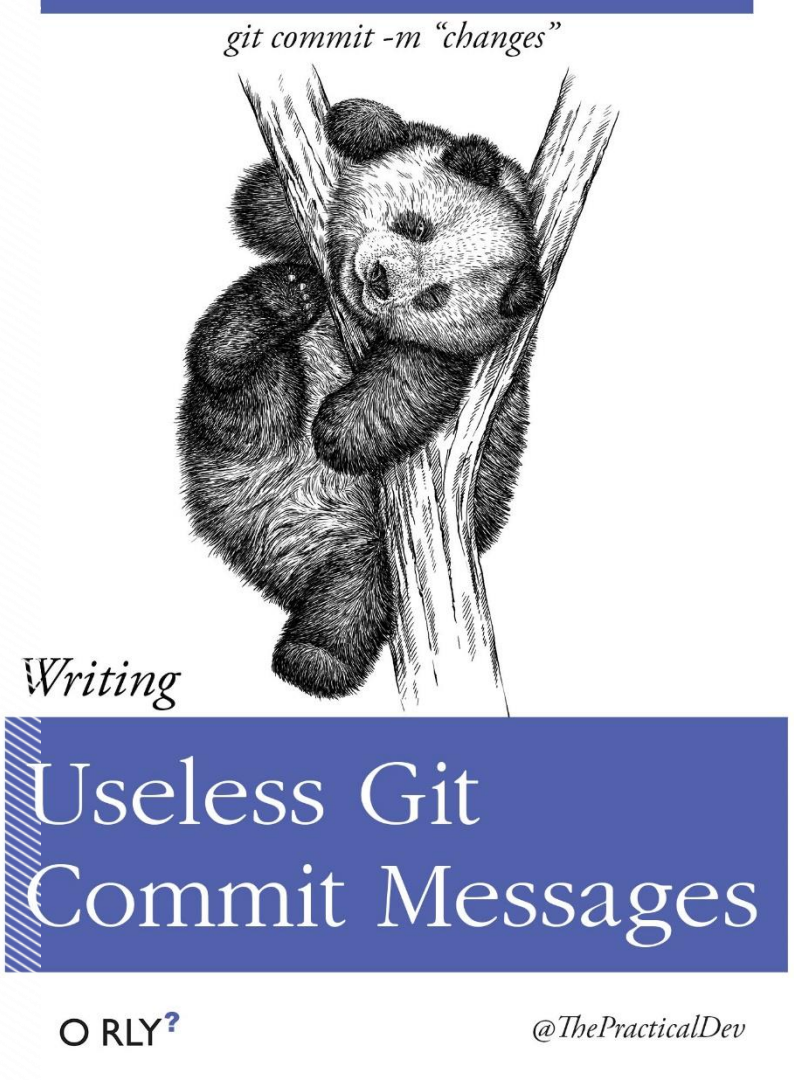
Git Basics

Frontend Junior Program - 2022



Agenda

1. Intro
2. Git
3. Git commands
4. Merge and Rebase
5. Branching strategies

About

Version Control System –

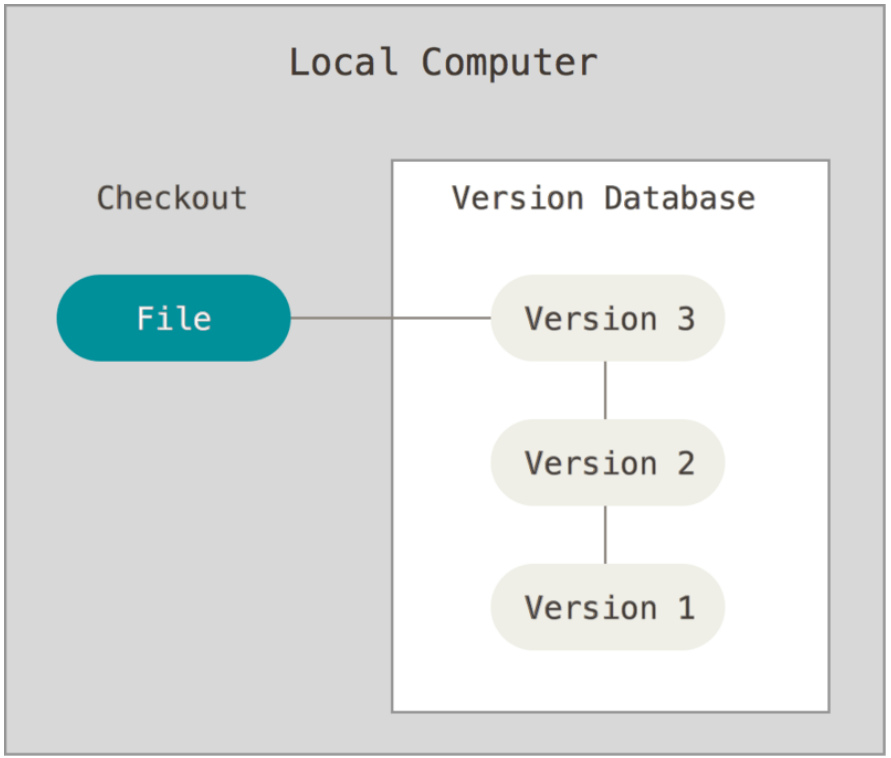
a system that stores file or file state (e.g., date, owner) changes during a period of time and the one that allows to return to a specific file state (file version).

VCS types:

1. Local
2. Centralized
3. Distributed

Local version control system

As a rule, local VCS has a database that stores information regarding all changes in project files, thus performing version control.

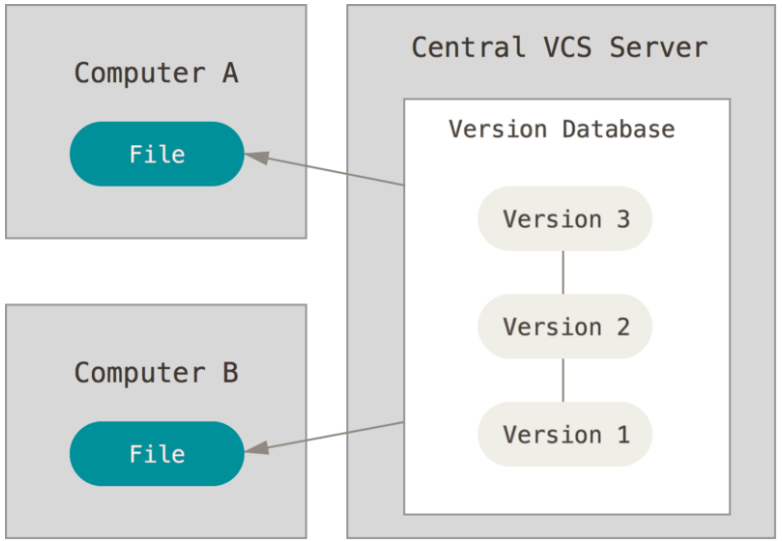


Centralized version control system (CVCS)

Most popular examples are **CVS** and **SVN**.
Such VCS consists of a single server (containing all the file versions) and some clients that receive files from its centralized storage.

Advantages:

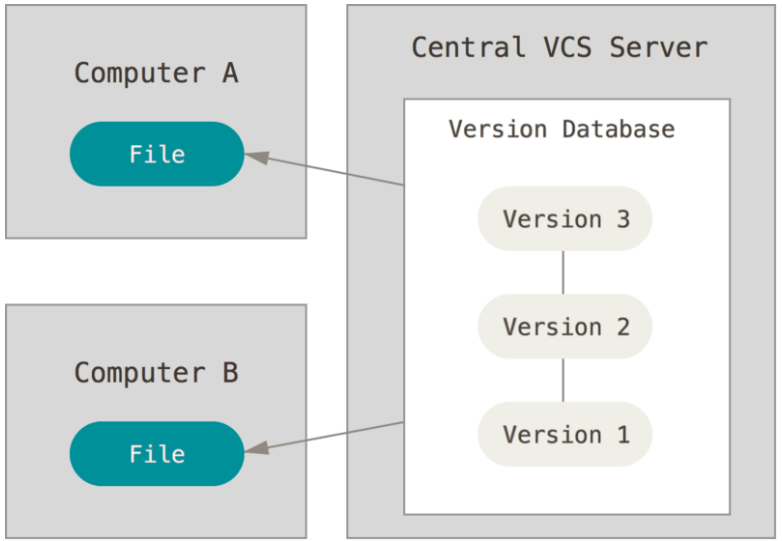
- Visibility – all project developers know what is being done by each one of them.
- Administration – it is easier to administer CVCS than to operate local databases on each client.



Centralized version control system (CVCS)

Disadvantages: single point of failure, represented by a centralized server.

- If this server is not working for an hour, during that time no one will be able to use version control.
- If the hard drive (where centralized DB was stored) has been damaged and recent backups are missing, you will lose all project history, excluding single repository snapshots that are stored on local machines.

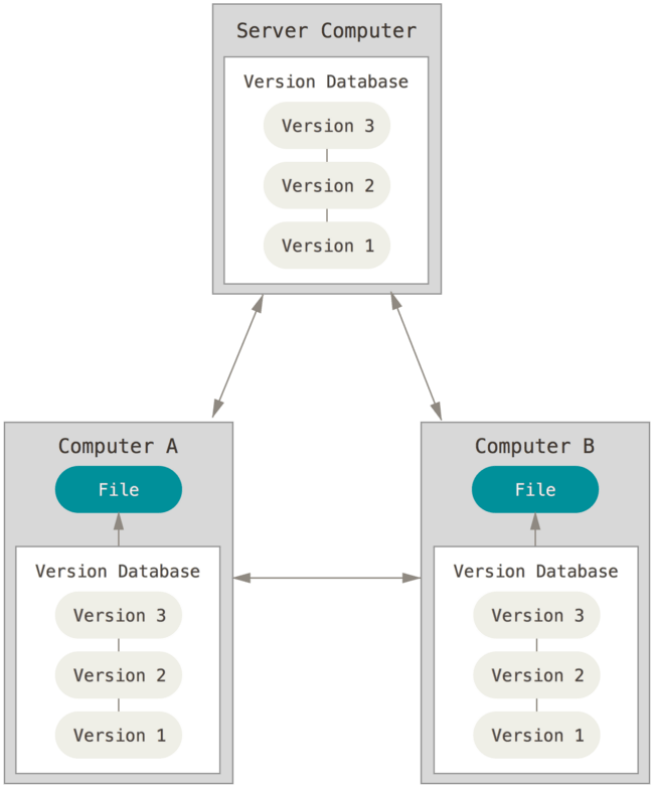


Distributed version control systems (DVCS)

The main difference between centralized and distributed version control systems is that the clients do not just download snapshots of all files (file status at specific moment): they fully copy the repository.

In this case, if one of the servers (where developers exchange information) will die, any client repository can be copied to a different server in order to continue work.

Each repository copy is a full backup of all data.

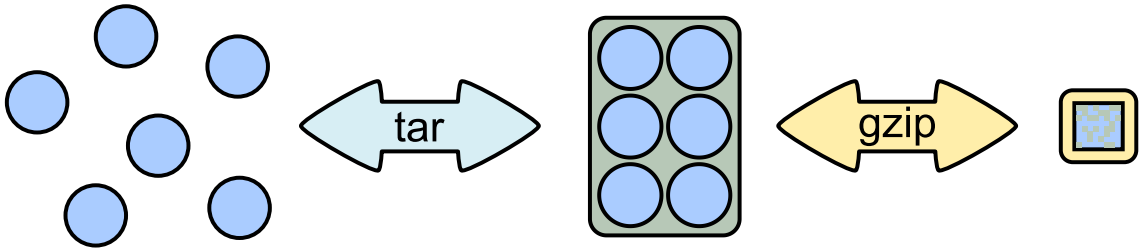


Git History

- 1991-2002: Linux development uses tarball
- 2002: Linus Torvalds uses BitKeeper for Linux
- April 6, 2005
  - BitKeeper drops free license on some tools
  - Linus writes this own DVCS: GIT
- June 16, 2005
  - GIT is officially used to track Linux

"I'm an egotistical bastard, and I name all my projects after myself. First `**Linux**`, now `**git**`."

Tarball

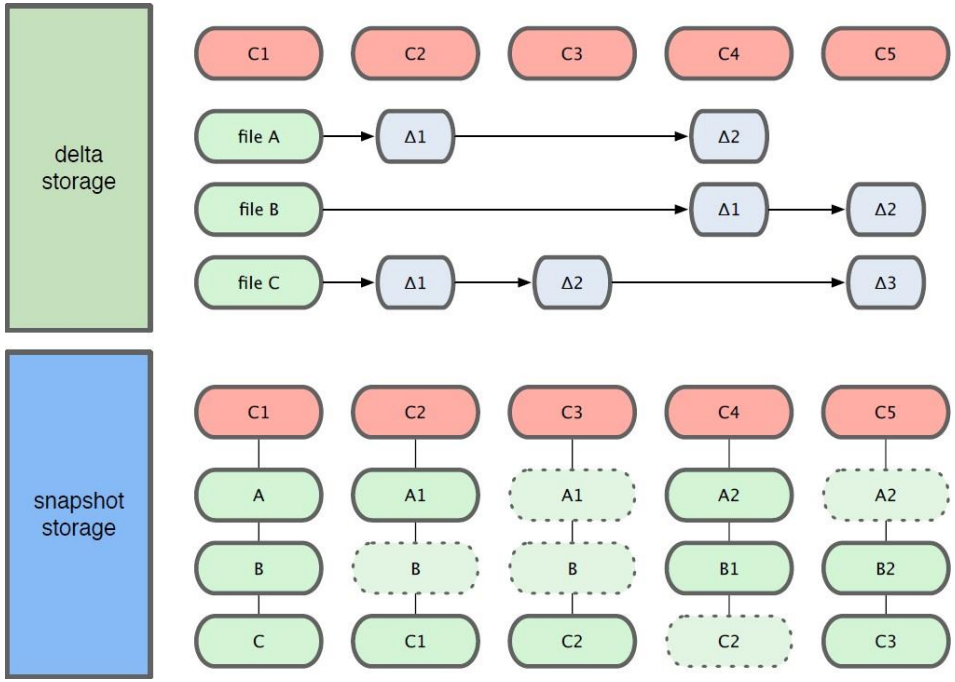


Git basics: How GIT stores information about changes

1) Data storage as set of changes from the initial version of each file.
(CVS, Subversion, Perforce, Bazaar and others)

2) Data storage as project snapshots over time.

Notice:
For productivity increase, if the files had not been changed, Git does not save these files again. It creates link to previous version of identical file that has been already stored.



Installing

Windows –

- https://git-scm.com/download/
- https://gitforwindows.org/

Linux –

- `apt-get install git`
- `yum install git-core`

macOS:

- `brew install git`

Initial setup Git: git config

git config – command line utility that allows to view and set parameters, which control all aspects of working with Git.

Initial commands:

```
$ git --version
$ git config -list                                  //view of current parameters
$ git config --global user.name "John Doe"          // parameters setup
$ git config --global user.email johndoe@example.com
$ git config user.name                              // parameters view via key
```

Help:

```
$ git help
$ git help <command>
$ git <command> --help
```

.gitignore

.gitignore file – defines files or file patterns that GIT should not manage – matching files will be ignored by GIT commands.

Such files usually include automatically generated files (build result, logs, results of program setup etc.)

```
# compiled output
/dist
/tmp
/out-tsc

# Runtime data
pids
*.pid
*.seed
*.pid.lock
```

Creating Git-repository

Repository creating

```
# creates in current directory new subdirectory with name .git (Git-directory)
$ git init [folder]

$ git add .
$ git commit -m 'init commit'
```

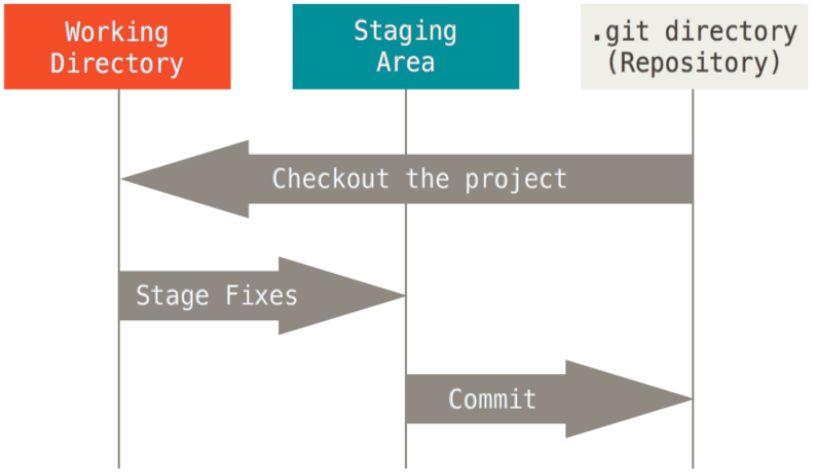
Creating Git-repository

Repository cloning

```
$ git clone [url] [folder]
$ git config credential.helper store    // creates .git-credentials and will place
                                           in it https://user:pass@example.com

# examples
$ git clone git://github.com/matthewmccullough/hellogitworld.git
$ git clone https://github.com/matthewmccullough/hellogitworld.git
$ git clone git@github.com:matthewmccullough/hellogitworld.git my_custom_folder
```

Git basics: Three states

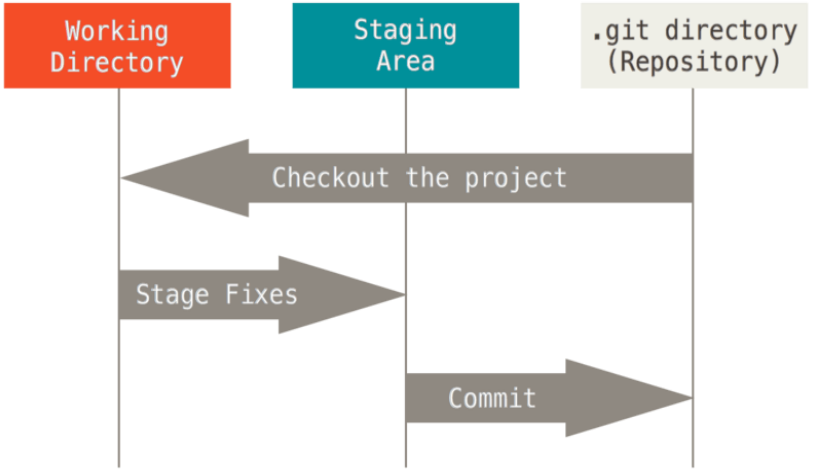
- Working directory: area of prepared files and Git directory.
- Staging area: the file located in your Git-directory, containing information regarding what changes will go into next commit. This area is also called “index”.
- Git-directory: the location, where Git stores metadata and objects base of your project, this is the part that is copied while cloning repository.



Git basics: Three states

Files in GIT can be in one of three states:

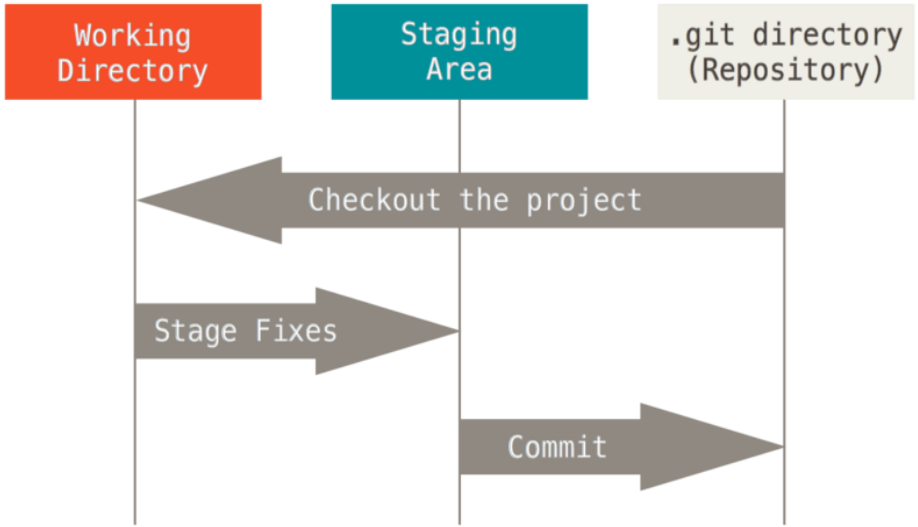
1. Modified – file is just modified
2. Staged – files are added in index
3. Committed – files are already saved in your local base



Git basics: Three states

Basic approach to workflow:

1. You modify files in your working directory.
2. You add files in index, thus adding their snapshots in staging area.
3. When you make commit, files from index are used as they are, and this snapshot is saved into your Git directory.

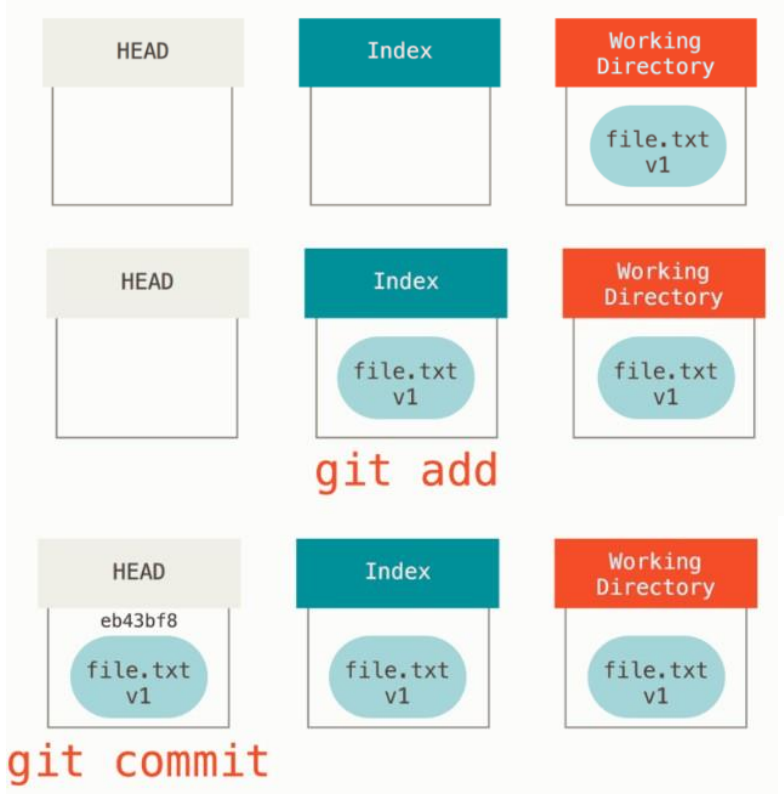


Recording changes

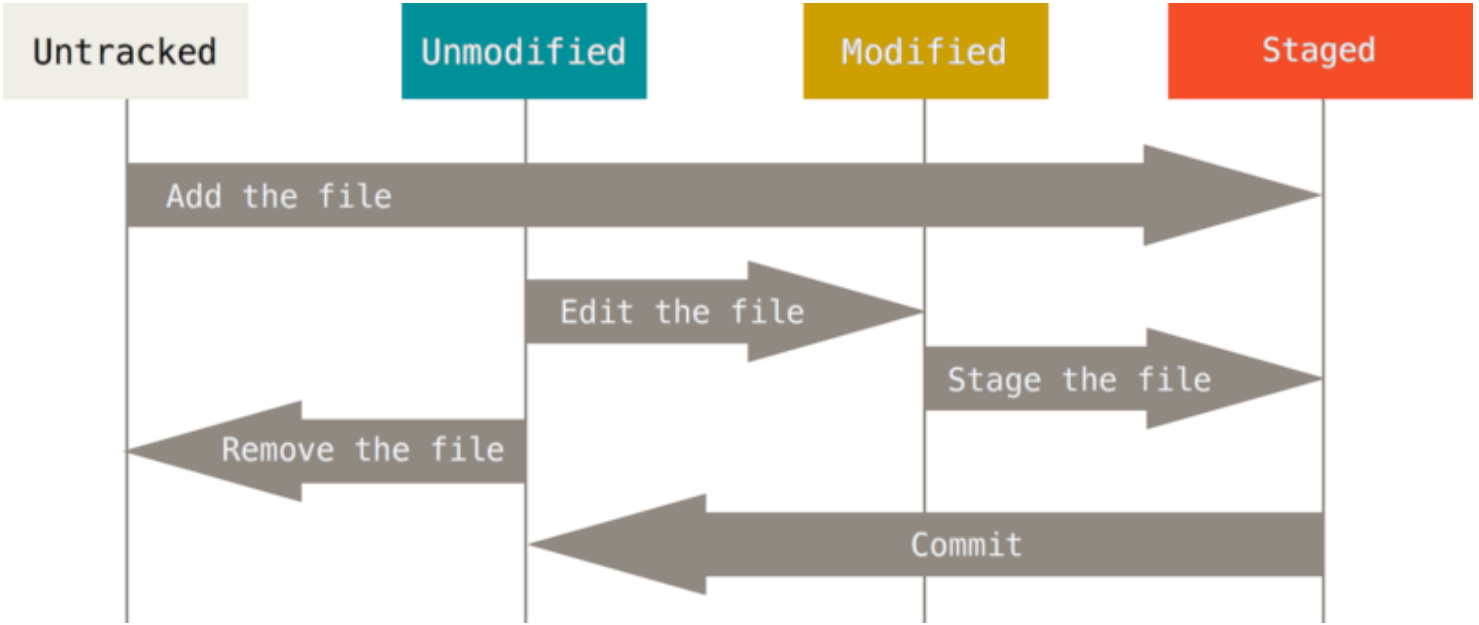
1. You modify files in your working directory.

2. You add files in index, thus adding their snapshots in staging area.

3. When you make commit, files from index are used as they are, and this snapshot is saved into your Git directory.



Recording changes into repository



GIT COMMANDS

Recording changes into repository

Tracking new files

```
$ git status
//create README.txt
$ git add .
$ git status
```

Staging modified files

```
$ git status
//update README.txt
$ git status
$ git add README.txt
$ git status
```

Removing files

```
$ git status
$ git rm README.txt
$ git rm -f README.txt
$ git rm --cached README.txt
```

Moving files

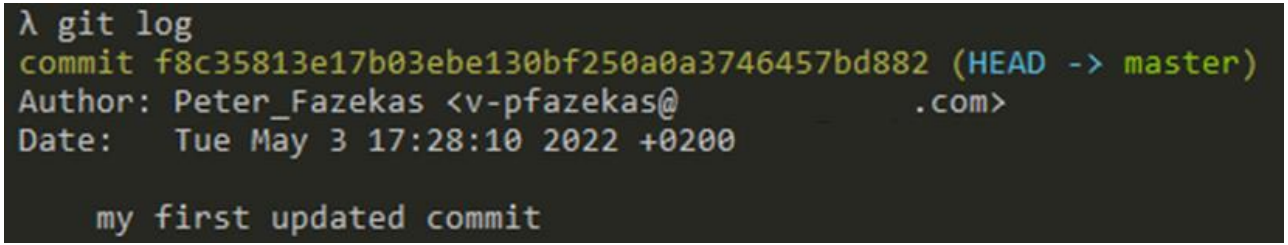
```
$ git mv README.txt README.md
```

Committing changes

```
$ git commit -m 'commit message'
```

LOG

```
# commits history view
$ git log [--oneline] [--graph] [--all] [--decorate] [-cnt_commit] [branch]
```



COMMIT

git commit: creates a new commit from the staged files.

```
# commit with comments
$ git commit -m 'comment'
```

Last commit can be changed with the –amend option.
This command brings the system default editor to change commit message.

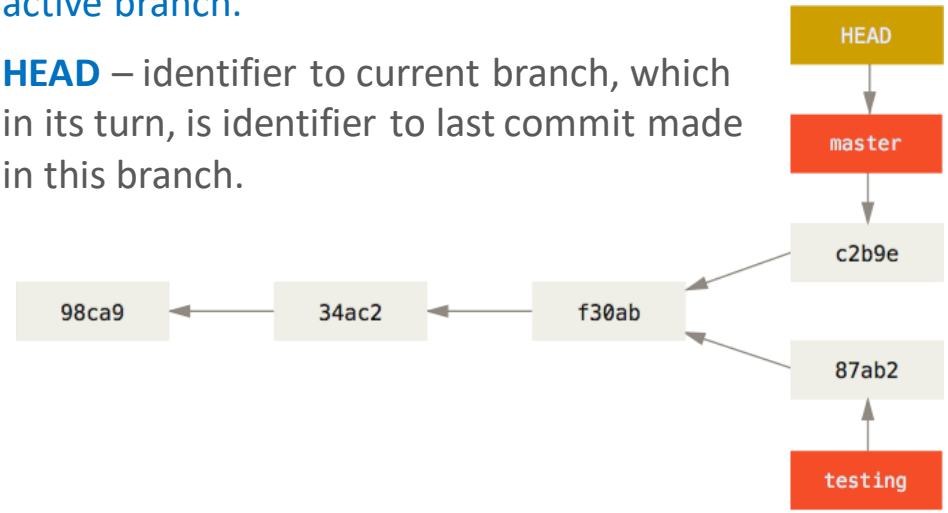
```
$ git add forgotten_file
$ git commit --amend
```

BRANCH

We use branches to make parallel work. The branch is simply a pointer to a git commit.

Git commit command adds commit to the active branch.

HEAD – identifier to current branch, which in its turn, is identifier to last commit made in this branch.




```
# view list of local branches
git branch

# view list of local and removed branches
git branch -a

# create branch
git branch bname

# switch to branch
git checkout some_branch

# create branch and switch to it
git checkout -b bname

# delete branch after merge
git branch -d bname

# delete branch
git branch -D bname
```

REMOTE – Remote repository that was not deleted

Remote repositories – are repositories that your local repository relates to, available via network. You can synchronize the repositories with each other.

Remote branches – are local branches that cannot be moved, and they act as markers for reminding where they were in the remote repository during the last login.

Upstream branch – local branch connected with remote.

REMOTE – Remote repository that was not deleted

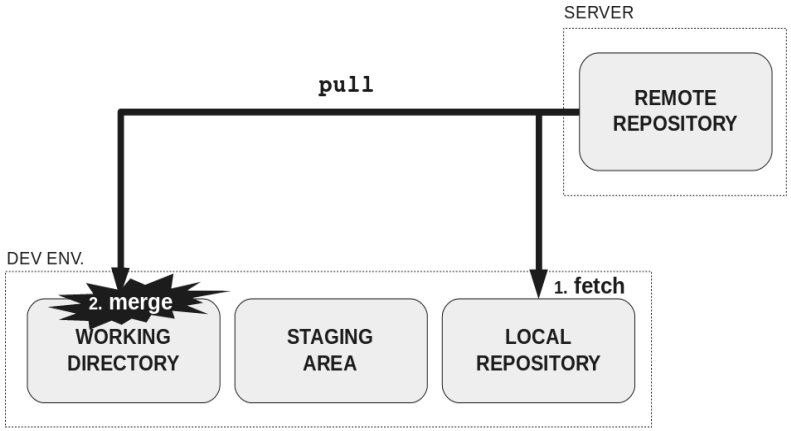
```
$ git clone https://github.com/schacon/ticgit

# View remote repositories
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)

# List of remote repositories (on server)
$ git remote show origin

# Adding remote repositories
$ git remote add origin https://github.com/schacon/ticgit

# Receive changes from remote repository
$ git fetch
$ git pull
```

REMOTE – Remote repository that was not deleted

For synchronizing of remote branches with repository command **fetch** is used, which moves remote branches, but does not change work-dir.

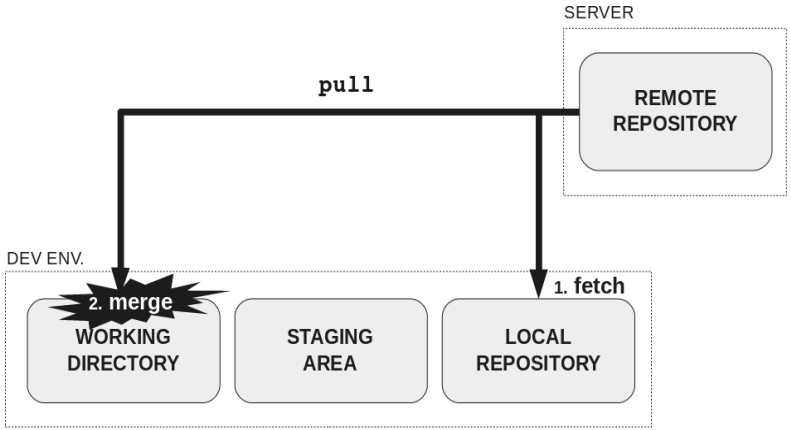
git pull automatically extracts and then gathers data from remote branch into your current one (git executes command fetch, and then attempts to make command merge)

push sends local changes to tracking remote branch in repository.

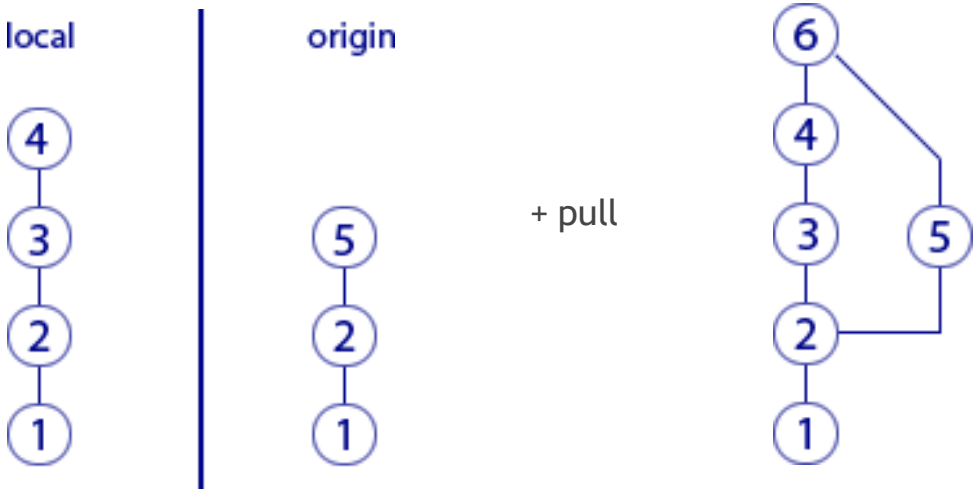


REMOTE – Remote repository that was not deleted

```
# list of tracking branches
$ git branch -v

# Download origin/dev branch, create local dev, make it tracking, change work-dir
$ git checkout --track origin/dev
$ git checkout -b dev origin/dev

# make local branch tracking
$ git branch -u origin/dev

# make local branch untracking
$ git branch -d dev origin/dev

# create remote branch
$ git push dev origin

# deleting branches in remote repository
$ git push origin --delete dev
```



origin/local

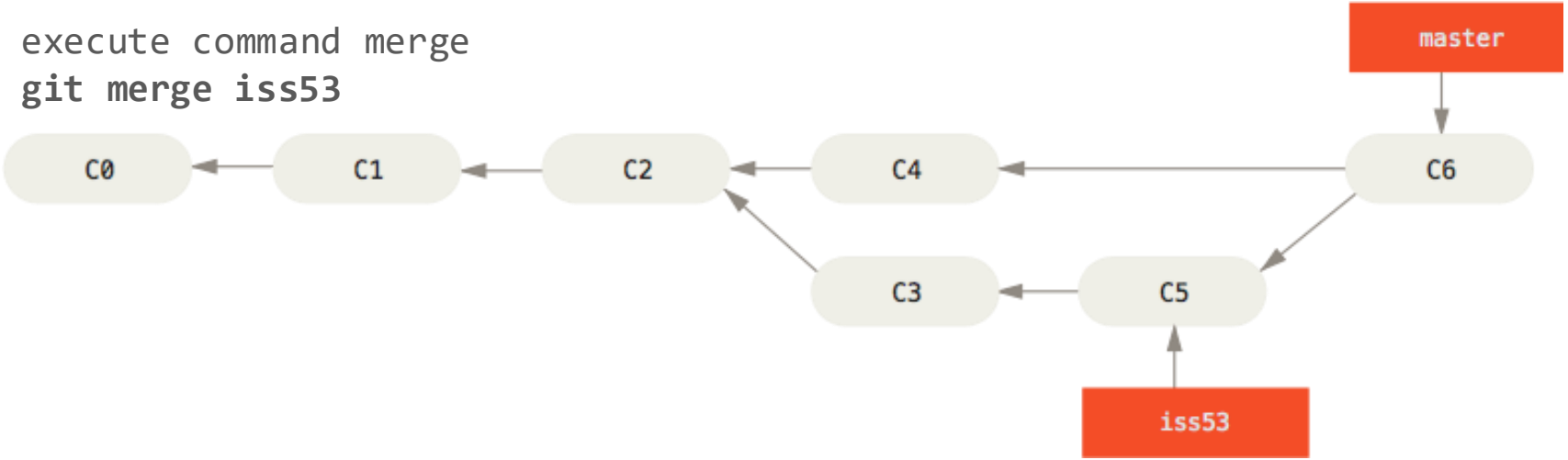


MERGE AND REBASE

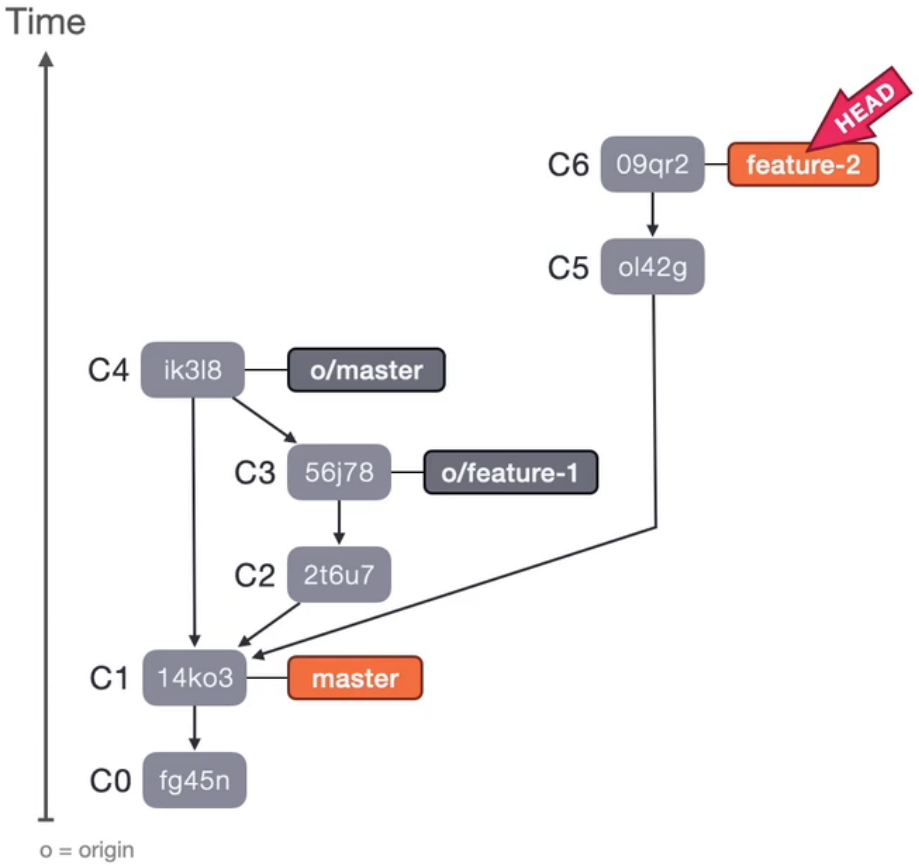
MERGE

Incorporate changes from a branch into the current branch.

```
# switch to branch, where we want to make changes
$ git checkout master

# execute command merge
$ git merge iss53
```

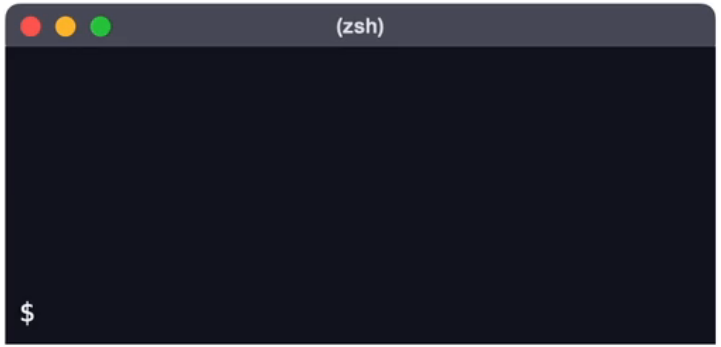


Git - merge



Joins two or more development histories together, preserving history as it happened.

Here, changes from ***master*** gets integrated into ***feature-2***.



Alexis Määttä Vinkler

MERGE CONFLICT

Conflict happens when merge two branches, and both branches contain changes of the same file(s), and GIT can not automatically resolve the conflict. User manually needs to solve the conflict.

MERGE CONFLICT

```
$ git merge iss53
Auto-merging hello.txt
CONFLICT (content): Merge conflict in
index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

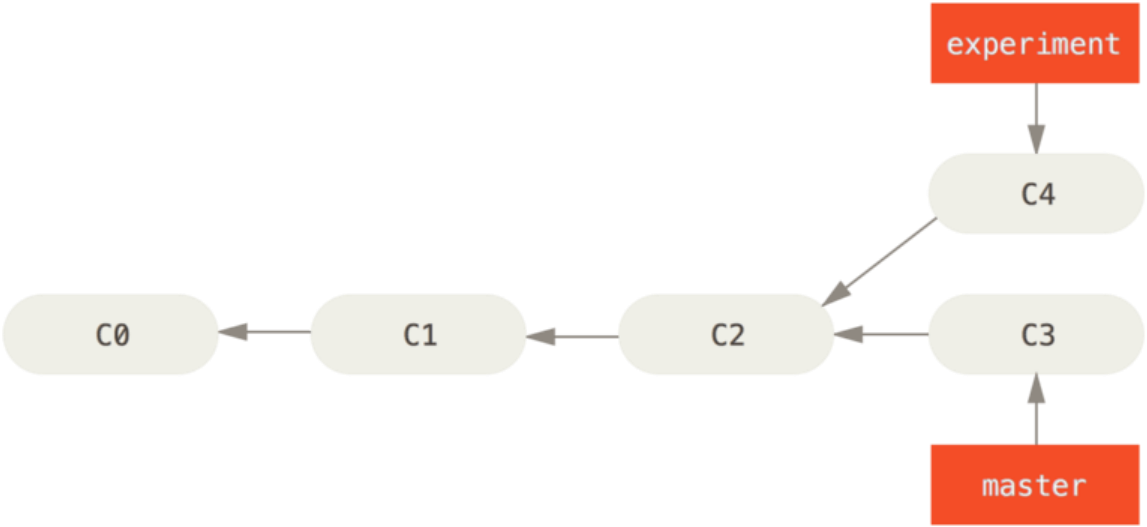
```
$ git status
On branch master
You have unmerged paths.
 (fix conflicts and run "git commit")
Unmerged paths:
 (use "git add <file>..." to mark resolution)
  both modified: hello.txt
no changes added to commit (use "git add"
and/or "git commit -a")
```

```
$ vim hello.txt
```

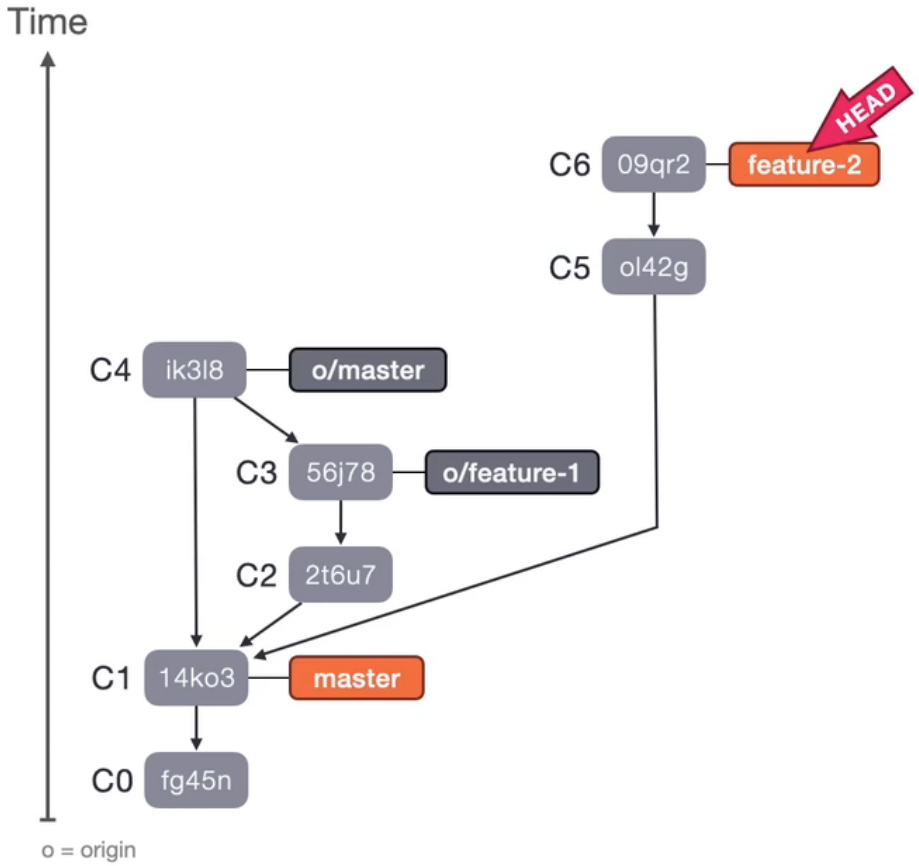
```
$ git commit –am 'merge'
```

```
<<<<<<< HEAD: hello.txt
Hello master
=======
Hello test
>>>>>>> iss53: hello.txt
```

REBASE

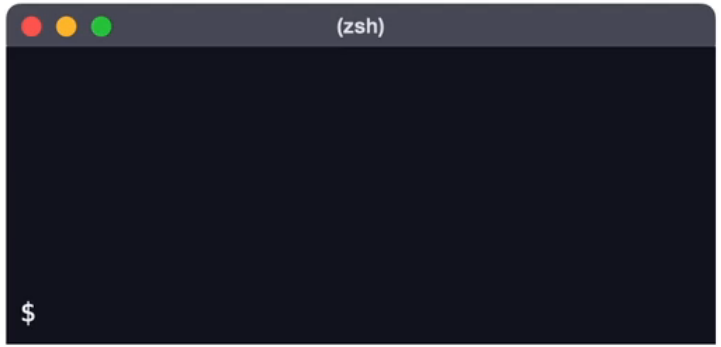


Git - rebase



Reapplies commits on top of another base branch, and rewrites history.

Here, changes from ***master*** gets integrated into ***feature-2***.

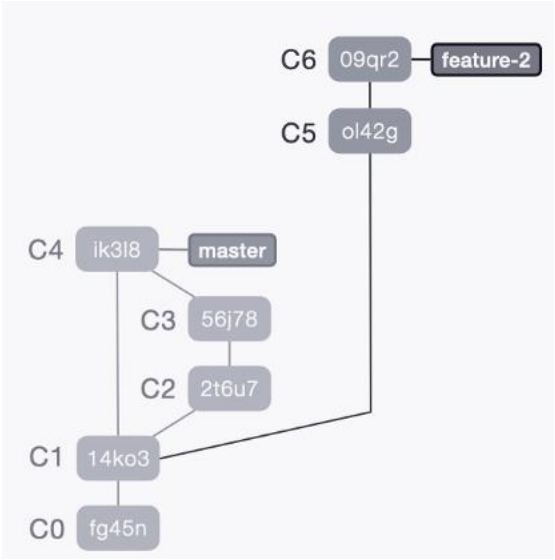


Alexis Määttä Vinkler

Merge vs Rebase

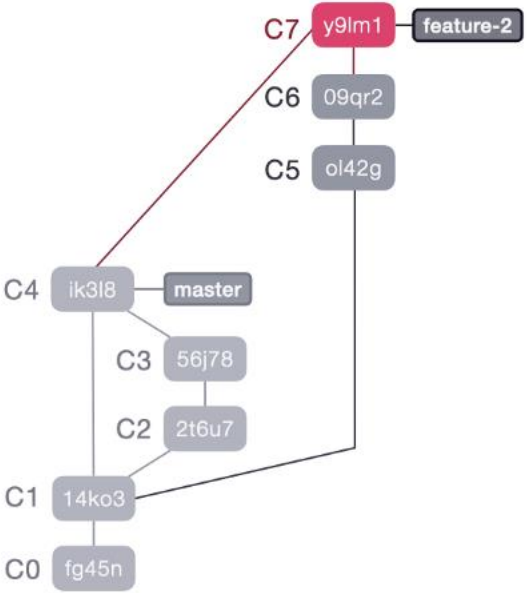
Start Case

There are two main ways to integrate changes between branches, merge and rebase.



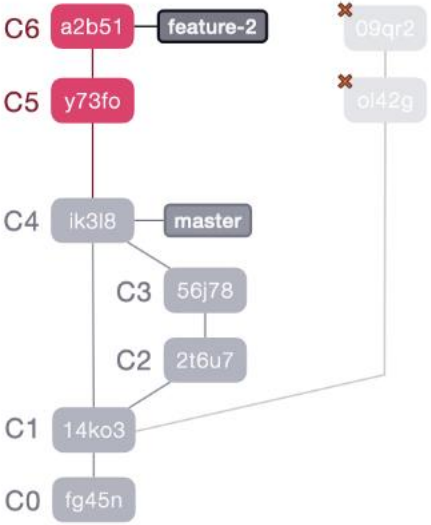
Post merge

Merge preserves history as it happened, creating only one new merge commit.

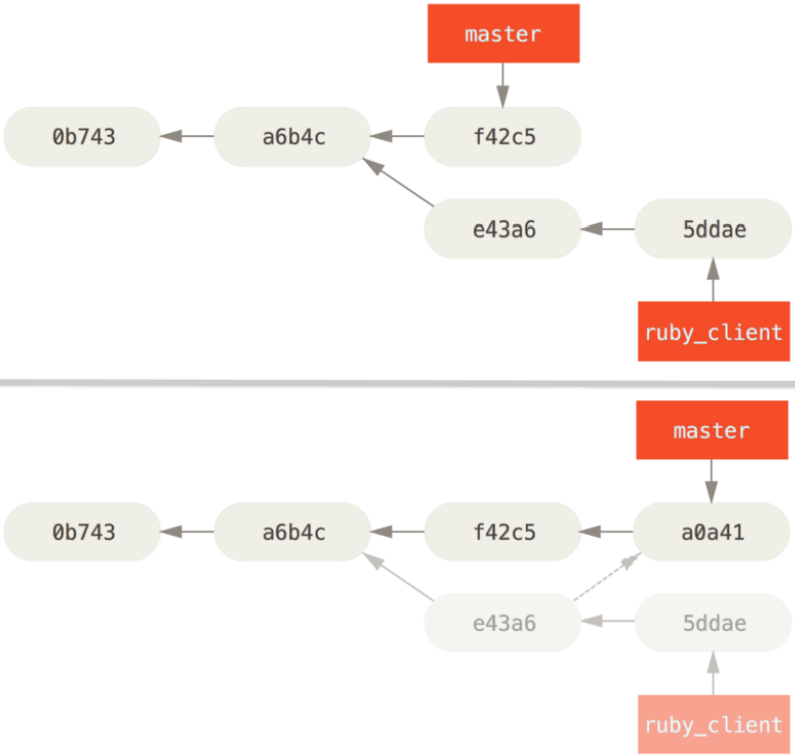


Post rebase

Rebase rewrites history, reapplying commits on top of another branch.



CHERRY-PICK

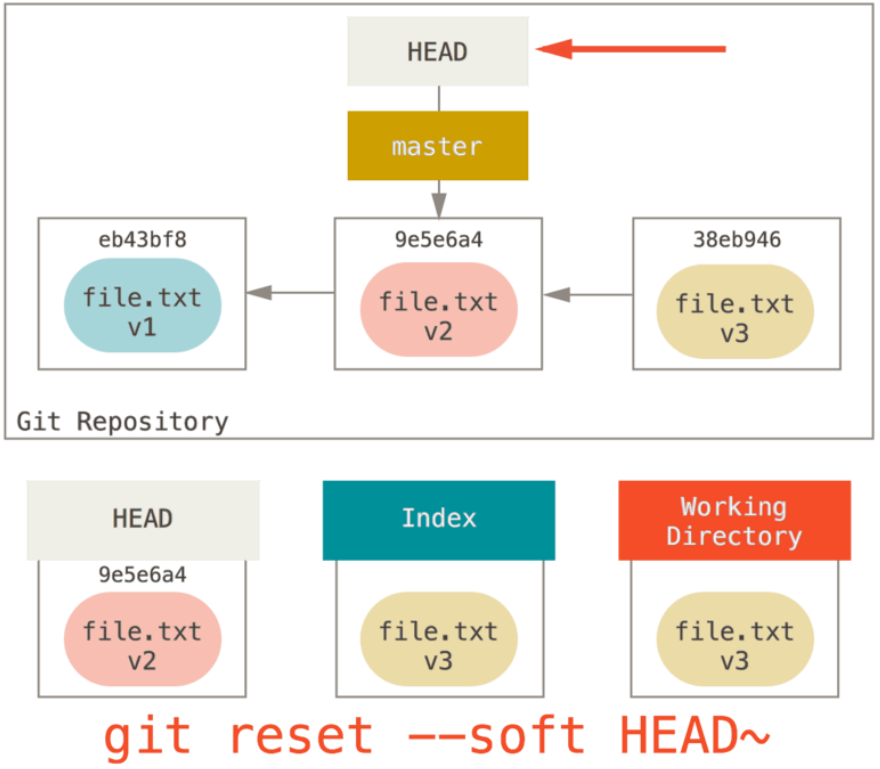


cherry-pick commits when you need to apply change(s) of a branch on another (i.e., hotfix)

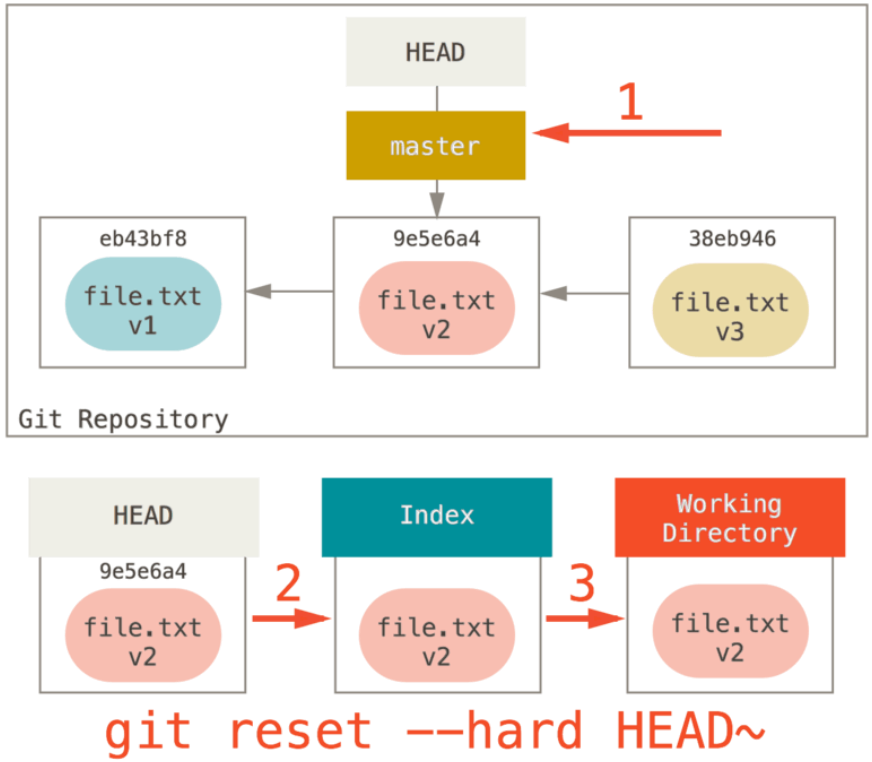
```
$ git cherry-pick <commit>…
```

RESET

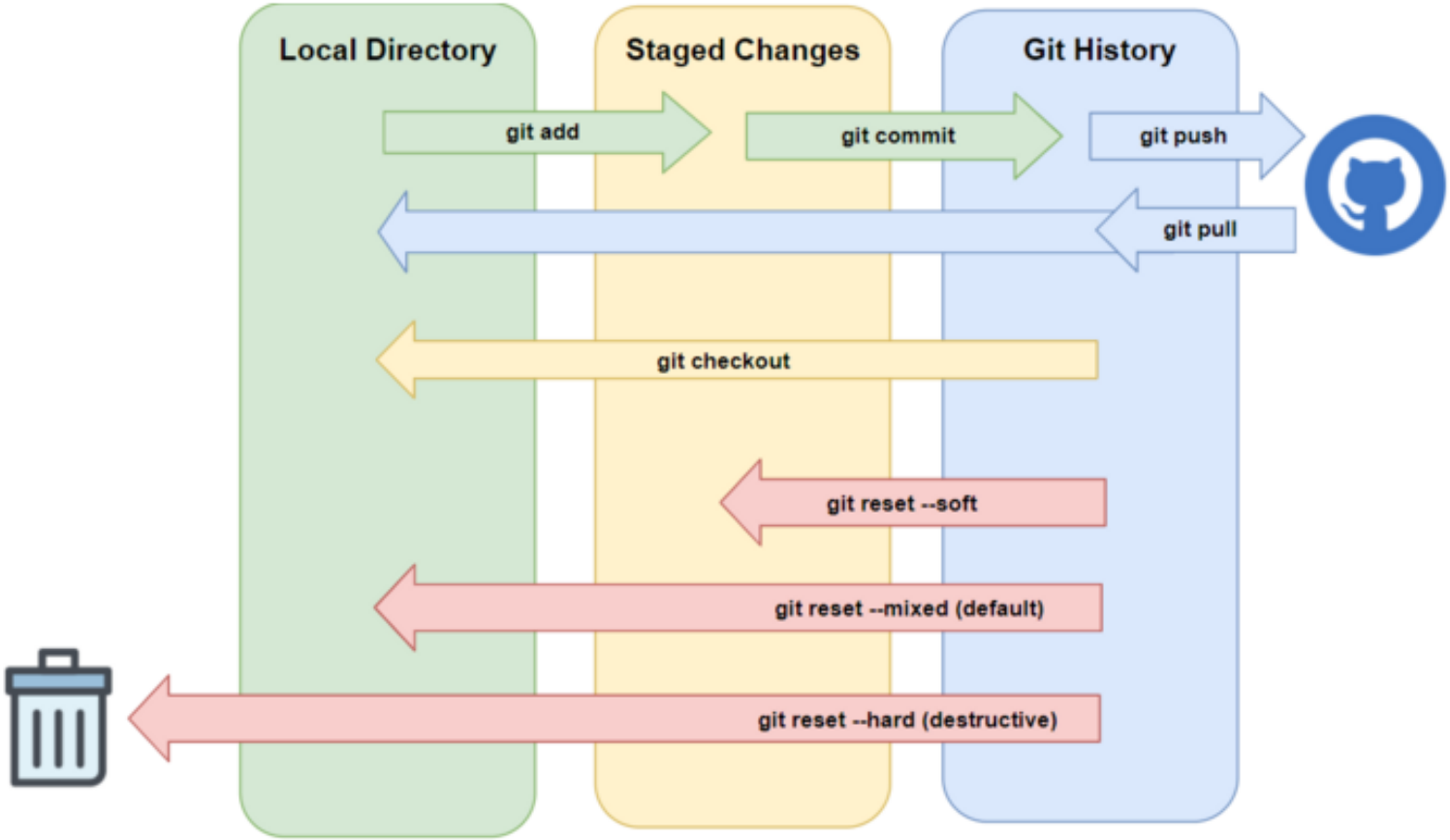
Use `git reset` to undo local changes that are already committed.



RESET

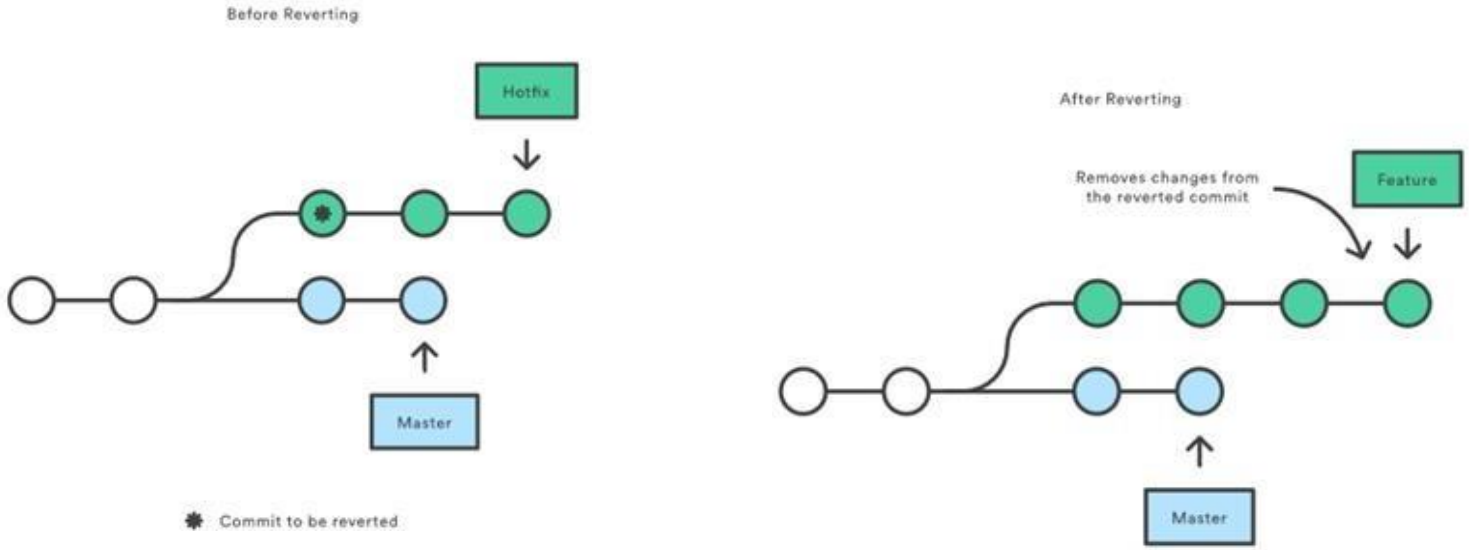

HEAD
master
1
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
9e5e6a4
file.txt
v2
2
Index
file.txt
v2
3
Working
Directory
file.txt
v2
git reset --hard HEAD~

RESET


Local Directory
Staged Changes
Git History
git add
git commit
git push
git pull
git checkout
git reset --soft
git reset --mixed (default)
git reset --hard (destructive)

REVERT

`git revert` creates a new commit to revert other commit(s).
Note: the original commit(s) remaining in the history.

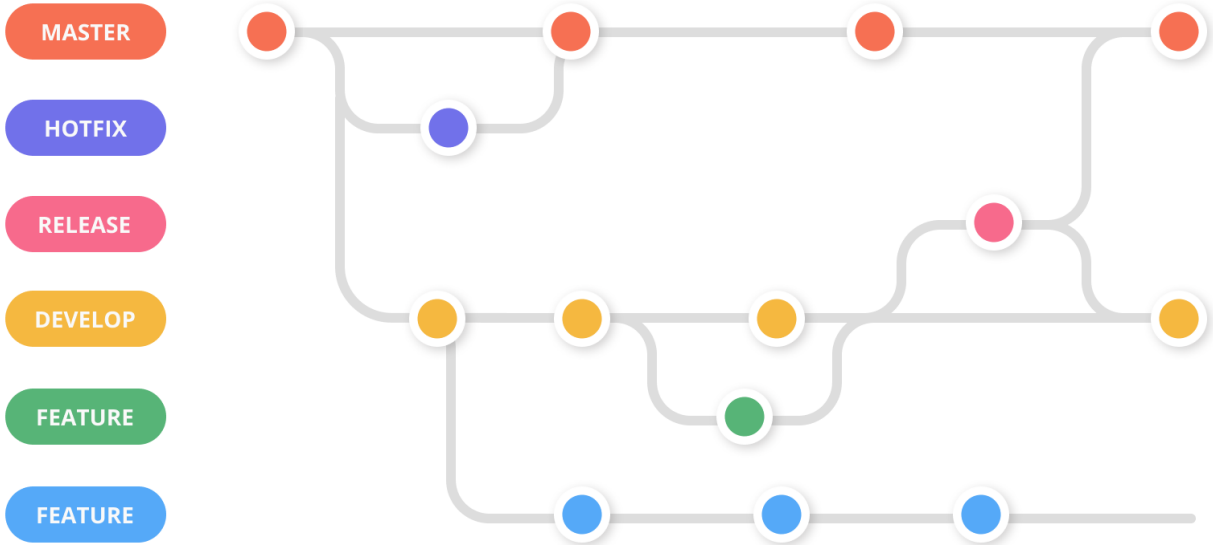



```
$ git revert HEAD~2
```

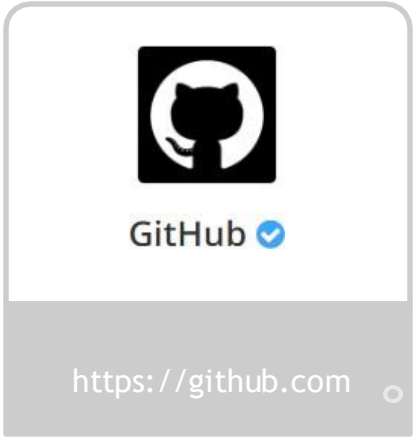
BRANCHING STRATEGIES

Branching strategies

Development teams choose a branching strategy to promote developer's changes into common, usually with well defined quality gates.



Code collaboration and version control tools



REVIEW

```
git init
git clone
git status
git add
git commit
git branch
git checkout
```

```
git merge
git rebase
git push
git fetch
git pull
git log
git remote
```

RESOURCES

Documentation:

- Reference Manual: https://git-scm.com/docs
- Git Cheat Sheet:
  - https://training.github.com/downloads/github-git-cheat-sheet/
  - https://education.github.com/git-cheat-sheet-education.pdf
- Book: Pro Git: https://git-scm.com/book/en/v2

Merge vs rebase:
https://betterprogramming.pub/differences-between-git-merge-and-rebase-and-why-you-should-care-ae41d96237b6

Branch strategy: http://nvie.com/posts/a-successful-git-branching-model/

Interactive: https://learngitbranching.js.org/

edu_hu@epam.com