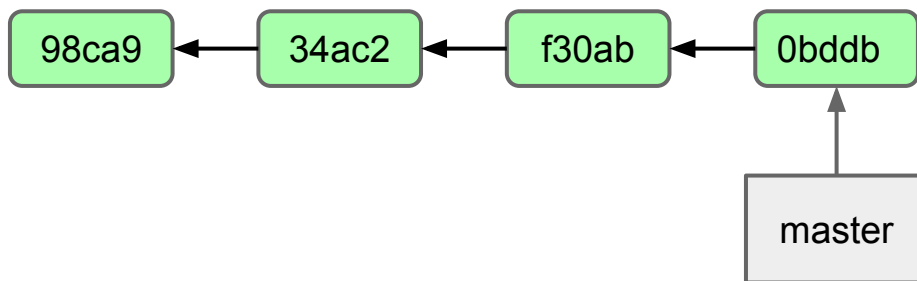
Remote Repository



Rebasing Remote Tracking Branch



Remote Repository

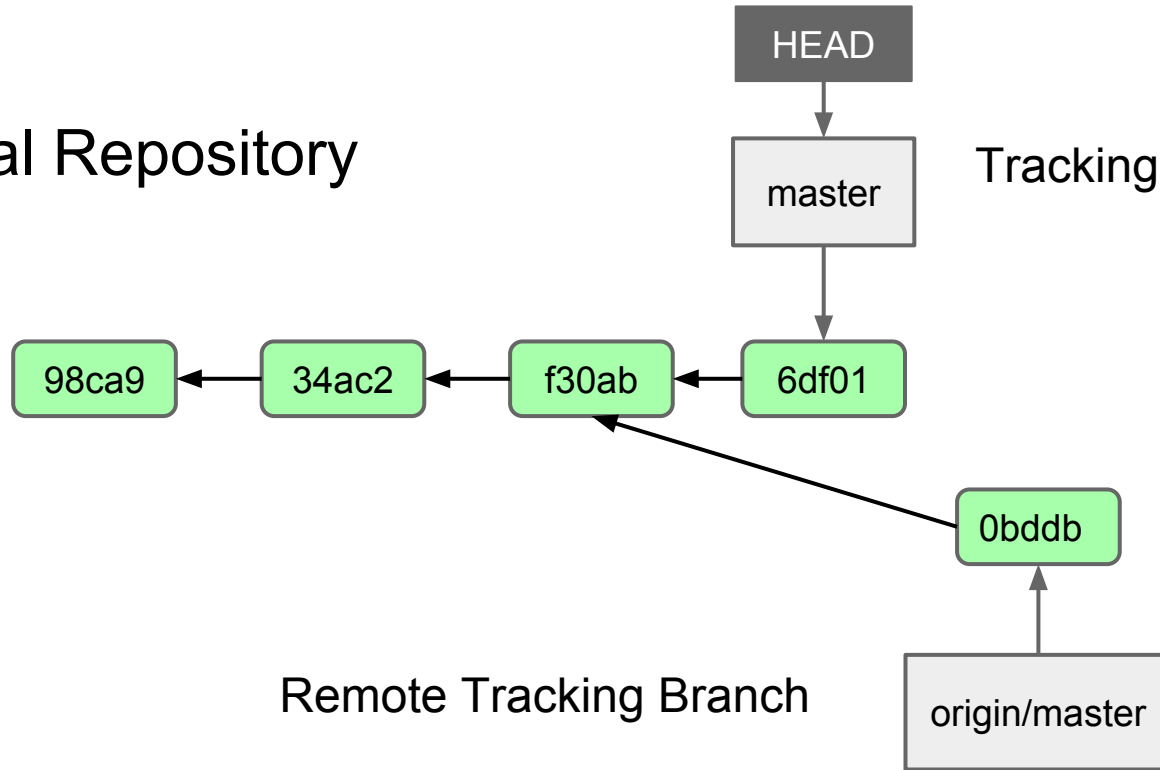


- **Push another Commit to remote repository from another repository/user**

Rebasing Remote Tracking Branch

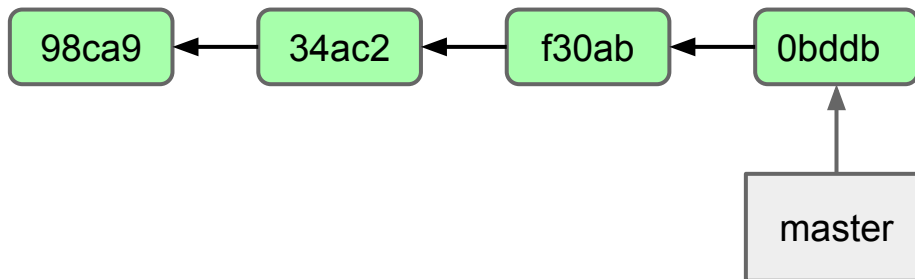
Local Repository

Tracking Branch (Local)



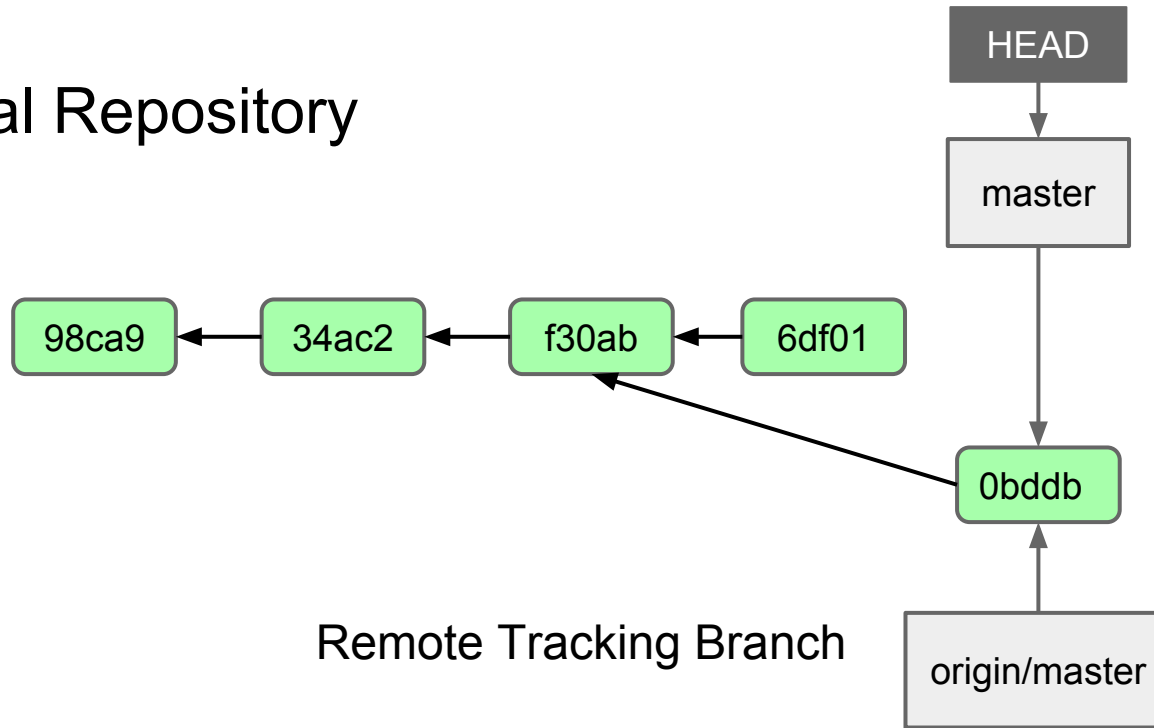
- **Fetch/Pull the Commit from remote repository to Remote Tracking Branch**

Remote Repository



Rebasing Remote Tracking Branch

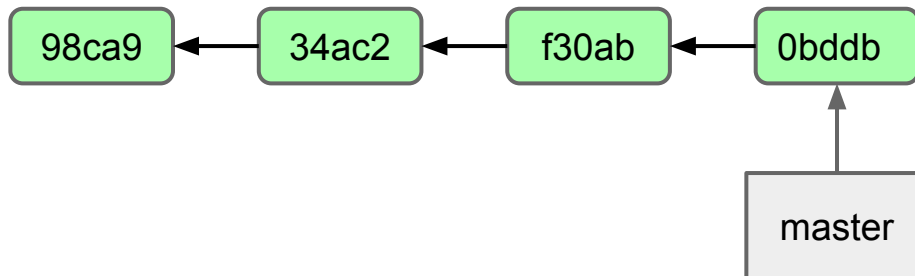
Local Repository



Tracking Branch (Local)

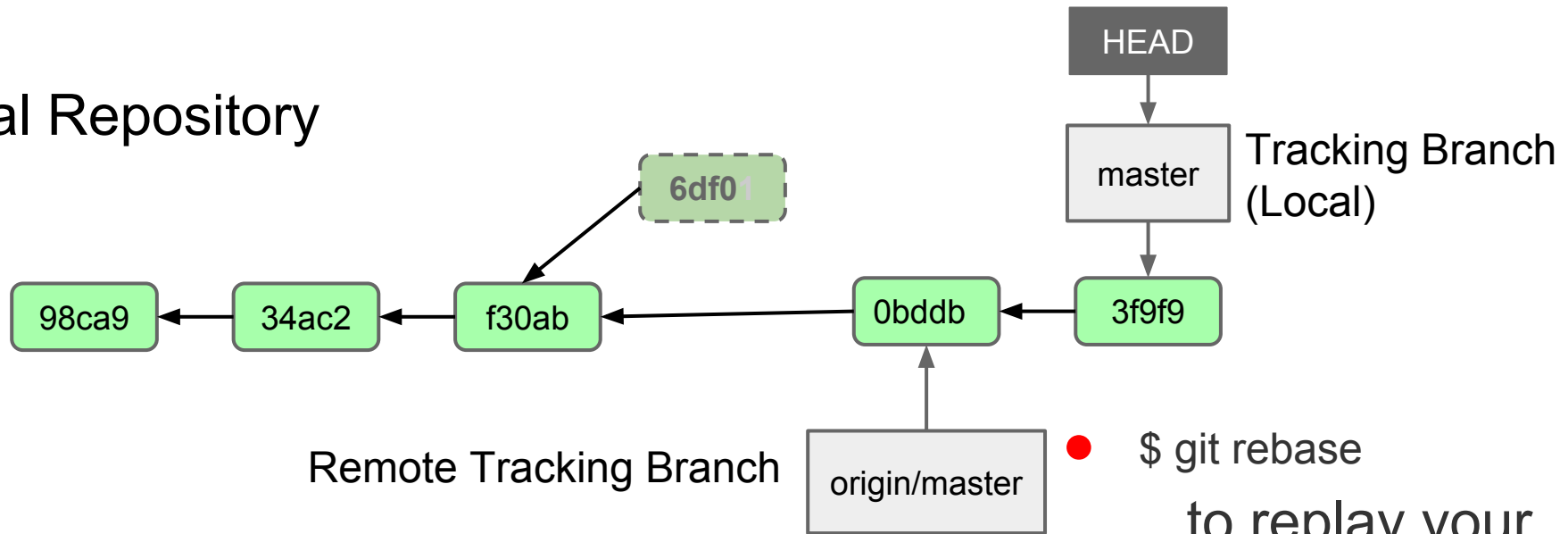
- \$ git rebase
- First, rewinding head to replay your work on top of it...

Remote Repository



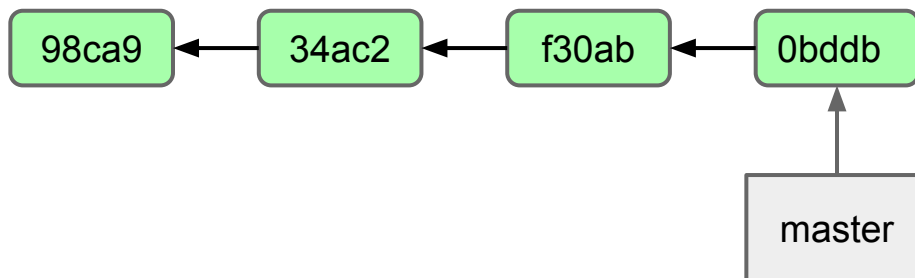
Rebasing Remote Tracking Branch

Local Repository



● \$ git rebase
...to replay your
work on top of
it...

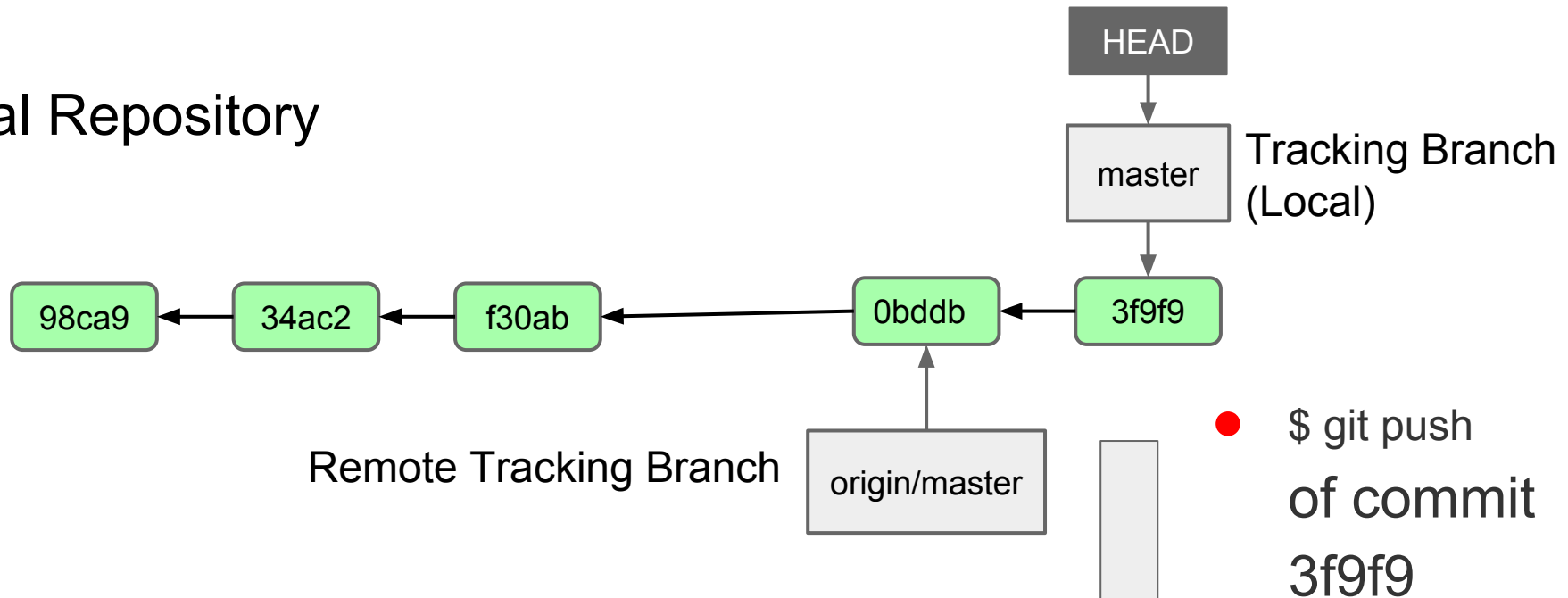
Remote Repository



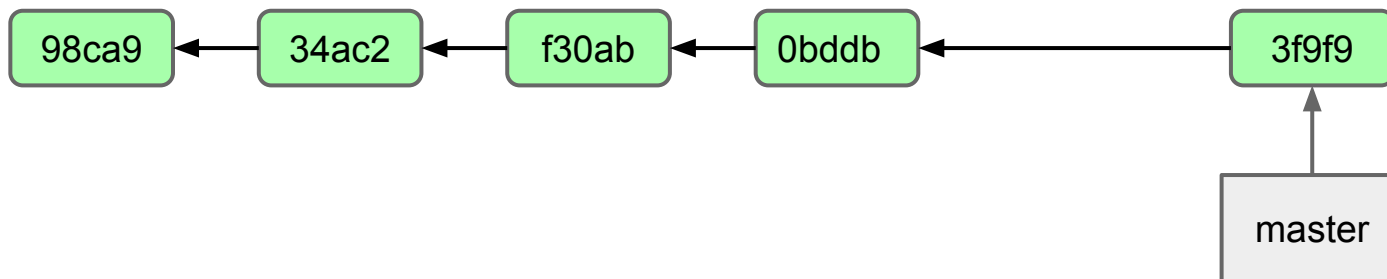
Applying: diff/patch
of commit 6dfo into
commit 3f9f9

Rebasing Remote Tracking Branch

Local Repository

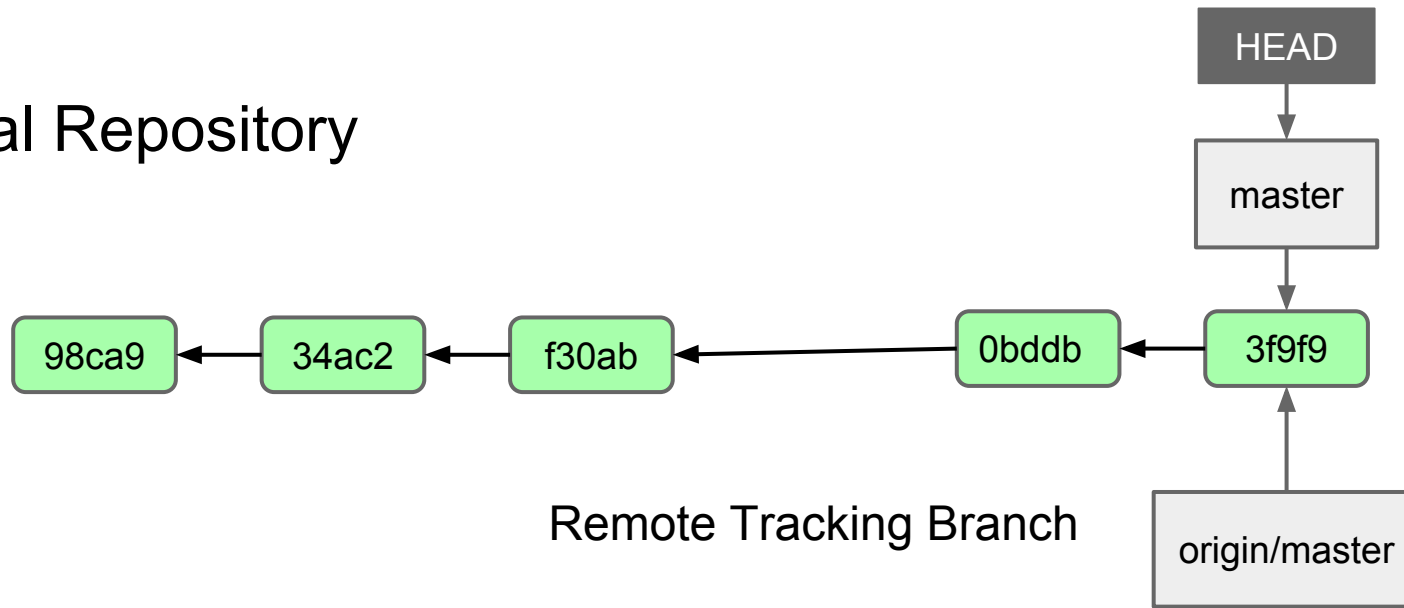


Remote Repository



Rebasing Remote Tracking Branch

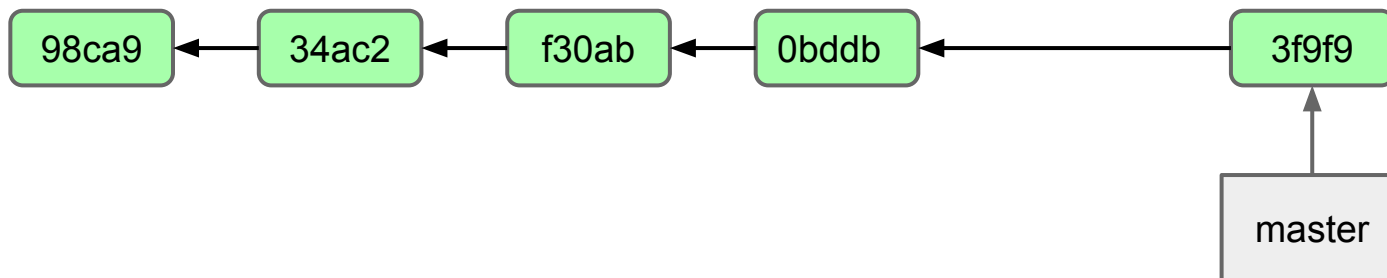
Local Repository



Tracking Branch (Local)

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

Remote Repository



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.

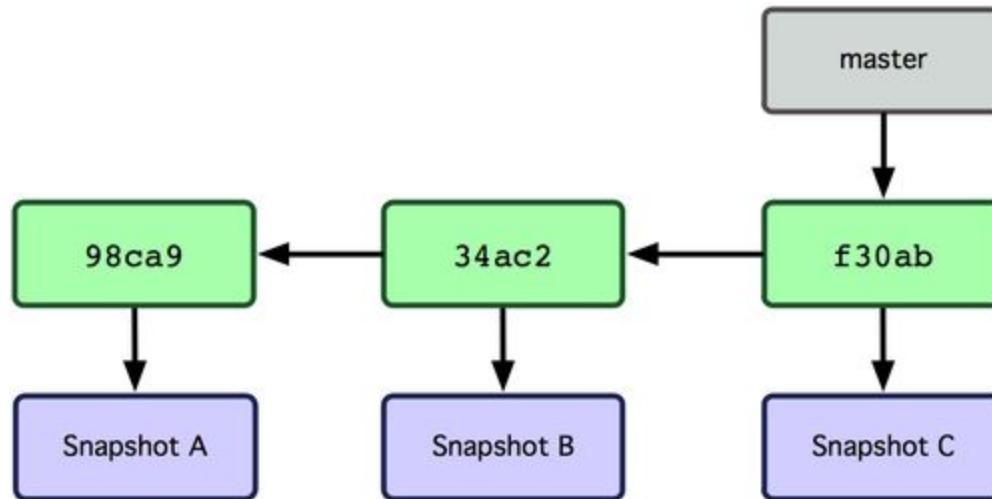
Demo

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

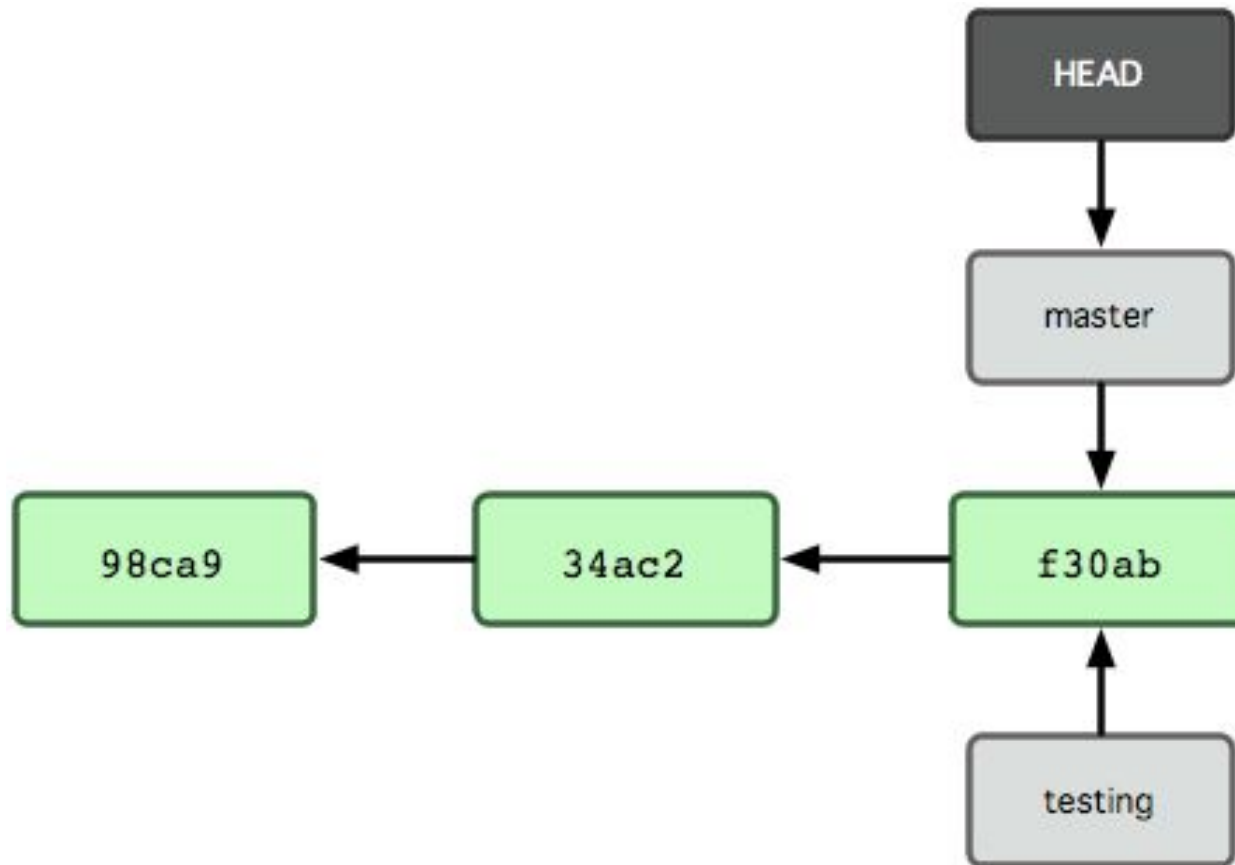
- Git branch is simply a movable pointer to a commit



- Pointer moves forward automatically with each commit on a branch

Creating new branch

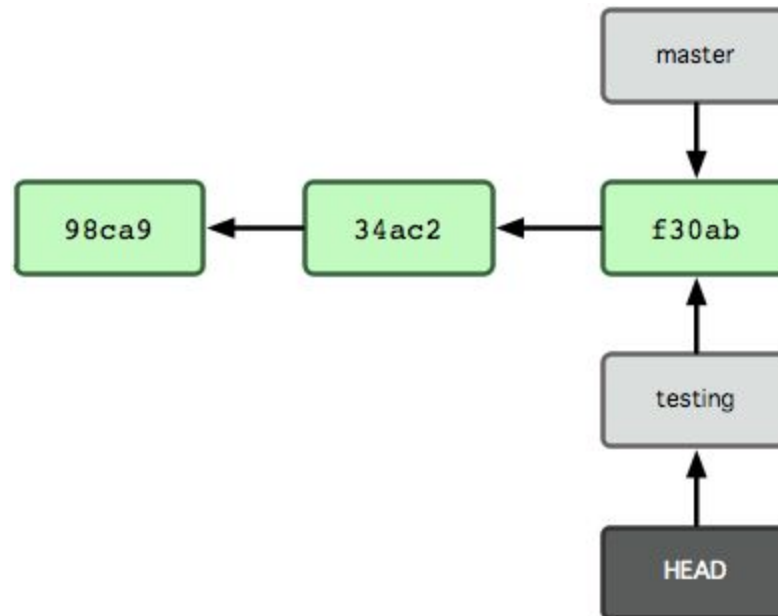
- New branch creates a new reference
 - > `git branch testing`



Switching to a branch

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

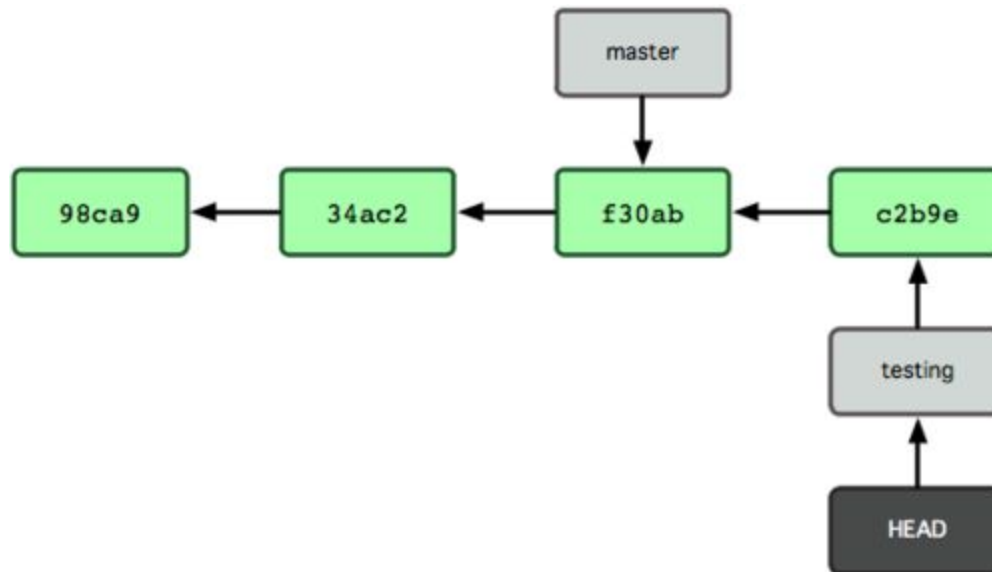
> `git checkout testing`



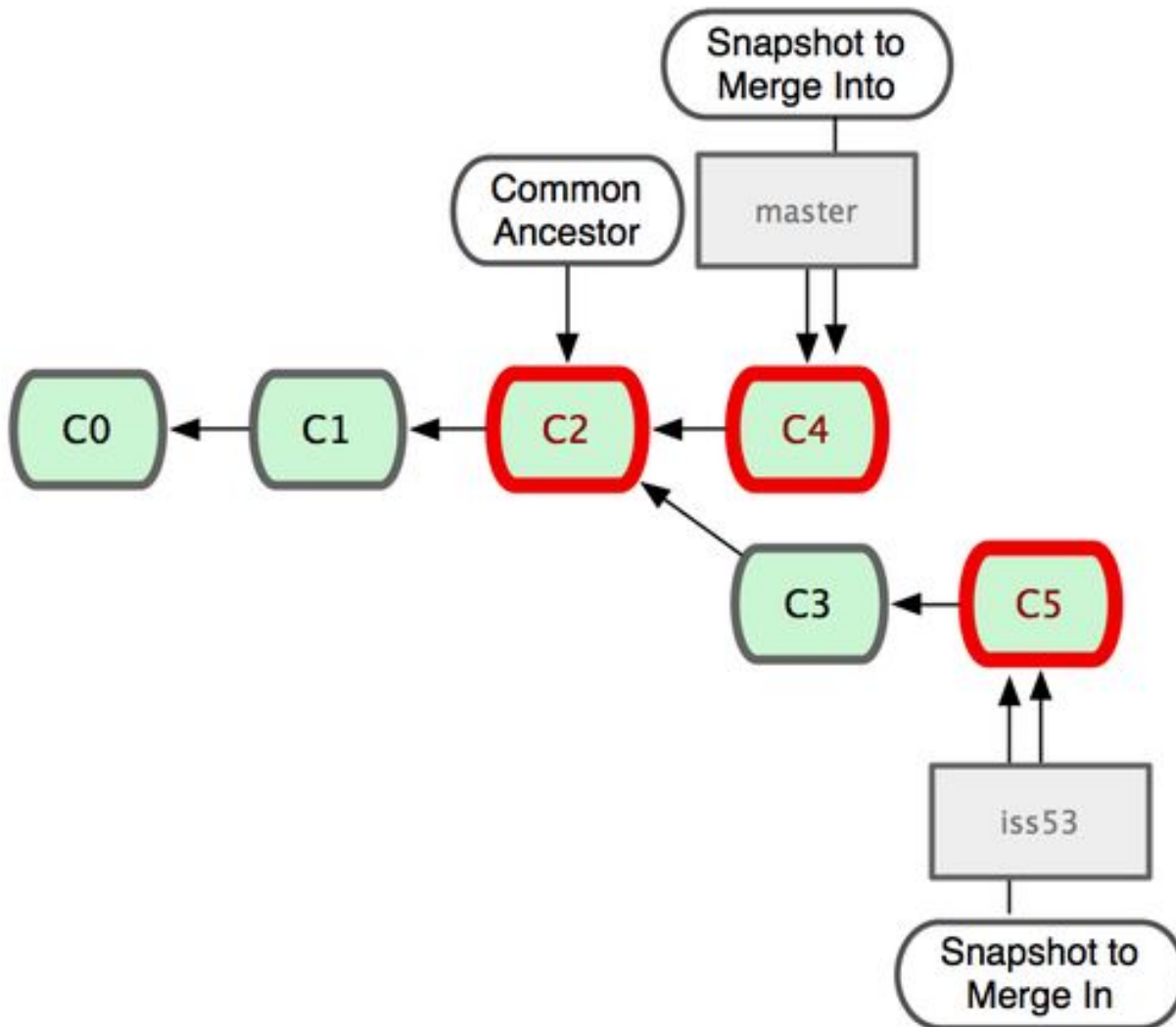
New commit moves current branch

```
> vi file04.txt
```

```
> git commit -a -m 'Commit message'
```

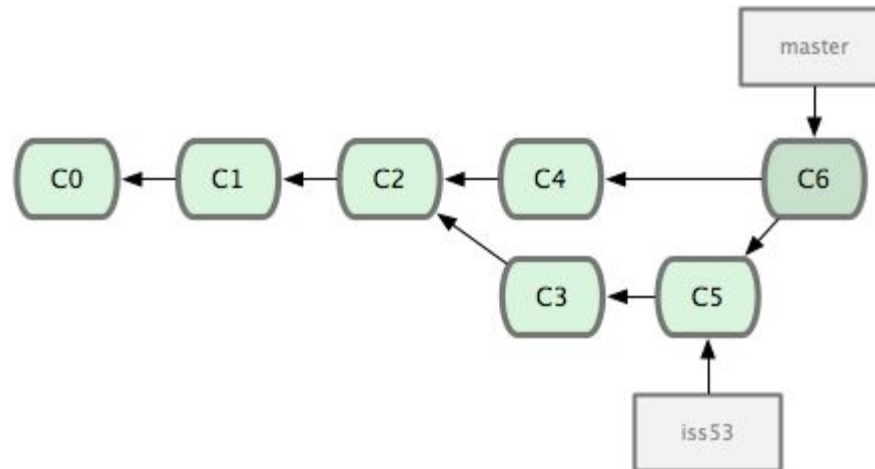


Merging branches



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

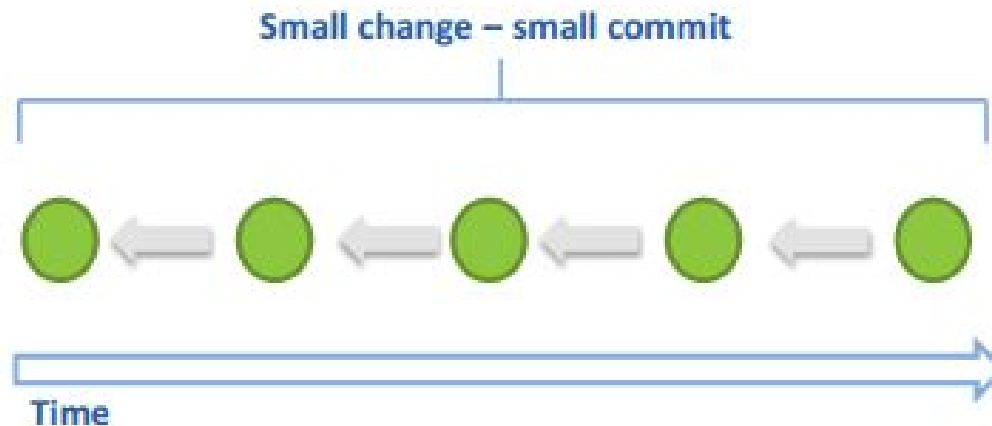
Check out master branch and commit new change

- Merge bugfix branch

Overview results

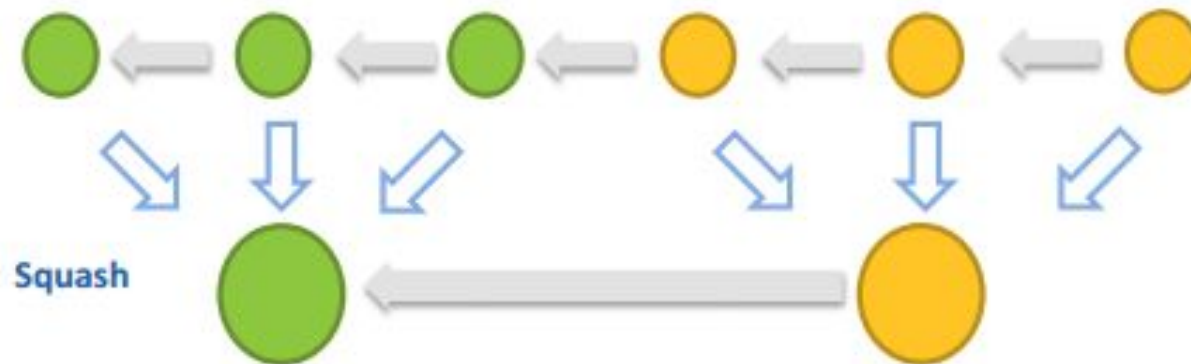
Best practices – commit

Keep changes small wherever possible and commit frequently



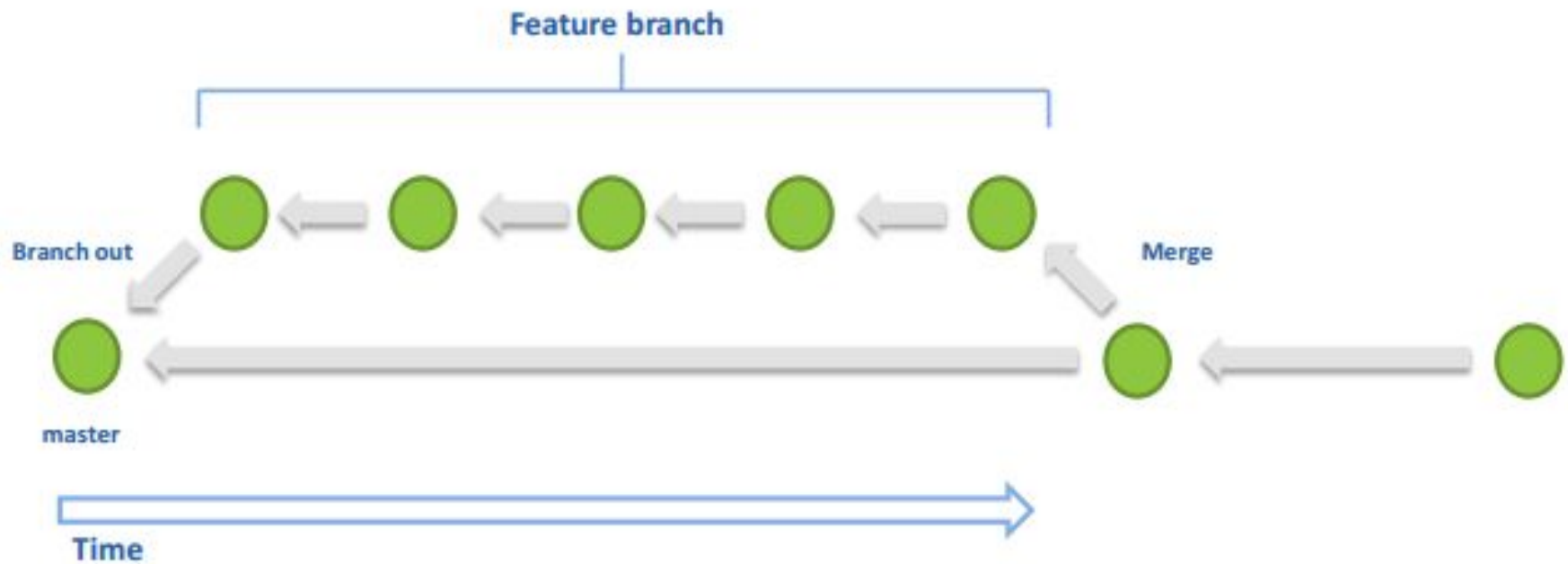
Best practices – squashing

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



Best practices – local feature branches

Work on feature branches locally



Best practices – Branch Layout

Branch layout

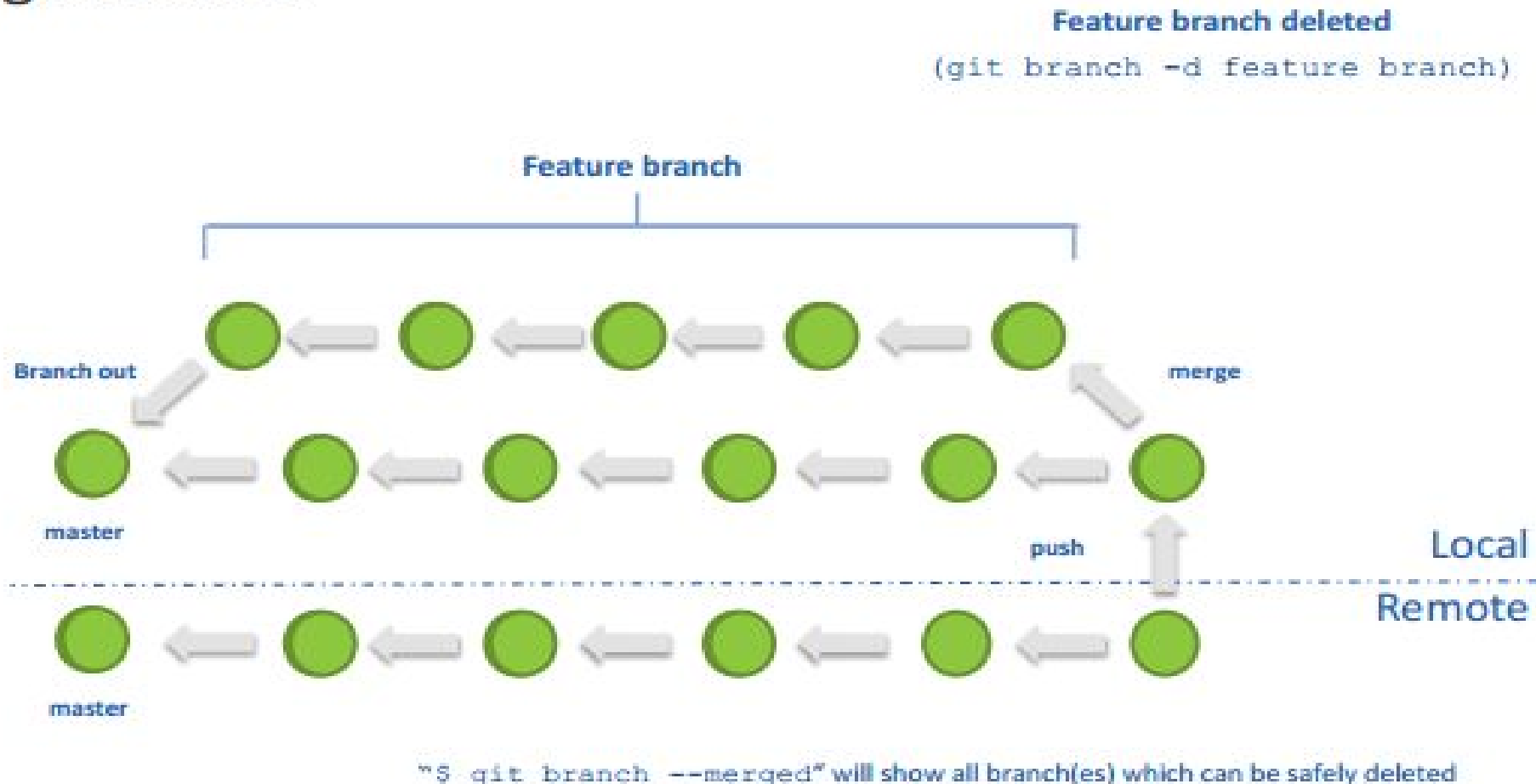
The branch layout is up to you, but there are some best practices though:

```
$ 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
```

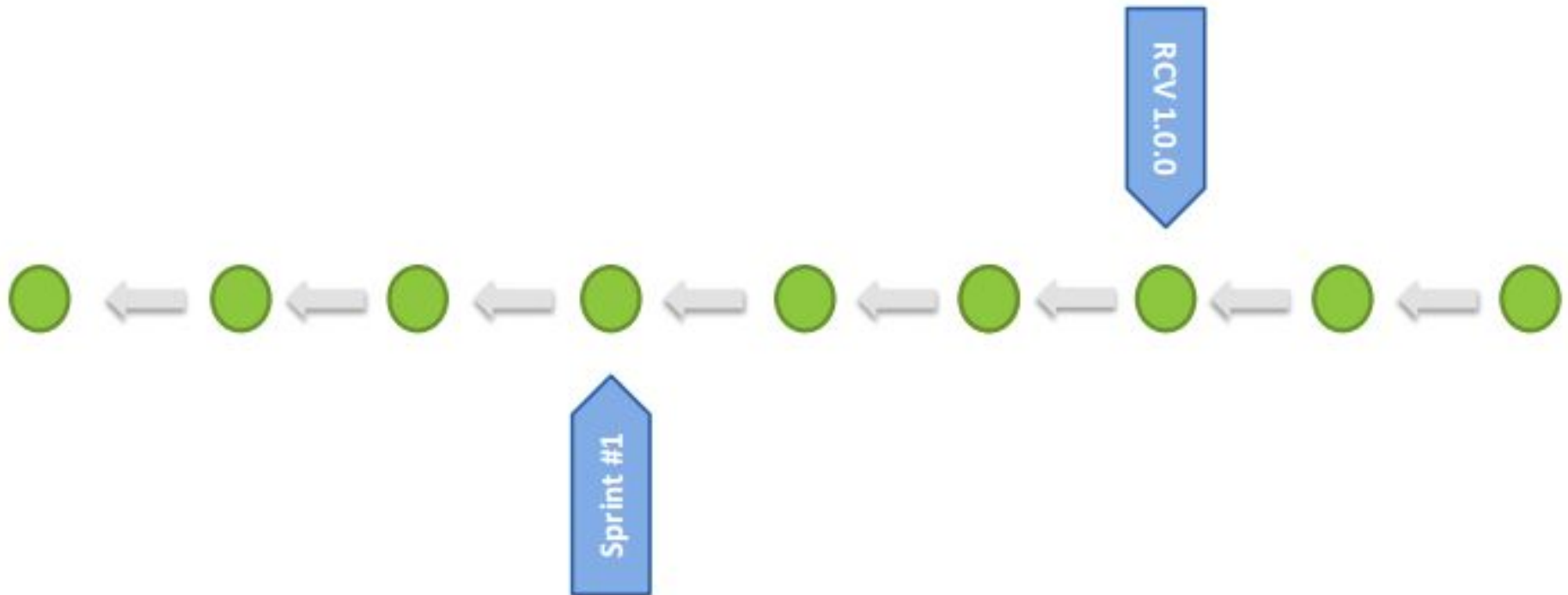
Best practices – clean up local branches

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



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

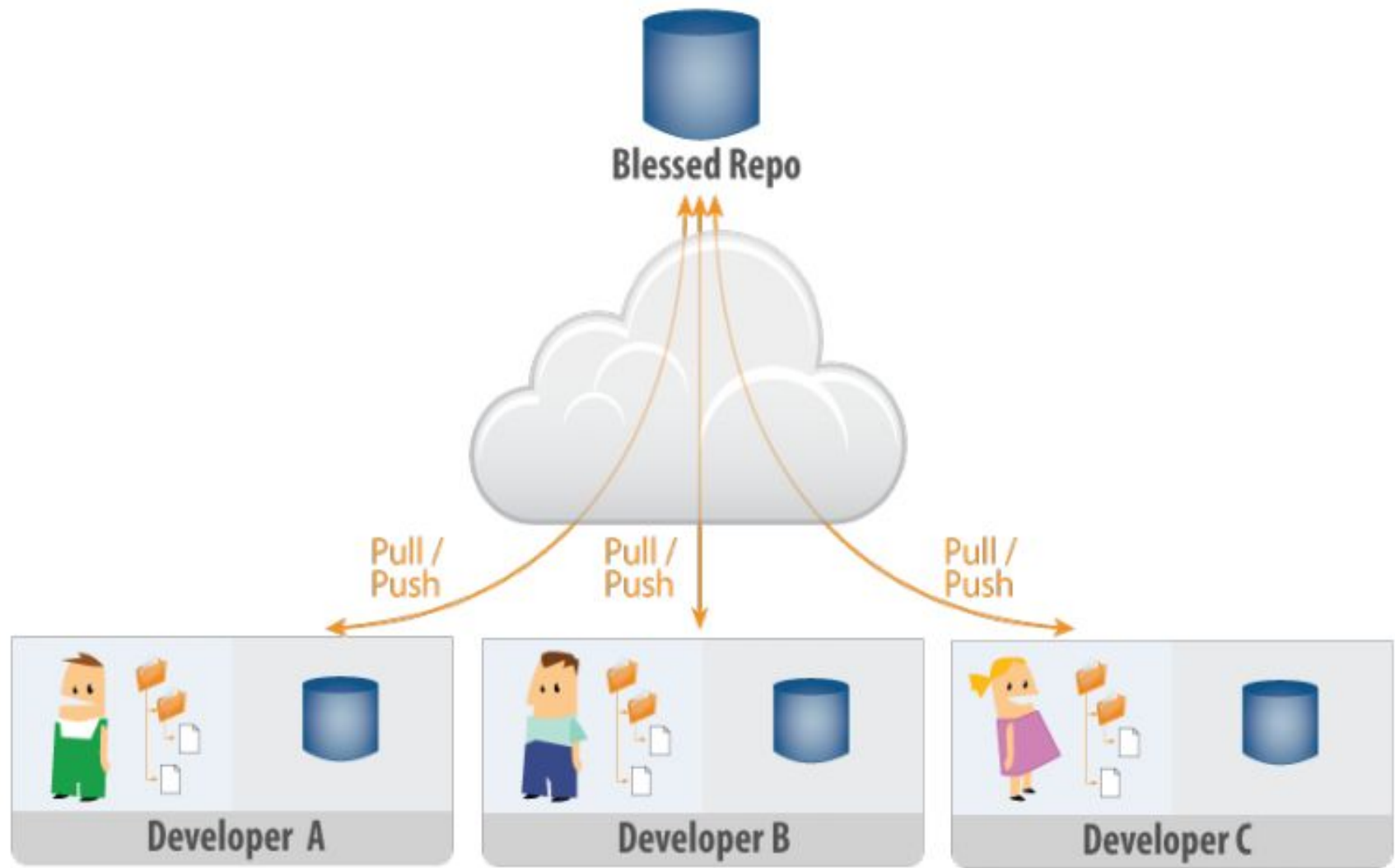
- Format patch in 1 repository, and apply it in another repository
- We will create 2 local commits and squash them to 1 commit by interactive rebase, and push only 1 commit to remote repository
- Create commit in 1 repository and pull it from another repository, without pushing to origin repository

Question?

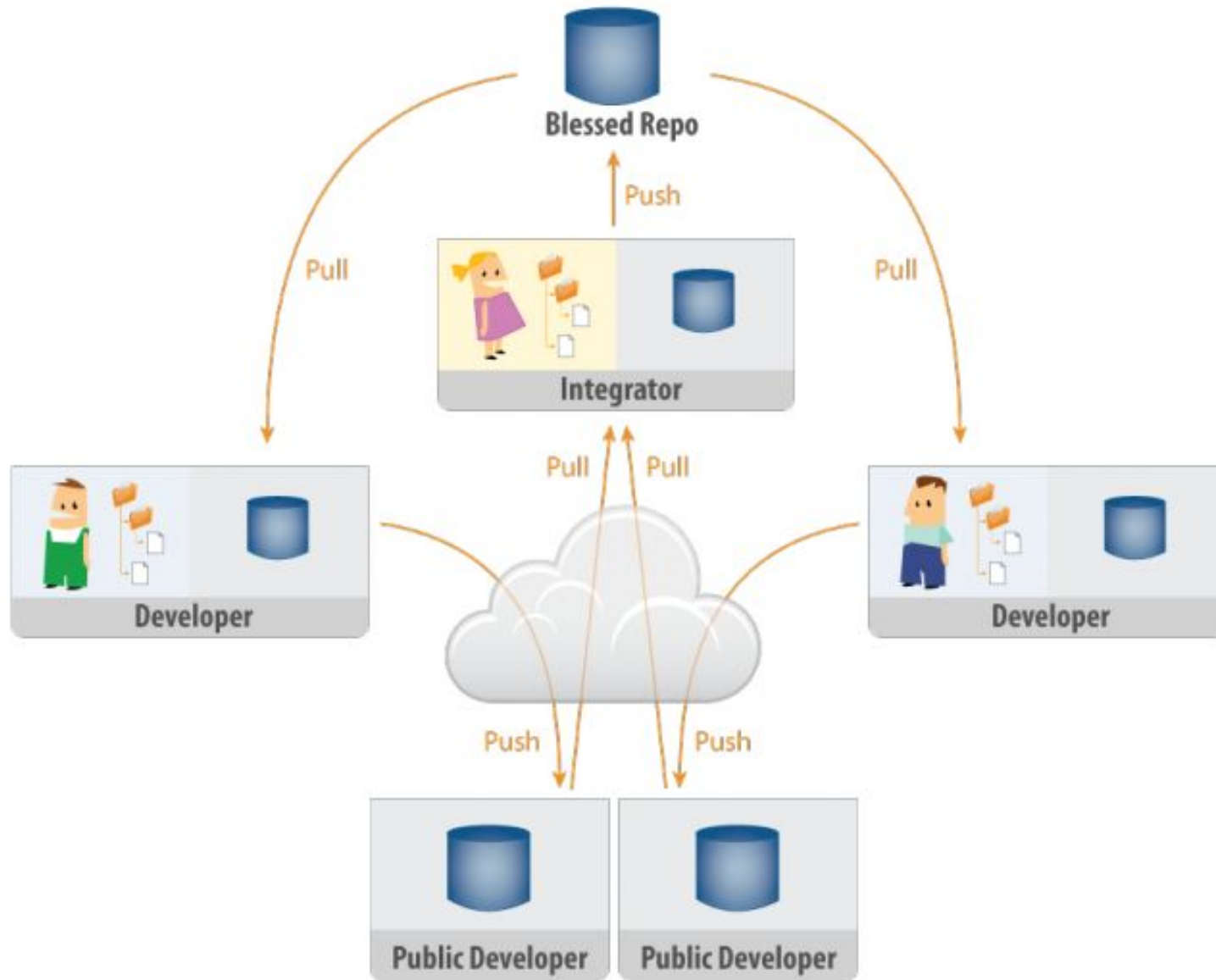
Thanks!

Backup slides

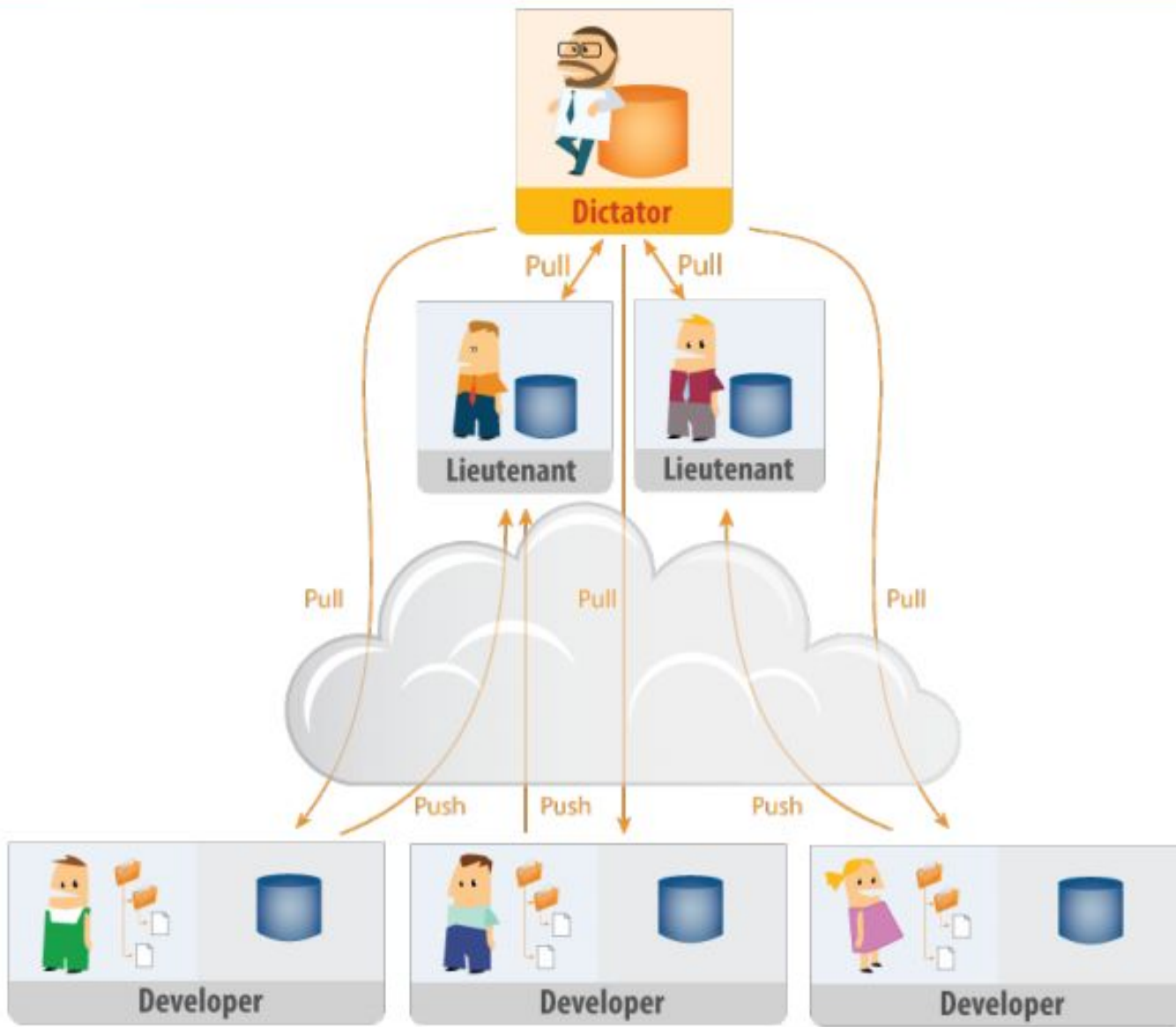
Centralized Workflow



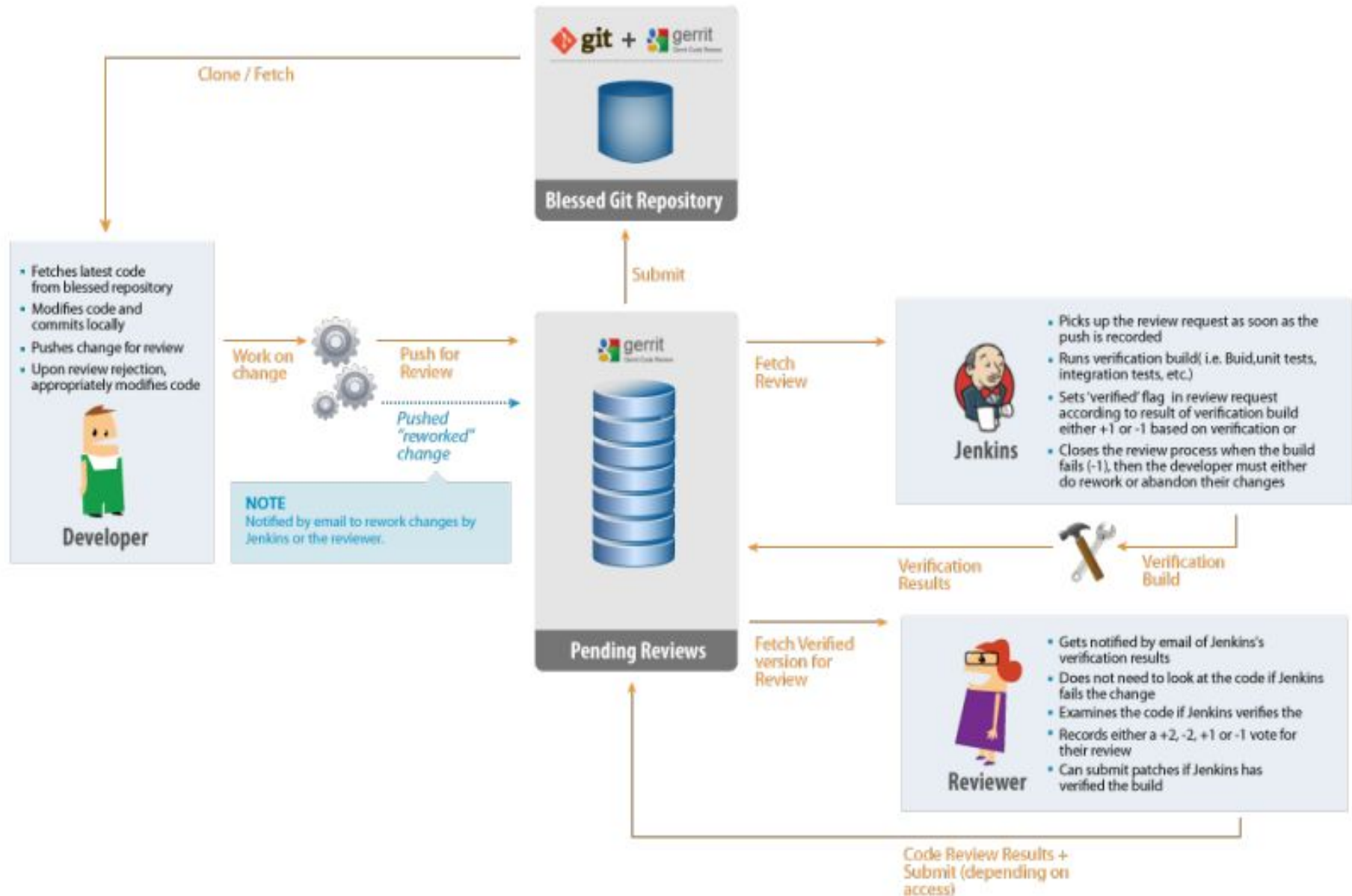
Integrators Workflow



Dictator / Lieutenants Workflow



Gerrit Code Review Workflow



Detached HEAD

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



```
$ git checkout 494e2cb73ed6424b27f9766bf8a2cb29770ale7e
Note: checking out '494e2cb73ed6424b27f9766bf8a2cb29770ale7e'.
```

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 committed:
#   (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
```

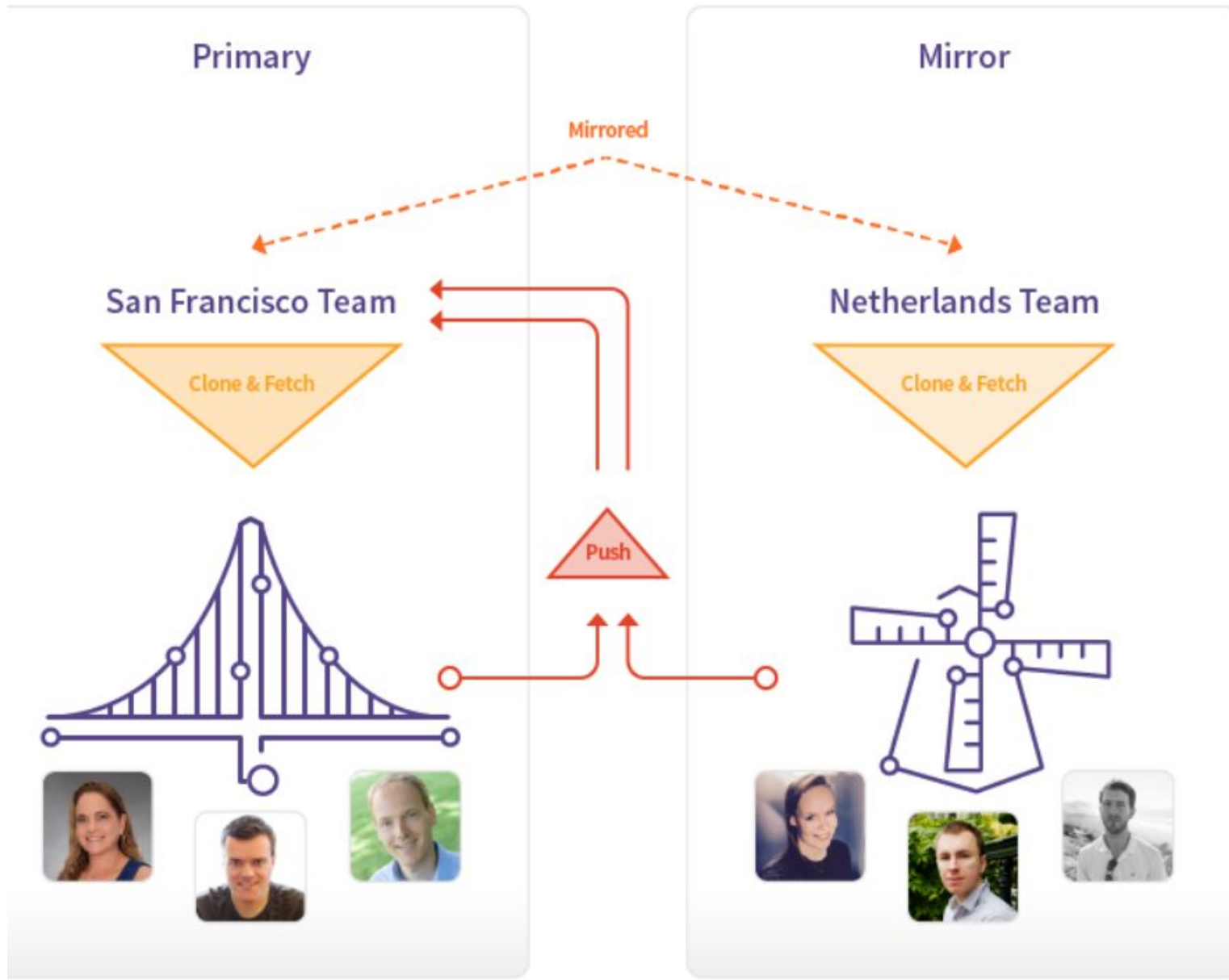
git stash takes current state of your working directory (what is staged, modified, etc.) and saves it as a stack of unfinished changes in **refs/stash**.

```
$ 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

