

Version control with git

Florian Fink

CIS LMU

Version control

- ▶ “Version control [...] is the management of changes to documents, computer programs, large web sites, and other collections of information.” (Wikipedia)
- ▶ Track changes to files over time.
- ▶ Possibility to undo and redo changes.
- ▶ Parallel code development.
- ▶ Also for writing theses, reports, archiving results from experiments, ...
- ▶ Today: basic git concepts using the command line.

Git

- ▶ Developed in 2005 by Linus Torvalds to be used as the version control for the Linux kernel.
- ▶ Free software under the GNU GPL.
- ▶ De facto standard for version control (no more svn, csv or rcs).
- ▶ Alternative: Mercurial.
- ▶ Distributed version control
 - ▶ Every Git working directory is a full-fledged repository with its complete history and full version-tracking capabilities.
 - ▶ Completely independent of network access or a central server.
- ▶ Rapid branching and merging.
- ▶ Documentation: <https://git-scm.com/book/en/v2>

Initial configuration

Git needs a basic configuration to be operational:

```
$ git config --global user.name 'Jon Doe'
```

```
$ git config --global user.email 'jon_doe@example.com'
```

You can also configure git by editing its configuration file

~/.gitconfig:

```
[user]
```

```
    name = flo
```

```
    email = flo@cis.lmu.de
```

```
[core]
```

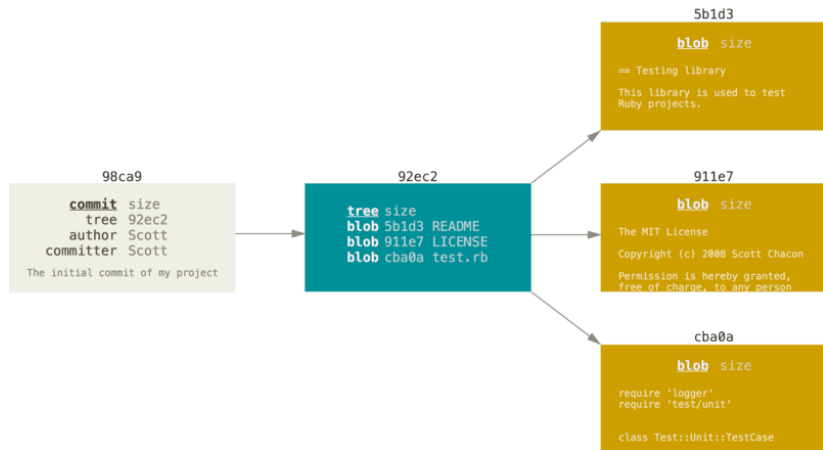
```
    editor = emacsclient -nw
```

```
[alias]
```

```
    adog = log --all --decorate --oneline --graph
```

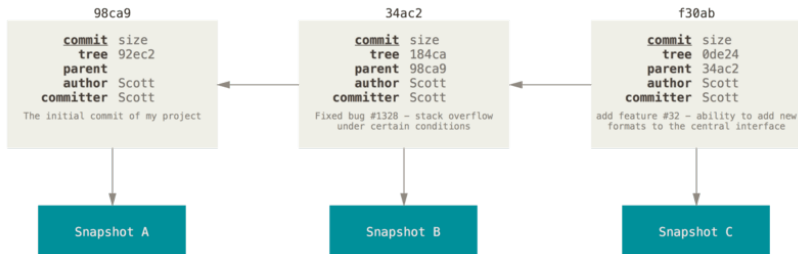
```
    dog = log --decorate --oneline --graph
```

Commits



- ▶ Commits are the basic building blocks in git.
- ▶ Each commit can be seen as a snapshot of all the files in the repository.
- ▶ Every commit has its own unique ID.

Commit chains



- ▶ Every commit (other than the first) has a pointer to its parent.
- ▶ The history of a repository is the chain of commits from the current to the oldest commit.
- ▶ The history may contains branches (i.e. different commits with the same parent).

Git log

To inspect the commit chain of a repository, use `git log`.

```
$ cd /path/to/repository
```

```
$ git log
```

```
commit 63f295ae21fa9bad1ebff4c21bb2d2e81fea9b0f
```

```
Author: flo <flo@cis.lmu.de>
```

```
Date:   Wed Nov 4 12:34:57 2020 +0100
```

```
    Implement fib function
```

```
commit aeb2a45ab1b17181ae72d8445f82bf274afd203e
```

```
Author: flo <flo@cis.lmu.de>
```

```
Date:   Wed Nov 4 12:33:07 2020 +0100
```

```
    Add license
```

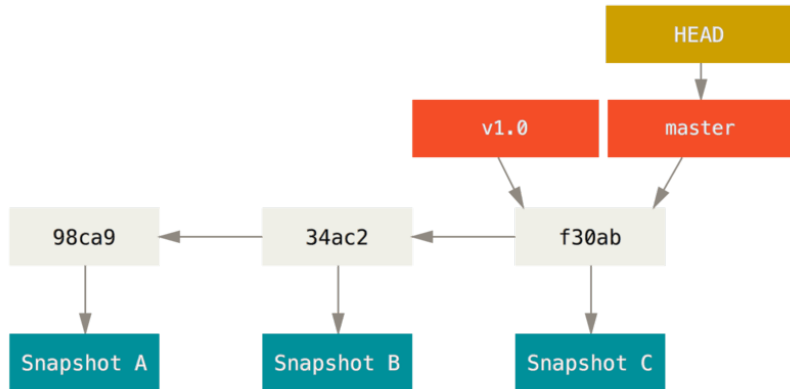
```
# ...
```

Git log options

For a better overview of the commit history of a repository there is a plethora of options to the `git log` command:

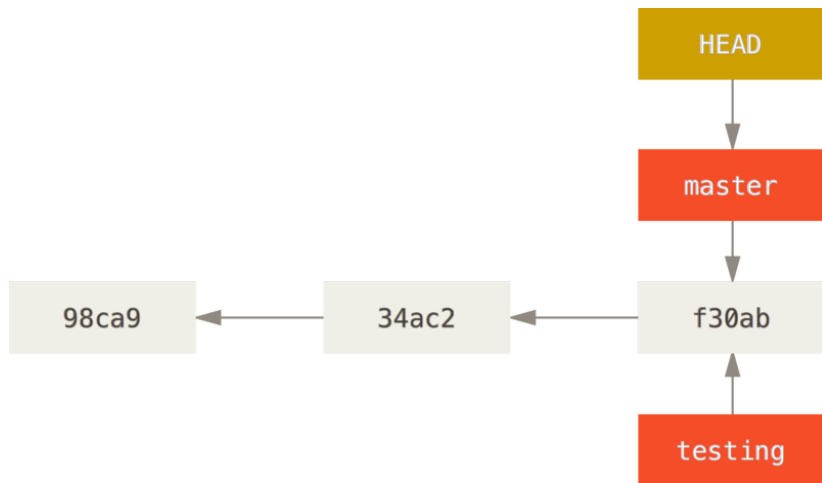
```
$ cd /path/to/repository
$ git log --decorate --oneline --graph
* 409f9f1 (HEAD -> master) Fixed stupid mistake
* 63f295a Implement fib function
* aeb2a45 Add license
* 5bb4b28 Initial commit
```


Branches



- ▶ Branches are pointers to commits.
- ▶ Git automatically creates the `master` branch on initialization.
- ▶ `HEAD` always points to the currently checked out (active) branch (it's a pointer to a pointer).

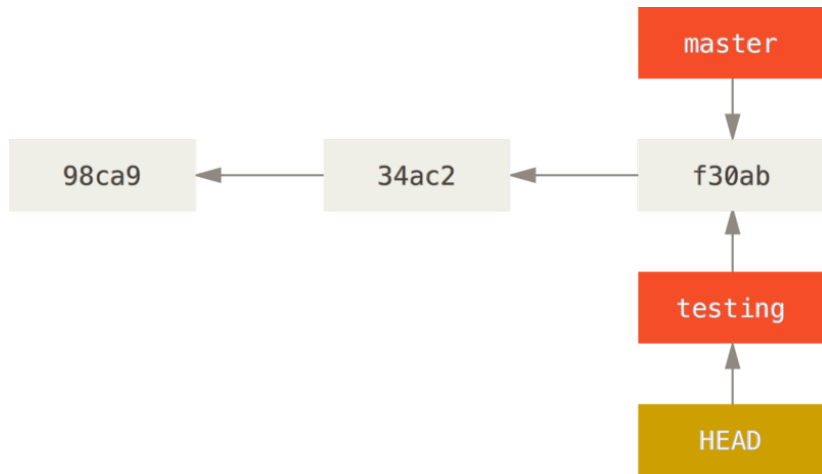
Creating branches



To create a new branch that points to the same branch where HEAD is pointing to, use `git branch`:

```
$ git branch testing
```

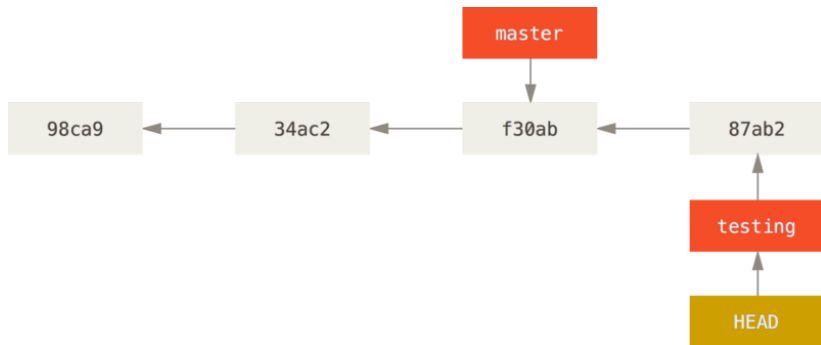
Switching branches



To switch to another branch, use `git checkout`:

```
$ git checkout testing
```

Switching branches

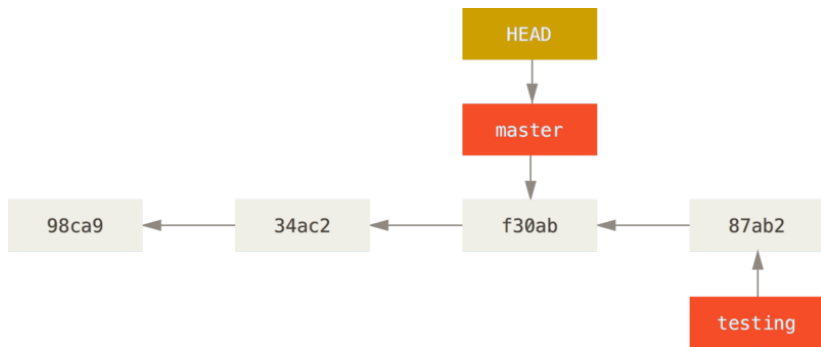


Now we do some changes

```
$ vim test.rb
```

```
$ git commit -a -m 'made some changes' # See later
```

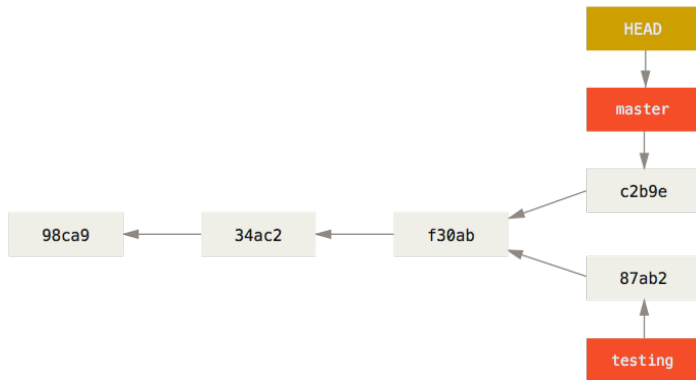
Switching branches



...checkout master again

```
$ git checkout master
```

Switching branches

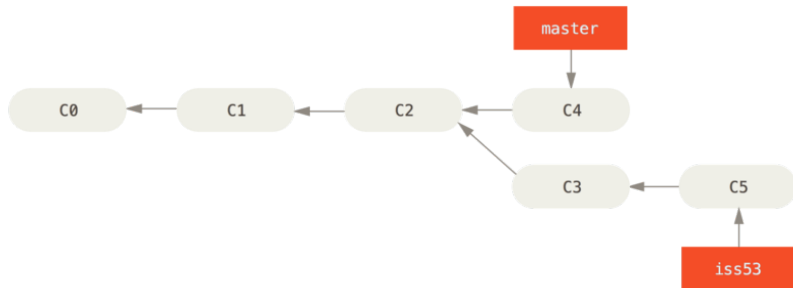


...and do some more changes

```
$ vim test.rb
```

```
$ git commit -a -m 'made other changes'
```

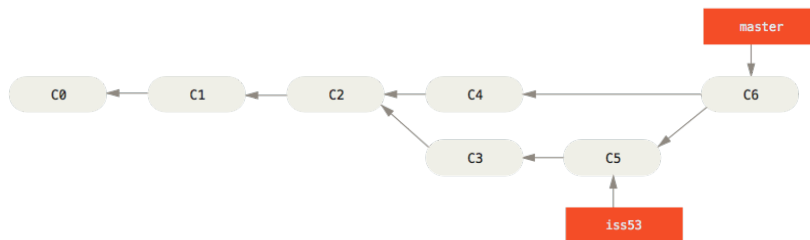
Merging



- ▶ If branches diverge they can be merged together into a branch.
- ▶ To merge another branch into the current one use:

```
$ git merge iss53 # HEAD points to master
```

Merging

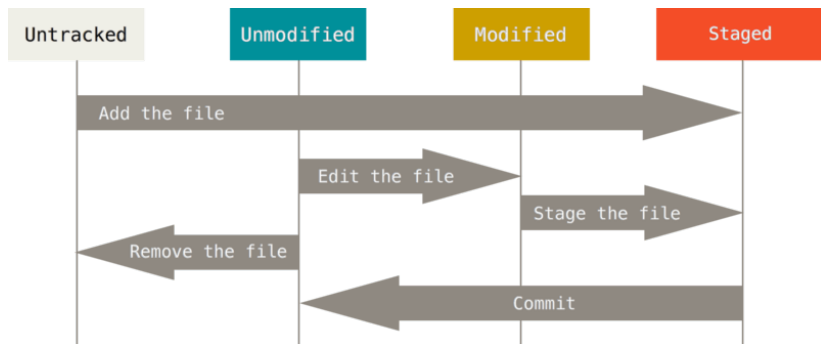


- ▶ The two branches are merged together in a new commit object.
- ▶ Git tries to merge files automatically, but sometimes it cannot resolve the resulting conflicts.
 - ▶ Conflicts need to be resolved before the merge can be committed.
 - ▶ After conflicts have been resolved the merge can be committed.

Important git commands I

command	action
<code>git config</code> <code>git log</code> <code>git branch branchname</code> <code>git checkout branchname</code> <code>git merge testing</code>	Configure git Inspect commit history Create a new branch <i>branchname</i> Checkout branch <i>branchname</i> Merge the branch <i>testing</i> into the current branch
<code>git checkout -b branchname</code>	Create and checkout branch <i>branchname</i>

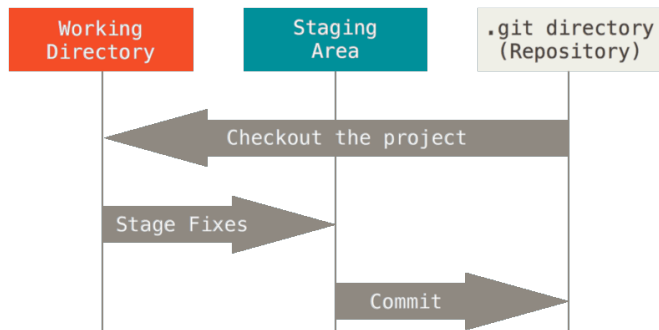
States



Files in the repository can be in the following states:

- ▶ **unmodified / committed:** data is safely stored in git's local database.
- ▶ **modified:** file is changed but not committed yet.
- ▶ **staged:** modified (or new) file is marked to get committed.
- ▶ **untracked:** file is not managed by git.

Areas



- ▶ **Working directory:** single checkout of one version of the project.
- ▶ **Staging Area:** storing files (in the .git directory) that will go into the next commit.
- ▶ **.git directory:** Contains the object database with the complete project history and all its meta-data.

Basic workflow

```
# Initialize an empty repository.  
$ cd /path/to/repository  
$ git init  
# Create and stage a new README file.  
$ vim README  
$ git add README  
$ git status # check the status of the repo.  
# Commit the staged file with a commit-message.  
$ git commit -m 'Initial commit'  
# Edit the README and stage the changes to it.  
$ vim README  
$ git add README  
$ git status # check the status of the repo.  
# Commit the staged file with a commit-message.  
$ git commit -m 'Update README'
```

Important git commands II

command	action
<code>git init</code> <code>git status</code> <code>git add file</code> <code>git commit -m 'msg'</code>	Initializes a new git repository in the current directory Check the status of the repository Add <i>file</i> to the staging area Commit staged files with a commit message <i>msg</i>
<code>git commit -am 'msg'</code>	Stage all modified files and commit them with the commit message <i>msg</i>

Git remote

- ▶ Up until now, all git commands ran locally on a machine (no internet connection needed).
- ▶ Git is a distributed versioning system, so it is possible to share git repositories online.
- ▶ One way to share git repositories is to use a hosting service:
 - ▶ IFI-Gitlab: <https://gitlab2.cip.ifi.lmu.de>
 - ▶ Github: <https://github.com>
 - ▶ Bitbucket: <https://bitbucket.org>
- ▶ Most hosting services offer git over ssh

Ssh keys









In order to use ssh, a private and public key pair is needed. If you do not have one, generate it:


```
# Generate ~/.ssh/id_rsa and ~/.ssh/id_rsa.pub  
$ ssh-keygen
```


The next step is to upload your **public** key to the hosting service. Copy the file to the clipboard and copy it into the ssh configuration form on your hosting platform:


```
$ cat ~/.ssh/id_rsa.pub | xclip
```


Gitlab add ssh-key


 **GitLab** Projects ▾ Groups ▾ More ▾  ▾ Search or jump to...      ▾  ▾


 **User Settings**


 Profile


 Account


 Applications


 Chat


 Access Tokens


 Emails


 Password

 Notifications

 **SSH Keys**

 GPG Keys

 Preferences

 Active Sessions

User Settings > **SSH Keys**

Add an SSH key

To add an SSH key you need to [generate one](#) or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/id_ed25519.pub' or '~/.ssh/id_rsa.pub' and begins with 'ssh-ed25519' or 'ssh-rsa'. Do not paste your private SSH key, as that can compromise your identity.

Typically starts with "ssh-ed25519 ..." or "ssh-rsa ..."

Title

e.g. My MacBook key

Expires at

mm / dd / yyyy

Give your individual key a title.

Add key



Remotes

- ▶ In git *remotes* are names for URLs to (bare) repositories (file paths, URLs, ...).

- ▶ You can add a remote to a repository:

```
$ git remote add myremote \
    git@gitlab2.cip/ifi.lmu.de:user/repo
```

- ▶ You can remove remotes from a repository:

```
$ git remote rm myremote
```

- ▶ You can list the remotes of a repository:

```
$ git remote -v
origin git@gitlab2.cisp.ifi.lmu.de:user/repo (fetch)
origin git@gitlab2.cisp.ifi.lmu.de:user/repo (push)
```

Cloning

- ▶ To access a remote repository, you have to *clone* it.
- ▶ You can clone a repository using ssh:

```
$ git clone git@gitlab2.cip.ifi.lmu.de:user/repo
```

or using https:

```
$ git clone https://gitlab2.cip.ifi.lmu.de/user/repo
```
- ▶ The cloned repository is put under a directory in your current path with the same name as the repository.
- ▶ Each cloned repository automatically contains a remote named *origin*.

Fetching, pulling and pushing

- ▶ Fetch all the information from mynewremote:

```
$ git fetch mynewremote
```

```
* [new branch]   master   -> mynewremote/master
```

```
* [new branch]   dev      -> mynewremote/dev
```

The local repository now contains a branch
mynewremote/master that can be merged with local branches.

- ▶ git pull: fetch and then merge into the appropriate branch
(all staged changes must be committed before the pull can happen).
- ▶ git push: push your changes to the remote (if the remote has newer changes, you need to pull the changes first).
- ▶ It is possible to specify the remote and the branch name, if the defaults (e.g. origin or master) are not appropriate:

```
$ git pull otherremote dev
```

```
$ git push otherremote othermaster
```

Typical workflow

```
# Get the current project state from the remote.
$ git clone ... # First use of the project.
$ git git pull ... # Later
# Resolve possible conflicts
$ vim conflict_file.txt
$ git add conflict_file.txt
$ git commit
# Make changes by adding and editing files
# in the repository.
$ git add ...
$ git status
$ git commit ...
# Fetch and merge changes from the remote
# (resolve possible conflicts).
$ git pull ...
# Push your local changes to the remote.
$ git push ...
```

Important git commands III

command	action
<code>git clone url</code>	Clone a repository from <i>url</i>
<code>git remote add rname url</code>	Add a new remote <i>rname</i>
<code>git remote rm rname</code>	Remove a remote <i>rname</i>
<code>git remote -v</code>	List remotes
<code>git fetch rname -v</code>	Fetch information from the remote <i>rname</i>
<code>git pull</code>	Fetch and merge changes from the remote <i>origin</i>
<code>git push</code>	Upload local changes to the remote <i>origin</i>

Git diff

- ▶ Detailed overview of changes in a file.
- ▶ Differences are shown line-by-line.
- ▶ `git diff`: what have you changed but not yet staged?
- ▶ `git diff -staged`: what changes are you about to commit?
- ▶ Example:

```
$ echo 'My project.' > README
$ git add README
$ git commit -m 'Add README'
$ echo 'More text.' >> README
$ git diff
@@ -1 +1,2 @@
    My project.
+More text.
```

Resolving conflicts

While working with git, there will occur conflicts occasionally. Git will warn you, if a conflict has occurred and you have to resolve the conflict manually.

```
$ git merge change # Merge branch change into master.  
Auto-merging README  
CONFLICT (content): merge conflict in README  
Automatic merge failed; fix conflicts and then commit ...  
$ cat README  
<<<<<< HEAD  
Hello  
=====  
Hallo  
>>>>>> change  
$ vim README  
$ git add README  
$ git commit # Commit message can be omitted.
```

Moving files

- ▶ Remove a file from the repository (it will be still in the commit history of the repository):

```
$ git rm README.txt # No one reads that.
```

```
$ git commit -m 'Remove README'
```

- ▶ Remove a file from the tracked files (without removing it from the working directory):

```
$ git rm --cached README.md
```

```
$ git commit -m 'Remove README'
```

- ▶ Move a file:

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

```
$ git commit -m 'Rename README.md -> README'
```


Undoing things

- ▶ You have already committed (but not pushed), but forgot to add a file, and/or want to amend the commit message:

```
$ git commit -m 'Initital commit'  
$ git add forgotten_file # Oops  
$ git commit --amend
```

- ▶ You want to unstage a file that you just have staged:

```
$ git add * # Oops  
$ git reset HEAD README.md
```

- ▶ Unmodifying a file. You want to revert back to the version of the file that was last committed:

```
$ echo '# end of file' > CONTRIBUTING.md # Oops  
$ git checkout CONTRIBUTING.md
```

CAREFULL: all uncommitted modifications are lost for all eternity!

Questions?