Software version control

Stefan Richter (DESY, MCnet)

Why version control?

- Combine work of multiple collaborators
- Understand changes
- Support incremental development
- Compare and revert to earlier versions
- Backup
- Parallel versions
- Document development (for other developers and yourself, not for users)

→ **version control is awesome. Use it all the time.**

What is Git?

A distributed **version control system** (VCS) whose primary user interface is the Unix command line. It basically keeps a "non-human-readable" database of the files you put under version control ("track") and provides commands to access and update that database.

Graphical user interfaces, integration in Integrated Design Environments, and **web platforms GitHub/GitLab/…** have formed around the Git core software.

The aim here is not to tell you every single Git command in existence or even to teach you all the functionality. The aim is to familiarise you with the *principles of version control*, *some good practices*, and *get you started on the practical matters*.

Practical introduction by example

We're going to walk you through an example. The things we show you here will teach you all you need to know to collaborate on your team project using Git.

Setting up

To initialise a new local repository do

```
>>> mkdir myrepo && cd myrepo
>>> git init
```

Now try `ls -a`. Notice anything interesting?

You can also clone existing repositories from a (usually remote) different location. Git supports this via http(s), ssh, etc.

We're going to create a remote repository on GitLab and clone it.

Please create an account on GitLab (http://gitlab.com) and create a public repository called `myrepo`. Then clone it to your local machine by doing

```
>>> git clone https://gitlab.com/<gitlab_username>/myrepo.git
>>> cd myrepo
```

At this point you could set some configurations.

```
>>> git config core.editor "emacs -nw"   # or your favourite light-weight editor
>>> git config color.ui true   # makes life more fun

>>> git config --list   # check that it worked!
```

To make settings for all repositories on your computer, add the flag `--global` after `git config`.

You should also set your user name and email like this:

```
>>> git config user.name "Stefan Richter"
>>> git config user.email "stefan.richter@example.com"
```

These will be associated with your commits.

Monitoring 1

Your first best friend in Git is the command `status`:

```
>>> git status
```

It shows you the files in the repository, both tracked and untracked by Git. Use this command all the time to know what's going on.

Committing

Commit = saved snapshot of tracked files. You can always revert to a commit! You can also compare them, share them, …

Commits are cheap. What do I mean by that?

Committing in Git works in two steps. First modified or untracked files are "registered" for the next commit by using `add`. This is called *staging*. The *staged* files are then committed with `commit`:

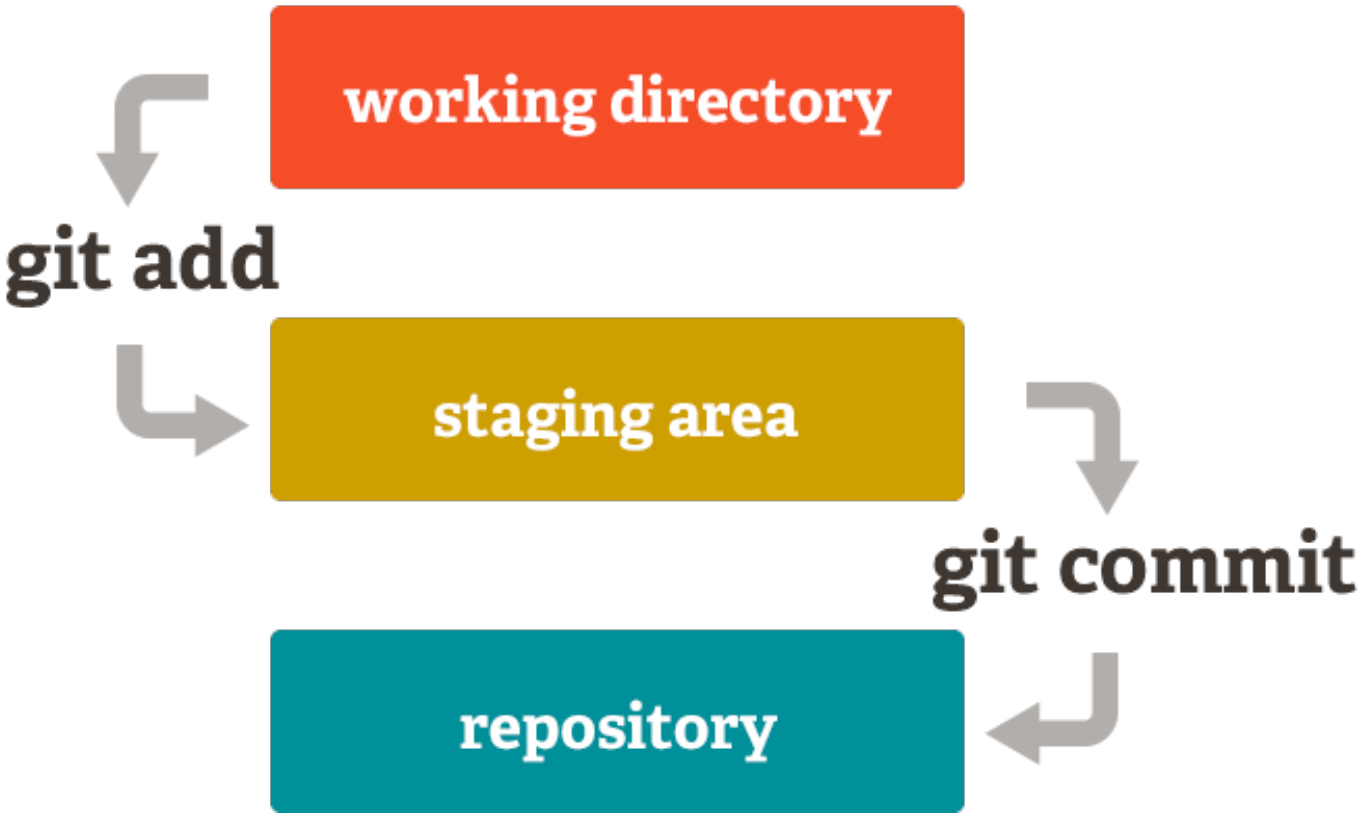

Image from https://git-scm.com (license)

Note: most other VCSs (e.g. Mercurial and SVN) don't have this two-step structure. They don't have a staging area.

```
>>> git add <path/to/file>   # file is now staged for commit
```

Note: in Mercurial and SVN, `add` is only used to put a previously untracked file under version control. In Git, it has a wider meaning!

```
>>> git commit
```

Then write a commit message. We'll give you hints for what is a good message.

Good commit messages matter! Here are some good recommendations (bedtime reading for you?).

Commits are identified by a unique hexadecimal number (a hash).

Monitoring 2

Your second best friend is `diff`. It shows you changes (differences) between versions. Without arguments, it shows all changes made to tracked files in the repository since the last commit.

```
>>> git diff
>>> git diff <path/to/file>
```

(`git diff` can also be used to show differences between arbitrary revisions. You can google it.)

Use

```
>>> git log
```

to see the commit history on your current branch. I use `git log -<number>` a lot to only show the `<number>` last commits, e.g.

```
>>> git log -3
```

A question and a suggestion:

- What happens if you track files other than flat text files?
- Create a hidden file `.gitignore` containing file patterns you want Git to ignore. These files won't show up in `git status`. E.g.

```
*.log
*.tmp
test_data/
my_personal_notes.txt
```

Branches

To check which branch you are on:

```
>>> git branch      # see where we are!
>>> git branch -a   # what's the difference?
```

Create a new branch:

```
>>> git branch dev1   # dev1 is the name of the branch
```

Switch to the branch using `checkout`:

```
>>> git checkout dev1
>>> git branch   # see where we are!
```

To merge my changes into another branch (let's say, `master`):

```
>>> git checkout master
>>> git merge dev1
```

Working with remote repositories: sharing

See what our remote is:

```
>>> git remote     # what's our remote
>>> git remote -v  # some more info
```

To update the local repository (*pull* changes):

```
>>> git pull
```

To update the remote repository (*push* changes):

```
>>> git push origin master
```

When pushing the first time, do

```
>>> git push -u origin master   # -u tells the remote to track this branch in the fu
```

A quick word on `origin` and `master`: these are the default names of the remote repository and the

first branch. They are not magical keywords and you could use different names. However, don't. Unless you have a good reason.

A common workflow that your team could adopt:

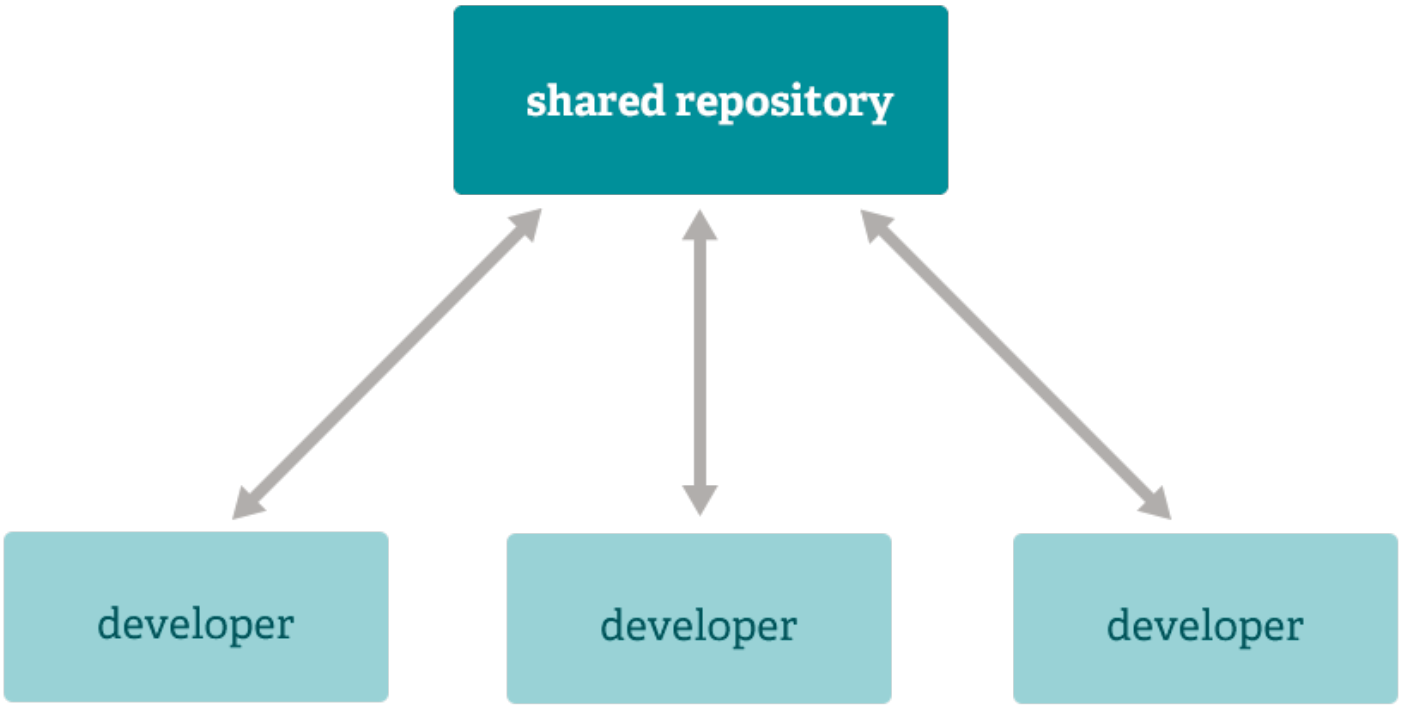

Image from https://git-scm.com (license).

Personally I like a model where every developer has one personal development branch on the shared repository, named after them (e.g. `stefan` in my case). (Everyone can have as many additional local branches as they like, but they're not tracked in the shared repository.) People push to their own branch, then request a merge into the master/release/common development/whatever branch. At this point, the others **review the code to be merged** for correctness, understandability, maintainability, style & conventions, robustness, resource effectiveness. When any necessary changes have been made/commited/pushed, the merge request is accepted.

Git's not perfect…

Git is widely used and has many powerful features, but it also has some annoying downsides. You might already have noticed that it's sometimes quite unintuitive and difficult to use…

In fact, a tutorial like this glosses over the total mess that you will from time to time end up in with Git!

Here is a good post about problematic things in Git: https://stevebennett.me/2012/02/24/10-things-i-hate-about-git. It is very instructive to read it! You will realise that sometimes it's not you who's crazy, it's Git.

There are good **alternatives** to Git: Mercurial (`hg`), which is "better", and Bazaar (`bzr`), which I know nothing about.

SVN is an older central (i.e. not distributed) VCS and not as powerful as Git and Mercurial. I don't use it voluntarily.

Finally

No, Obama! **Never ever use `git rebase` (or `git cherry-pick`)!** It rewrites repository history and can even create loss of commited information. Here's somebody who disagrees with me.