Git workshop

By Dovydas Musulas, 2022

Structure of the presentation

1. Version control systems, it’s history, emergence of git
2. Git design, Δ-based systems vs snapshots
3. Introduction to git (commands and more); practice – let’s *git* going!
4. Conventional commits and best practices

Version control systems

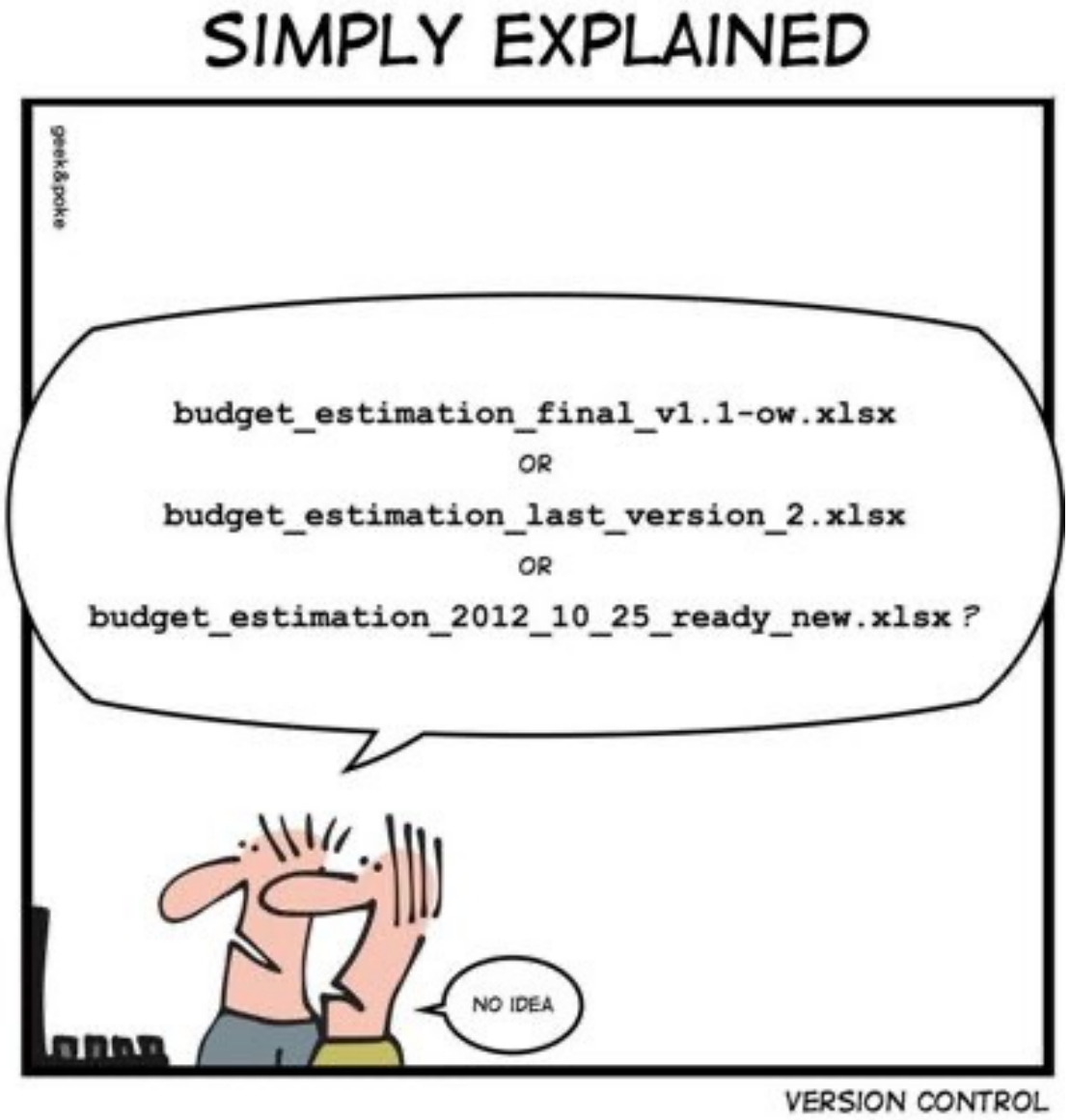
- **Version control**, also known as source control, is the practice of tracking and managing changes to software code.

- **Version control systems (VCS)** are software tools that help software teams manage changes to source code* over time.

* - not only source code but *almost* any kind of text document (docs, csv, etc.)

Why version control at all?

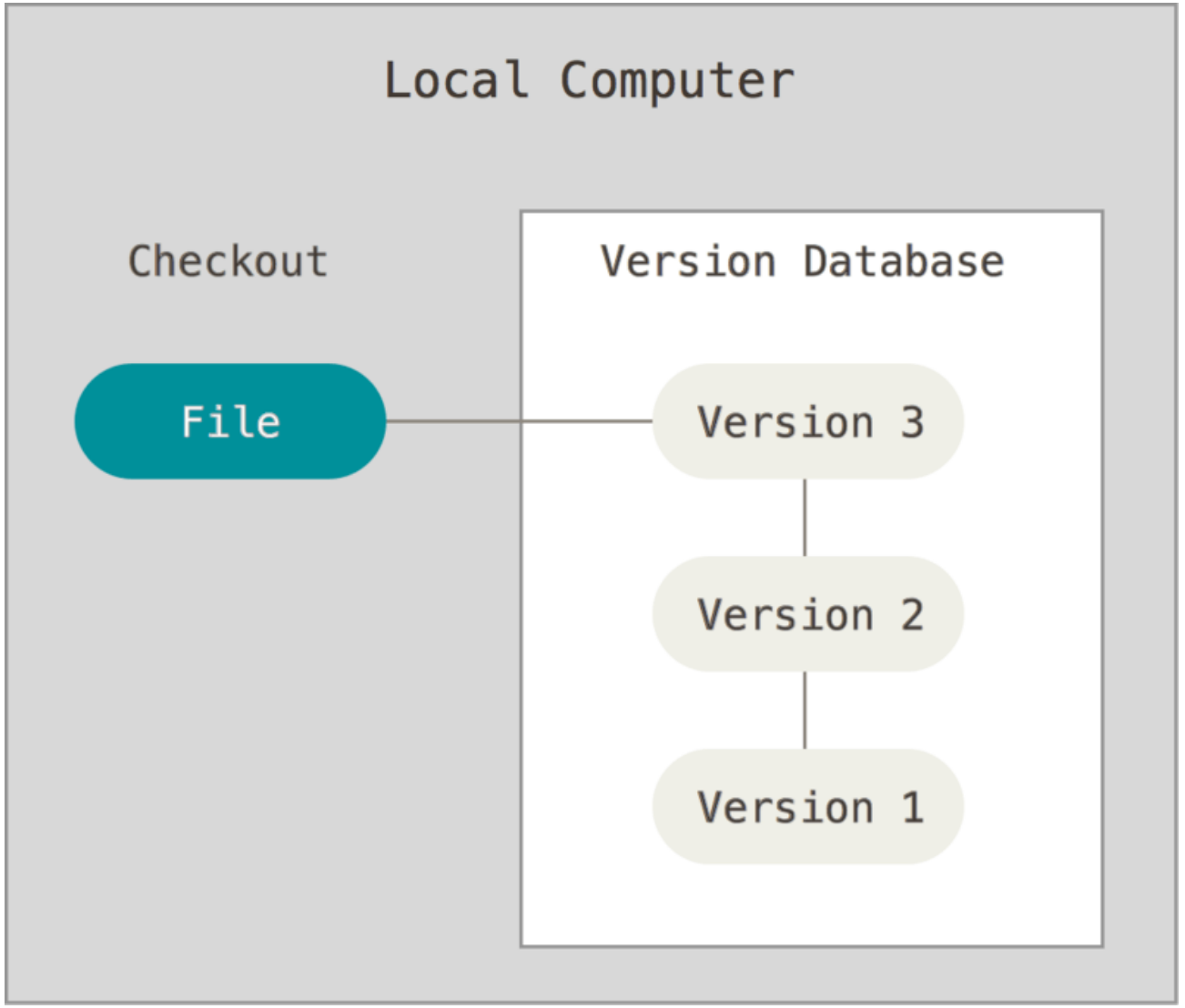
- Keeps track of changes in project (long-term history)
- Easier to collaborate, branch and merge
- Ability to CTRL+Z **(undo) anything.**



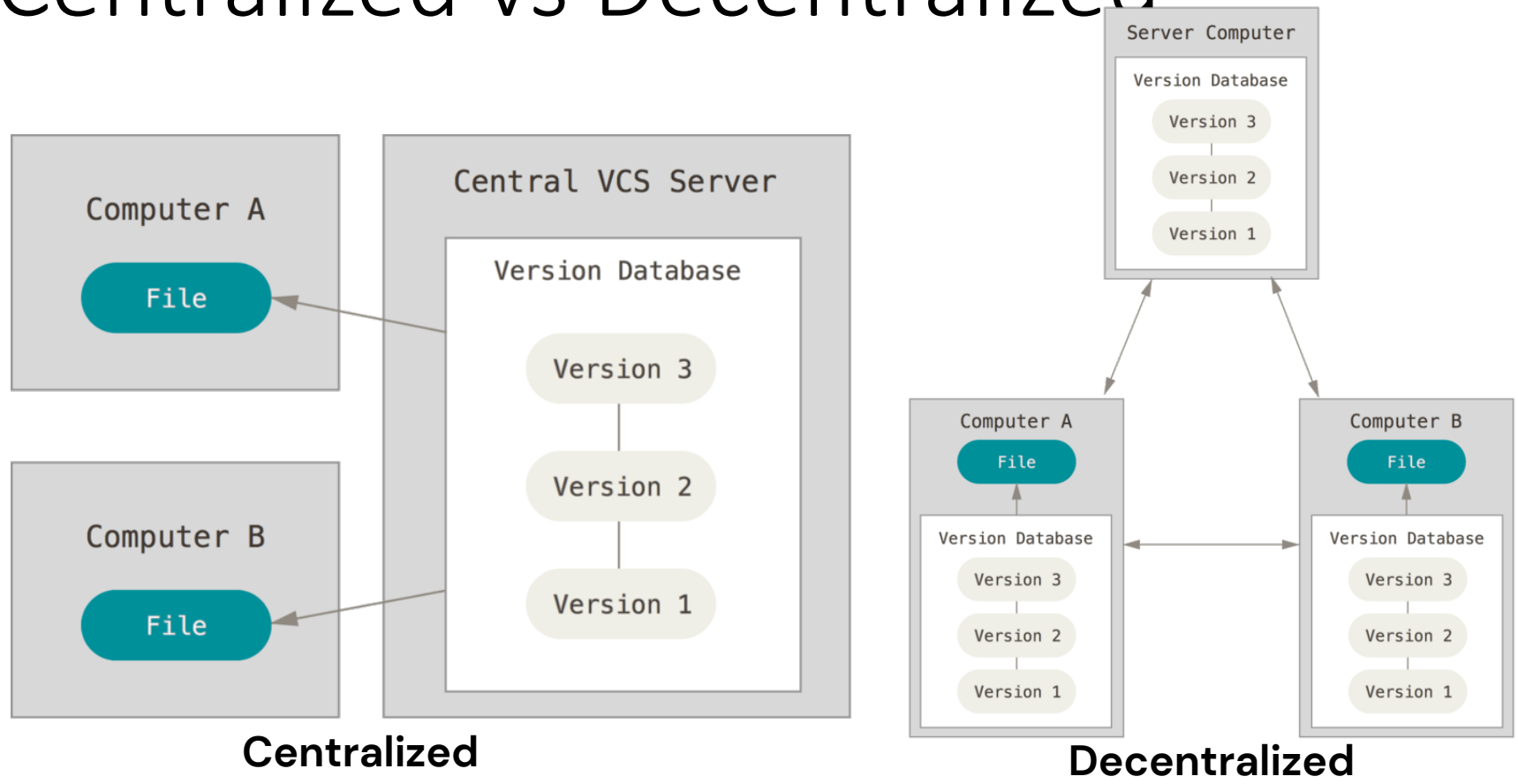
First emergence of VCS - Local

- Stores patch sets (changes between files) in version database.

- Patches successfully track how individual files evolved over time.



Centralized vs Decentralized



Centralized vs Decentralized - notes

- That is how centralized VSC emerged.
- Now instead of having patches (changes) locally they were stored in central server. This has many advantages such as transperancy of what is everyone doing in the project, easier to administrate. However problem with centralized approach is that everything is stored at one place (monolithic), merging is also quite burdersome.
- Decentralized version control system addresses this problem. Every single clone of the codebase is equally important. In case, server goes development can still progress. Git is the most popular DVCS.

Git history and characteristics

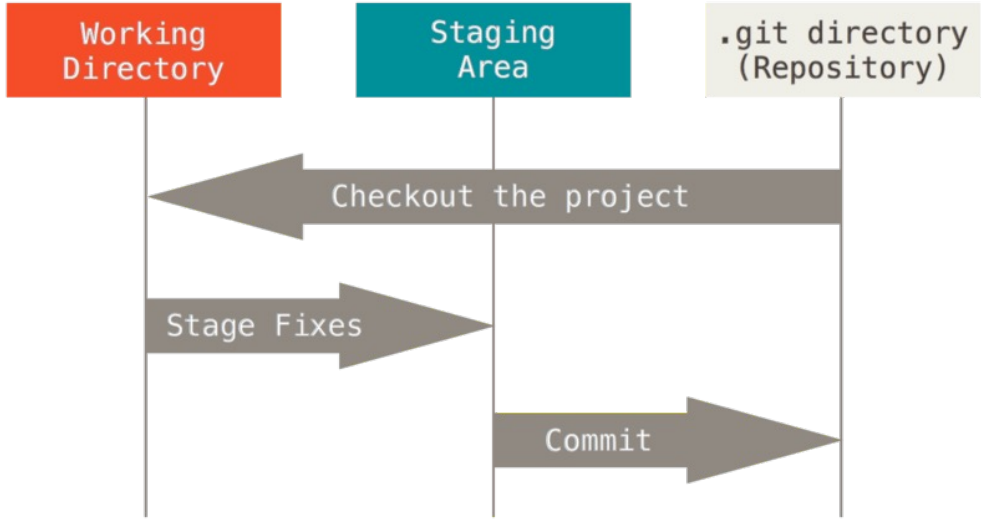
- Developed by **Linus Torvalds** in 2005 to host Linux kernel.
- Fully **distributed**, **fast**, **simple**, and supports non-linear development.
- Utilise snapshots rather than file changes.
- Has **integrity**, utilises checksums so if something is changed in the file git will know about it.

Git history and characteristics - notes

- Can host linux kernel so it was developed to handle extremely large repositories.
- Snapshots mean that rather than tracking file differences, git takes a snapshot of repository every time it is commited. Therefore, it makes easy to revert back to previous version of whole repository, and makes this system truly distributed, i.e. every single clone of the repository is equal.

Three file states

- **Modified** means that you have changed the file but have not committed it to your database yet.
- **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed** means that the data is safely stored in your local database.



Introduction to git

Config

Command	**Explanation**
git config --global user.name “<name>”	Change author name for commits (REQUIRED)
git config --global user.email “<email>”	Change author email for commit (REQUIRED)
git config --system core.editor “<editor>”	Change default text editor used by git (default = vim)
git config --list	List your configuration

Initializing / cloning repository

Git can be initialized in repository using simple command:

```
git init [repo_name]
```

Alternatively, existing repository can be cloned from remote source with:

```
git clone <url> [repo_name]
```

Your turn!

- Open your favourite terminal
- Create an empty repository anywhere in your computer

Staging

After doing the necessary changes, you have to stage your files with the following command:

```
git add <file/directory>
```

In case you no longer want to track file/folder use reset:

```
git reset <file/directory>
```

You can check status of your changes anytime with:

```
git status
```

To check what has changed in unstagged files you should use:

```
git diff
```

Committing

When the changes are staged, they are only tracked by git, but you have to COMMIT to them (this will open your default code editor for message):

```
git commit
```

This is shorthand to commit without opening editor:

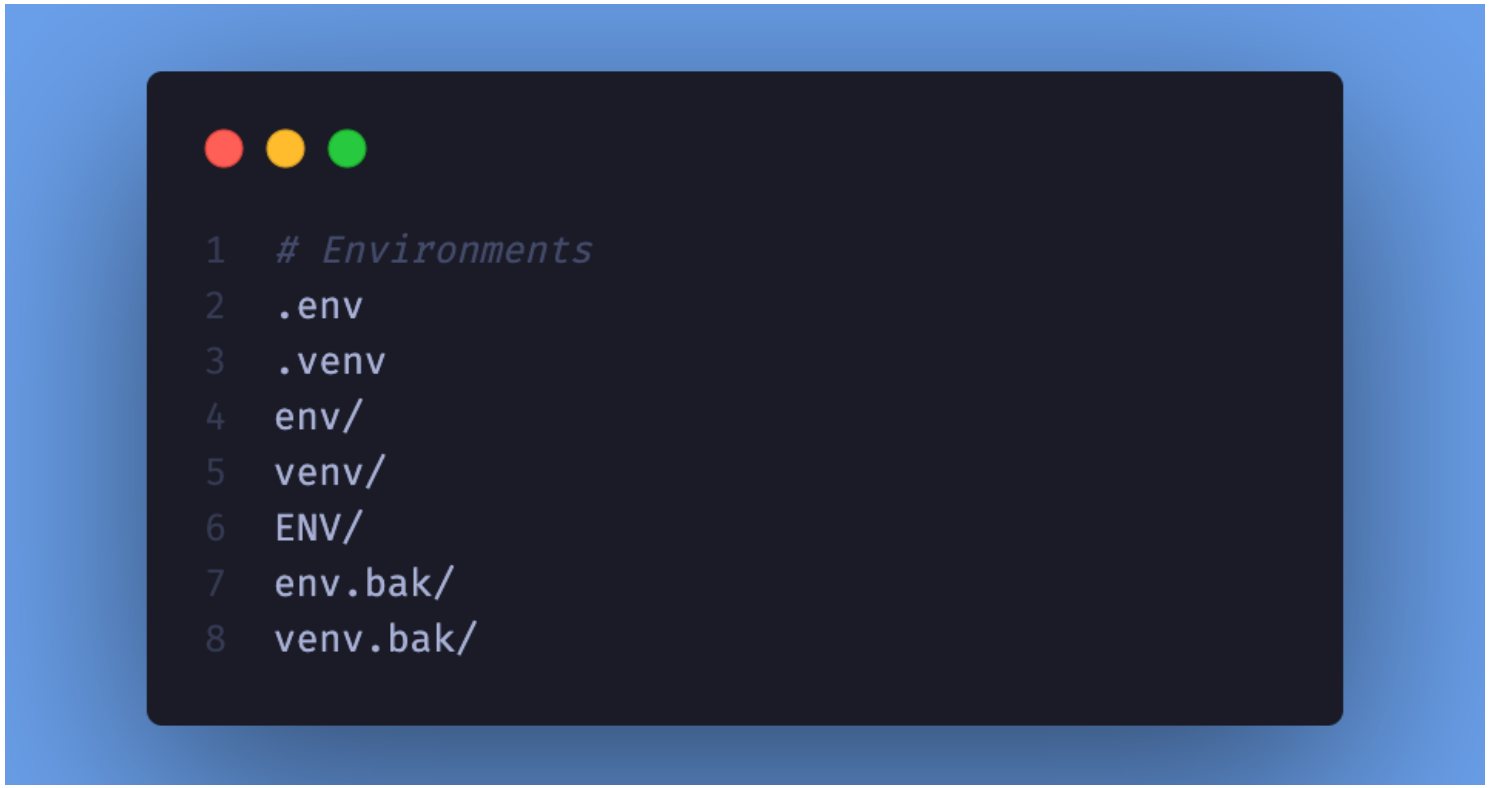
```
git commit -m <your message>
```

Check branch commit history with the following command:

```
git log
```

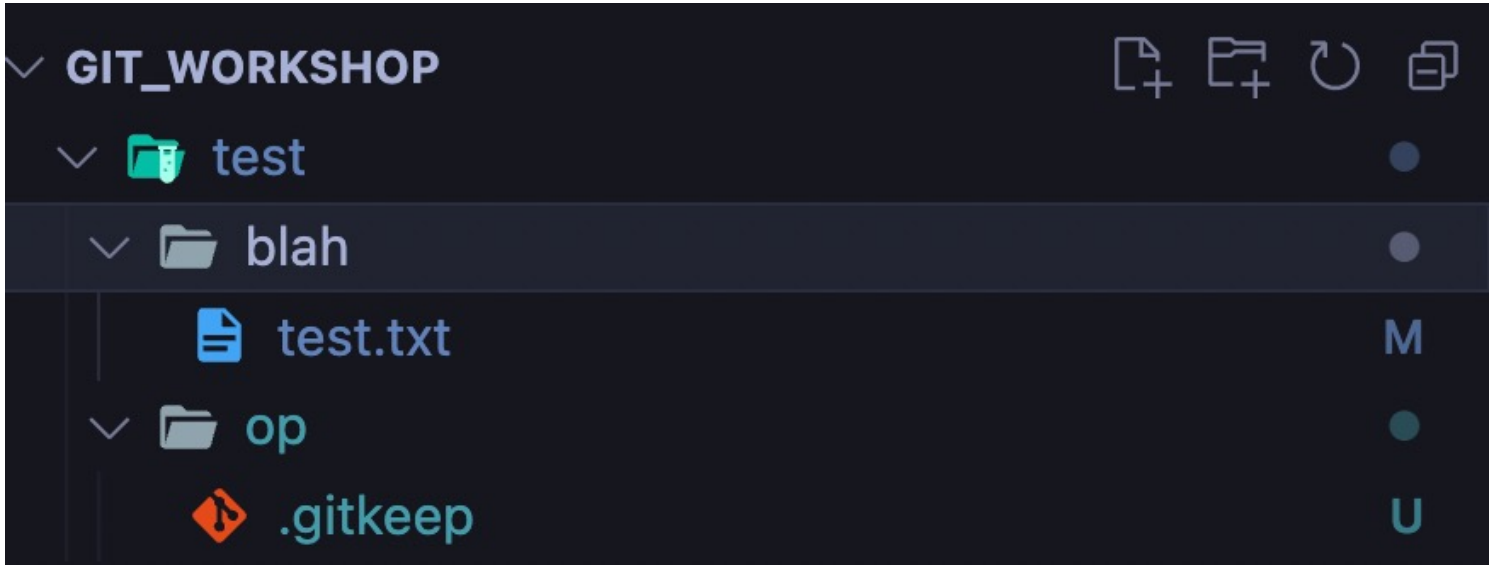
.gitignore

By default, git will keep track of all files on the repository. However, some of the files are not necessary to track and push to remotes (venvs, caches, dotfiles, logs). For this you can utilise .gitignore file.



.gitkeep

Git will not track empty directories, if you want it to do so create a placeholder file .gitkeep inside directory.



Your turn!

- Create a new file in directory
- Stage it
- Commit
- Create another file but do not track it
- Modify previous file
- Commit

Undoing mistakes

Whenever an “oopsie” happens and you want to restore from previous commit you can use:

```
git restore --source <commit_hash/ref> file
```

There is also **reset** which goes back in time to previous commit (discard newer ones).

And **revert** which creates new commit that undoes previous one.

- This is one of the ways how to undo mistakes there quite a lot of approaches that are suitable for different situation.
- BE VERY CAREFUL WITH THIS COMMAND! IT WILL OVERWRITE YOUR CHANGES

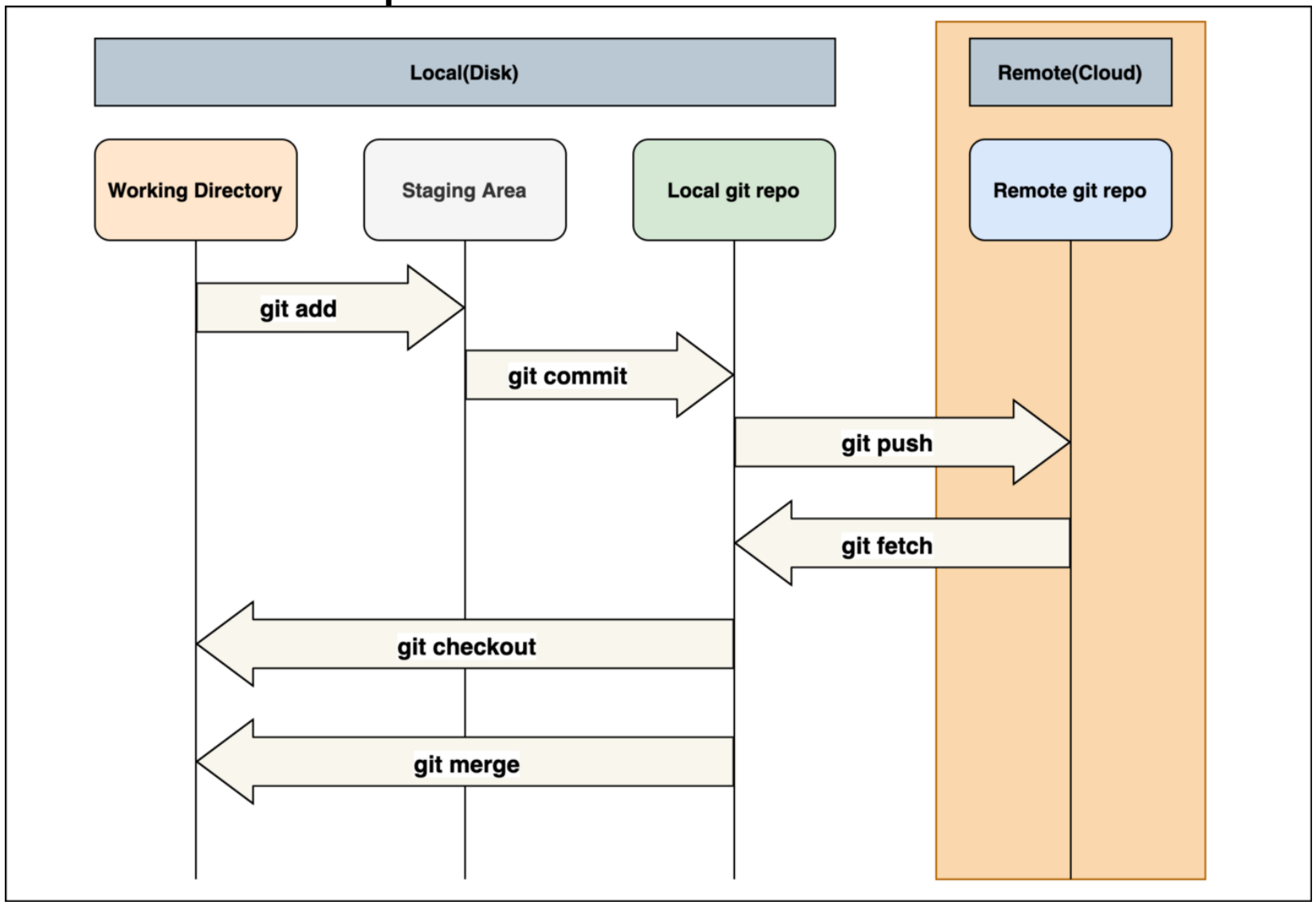
Your turn!

- Restore file from the first commit
- Modify it again
- Commit

Working with remote repositories

For me, the easiest workflow follows these simple steps:

1. Establish connection with remote sever (ssh, https)
2. Clone/pull the repository
3. Checkout a new branch
4. Commit your changes
5. Push them to remote
6. Open a merge request



Branches

Branching allow you to work on new features without affecting MASTER branch.

Merging means updating branch A source with branch B.

To create a new branch, you should use:

```
git branch <new branch>
```

Now you have to switch to the new branch:

```
git checkout <new branch>
```

To merge two branches use:

```
git merge <diverging branch>
```

- Branching means you diverge from the main line of development and continue to do work without messing with that main line
- Git encourages workflows that branch and merge often, even multiple times in a day
- You can have as many branches as you want. For instance, you are working on one issue but you have to do some more important fixes. Instead of rushing to finish first issue you can create new branch and do work there and later return to the issue.

Your turn!

- Checkout a new branch
- Alter file
- Commit
- Merge it back

Git conventions

Conventional commits

The basic structure for commits should be:

```
<type>[optional scope]: <description>
```

Type prefix should represent what is your changes trying to accomplish:

- feat – something is added to the code, new dimension, measure, explore etc.
- fix – what is broken has been fixed
- refactor – making code simpler without altering key functionalities
- style – cosmetic changes of code to make it simpler to read, pass linter tests etc.

Scope is what area of the codebase is affected (product, function)
Description – short summary of your changes.

How often should you commit?

Short answer – often!

Long answer - whenever you have a meaningful change which does not break the codebase.

- Added dimensions? – commit!
- Fixed typo? – commit!
- Created a view? - commit

You get it…

Sources for further understanding

- Git pro book by Scott Chacon and Ben Straub, https://git-scm.com/book/en/v2
- Learngitbranching.js, interactive game to better understand branches, pointers – https://learngitbranching.js.org/
- Conventional commits https://www.conventionalcommits.org/
- .gitignore templates https://github.com/github/gitignore