git - Part I
Introduction & Local Repositories

Stefan Pranger

stefan.pranger@student.tugraz.at

October 27, 2019

Overview

What is git?

Quoting Wikipedia:

git is a *distributed version-control system* for tracking changes in source code during software development. ... goals include speed, *data integrity*, and support for *distributed*, *non-linear workflows*.

Examples of distributed version-control systems: *CVS*, *BitKeeper*, *SVN*, *git*, ...

What is git?

Quoting Wikipedia:

git is a *distributed version-control system* for tracking changes in source code during software development.
... goals include speed, *data integrity*, and support for *distributed*, *non-linear workflows*.

Examples of distributed version-control systems: *CVS*, *BitKeeper*, *SVN*, *git*, ...

git $\neq$ github, gitlab, etc.

A brief History

git was developed by Linus Torvalds during the development of the linux kernel in 2005.

A brief History

git was developed by Linus Torvalds during the development of the linux kernel in 2005.

```
$ git clone http://github.com/torvalds/linux.git
$ cd linux
$ cloc .
> 18518596
```

A brief History

git was developed by Linus Torvalds during the development of the linux kernel in 2005.

```
$ git clone http://github.com/torvalds/linux.git
$ cd linux
$ cloc .
> 18518596
```

He needed a way for the developers to work with each other!

What do we use git for?

You may generally use git for

1. software projects,
   - in teams of developers or
   - even if you are working on a project on your own,

What do we use git for?

You may generally use git for

1. software projects,
   - in teams of developers or
   - even if you are working on a project on your own,
2. making these accessable to the public,

What do we use git for?

You may generally use git for

1. software projects,
   - in teams of developers or
   - even if you are working on a project on your own,
2. making these accessable to the public,
3. keeping track of the work on the project,

What do we use git for?

You may generally use git for

1. software projects,
   - in teams of developers or
   - even if you are working on a project on your own,
2. making these accessable to the public,
3. keeping track of the work on the project,
4. trace and fix bugs,

What do we use git for?

You may generally use git for

1. software projects,
   - in teams of developers or
   - even if you are working on a project on your own,
2. making these accessable to the public,
3. keeping track of the work on the project,
4. trace and fix bugs,

but also for

5. theses,
6. storing configuration files,
7. storing password files (`$> pass`),
8. etc.

When are you going to use git?

1. In almost every programming practical!

When are you going to use git?

1. In almost every programming practical!
2. But!
   - different platforms and
   - different hand-in requirements.

When are you going to use git?

1. In almost every programming practical!
2. But!
   - different platforms and
   - different hand-in requirements.
3. Maybe in your everyday work!

When are you going to use git?

1. In almost every programming practical!
2. But!
   - different platforms and
   - different hand-in requirements.
3. Maybe in your everyday work!

Why attend this course?

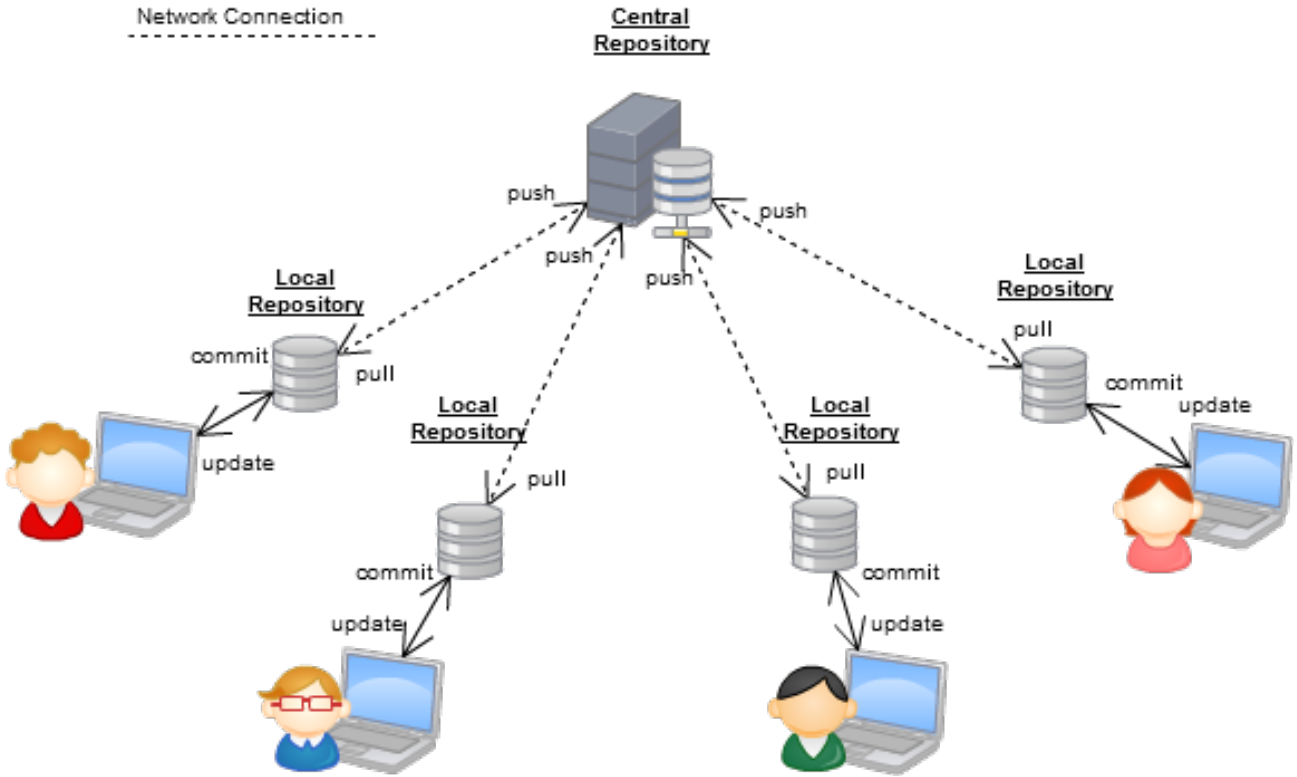
1. Get to understand background,
2. cover basic usage commands and
3. preparation (*and hopefully hours of saved time and struggles!*).

Road Map

- *Today*: Local Repository and git basics
- 24. 10.: Branches, Remotes, non-linear workflow and working in a team
- 29. 10.: Miscellaneous: Tagging, Reflog, .gitconfig, Best Practices and Workflows, Hooks, Background, etc. . .

Credit: *https://xkcd.com/1597/*

Distributed Systems



Credit: *https://jordankasper.com/lessons-learned-teaching-git/*

Similar to *DropBox*, *Google Drive*, *iCloud*, etc., but working very differently in the background!

Local Repository

Local Repository

- Almost all of your work happens here
- Helps keeping a history
- Tracks any changes made
- Divided into four states of stages

Repositories

The *heart* of a git project.

Repositories

The *heart* of a git project.

```
test>$ ls
drwxrwxr-x 40 4096 Sep 26 18:42 ..
drwxrwxr-x  3 4096 Sep 26 18:42 .
drwxrwxr-x  7 4096 Sep 26 18:42 .git
```

Repositories

The *heart* of a git project.

```
test>$ ls
drwxrwxr-x 40 4096 Sep 26 18:42 ..
drwxrwxr-x  3 4096 Sep 26 18:42 .
drwxrwxr-x  7 4096 Sep 26 18:42 .git
```

How can we *initialize* a git repository?

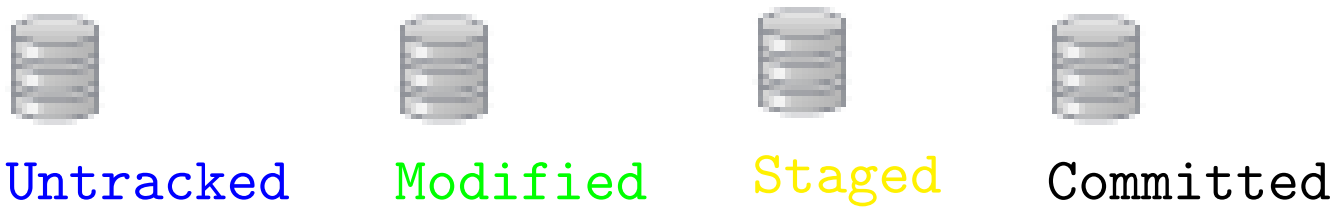
1. `$> git init`
2. `$> git clone <git-url>`

.git


```
test/.git$ ls
drwxrwxr-x 3 4096 Sep 26 18:42 ..
drwxrwxr-x 2 4096 Sep 26 18:42 info
-rw-rw-r-- 1   73 Sep 26 18:42 description
drwxrwxr-x 2 4096 Sep 26 18:42 branches
drwxrwxr-x 4 4096 Sep 26 18:42 refs
drwxrwxr-x 4 4096 Sep 26 18:42 objects
-rw-rw-r-- 1   23 Sep 26 18:42 HEAD
-rw-rw-r-- 1   92 Sep 26 18:42 config
drwxrwxr-x 7 4096 Sep 26 19:32 .
drwxrwxr-x 2 4096 Sep 30 21:17 hooks
```

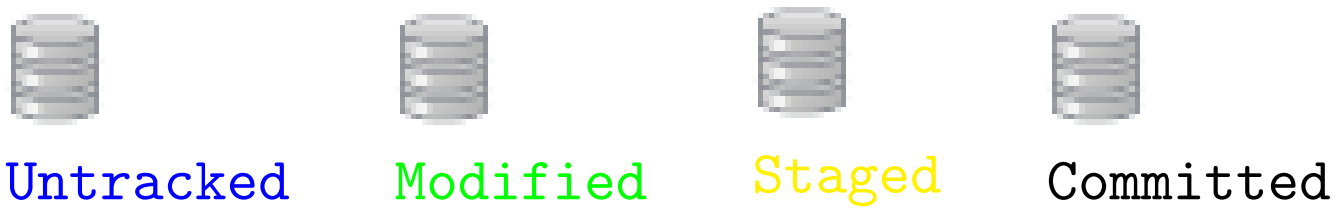
Stages

- Every file in your directory is either:
  1. Untracked
  2. Modified



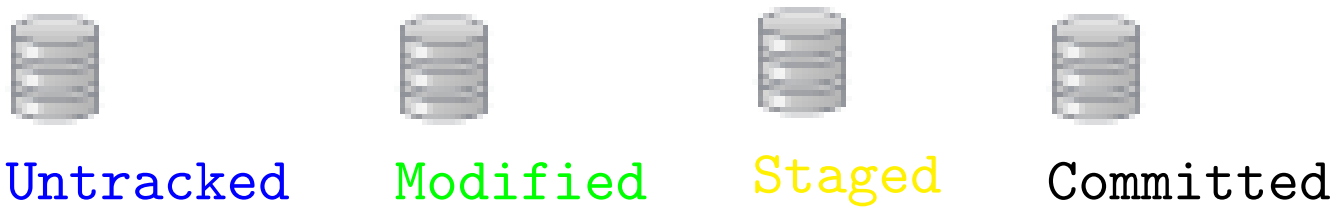
Stages

- Every file in your directory is either:
  1. Untracked
  2. Modified
  3. Staged



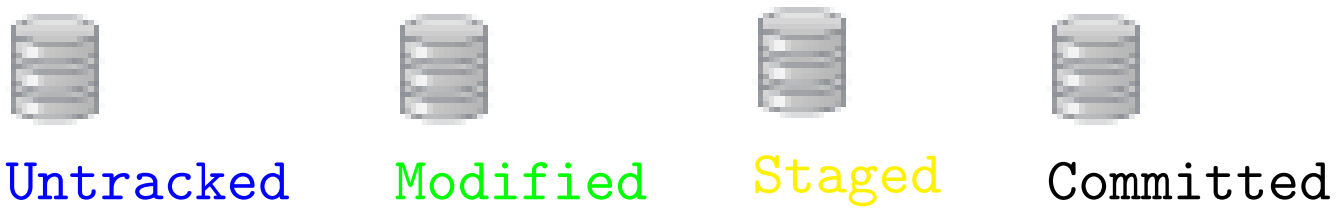
Stages

- Every file in your directory is either:
  1. Untracked
  2. Modified
  3. Staged
  4. Committed



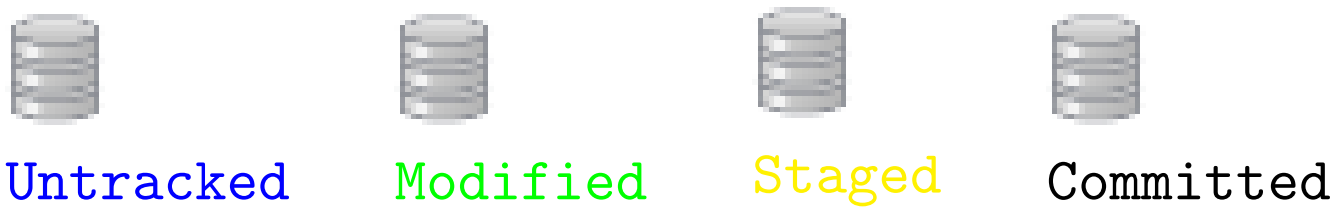
Stages

- Every file in your directory is either:
  1. Untracked
  2. Modified
  3. Staged
  4. Committed
  5. or (Unmodified)



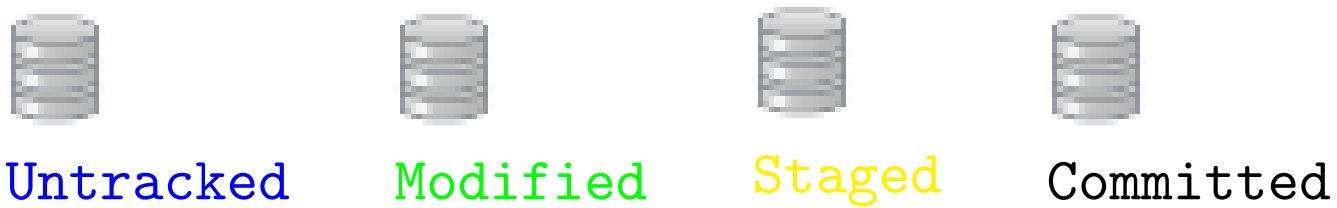
Stages

- Every file in your directory is either:
  1. Untracked
  2. Modified
  3. Staged
  4. Committed
  5. or (Unmodified)
- You move files between these stages



Stages

- Every file in your directory is either:
  1. `Untracked`
  2. `Modified`
  3. `Staged`
  4. `Committed`
  5. or `(Unmodified)`
- You move files between these stages
- git keeps track of moves from `Staged` to `Committed` (i.e. your *commits*)



Stages

How do we move files between the stages?
How can we fix mistakes before committing?
How do we inspect a git-directory (and the stages of individual files)?

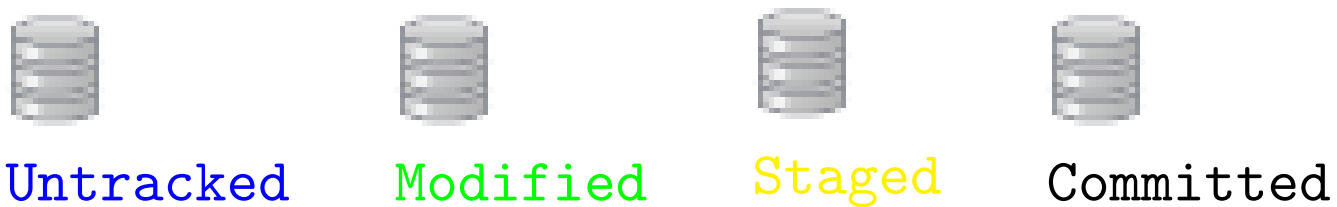
Commits

What is a *commit*?

Commits

What is a *commit*?

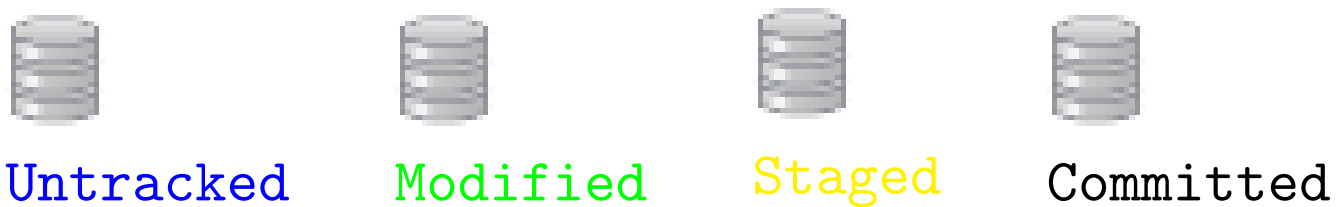
- One entry in the history of you project
- Saves your staged changes in your local repository
- Messaging interface for you and your team



Commits

What is a *commit*?

- Author
- Message
- Date
- Committed Files



Commits

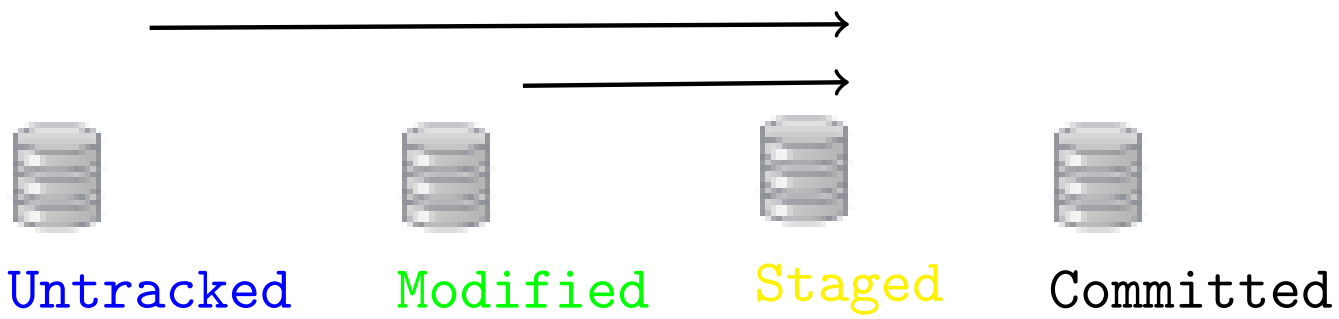
What is a *commit*?

- Author
- Message
- Date
- Committed Files

Adding and Staging

Move files from Untracked/Modified to Staged:

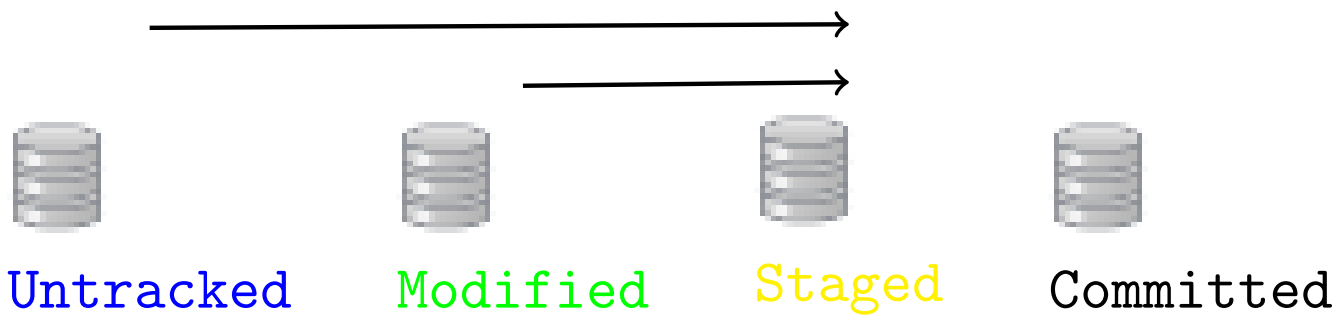
- `$> git add <file,dir>`,



Adding and Staging

Move files from Untracked/Modified to Staged:

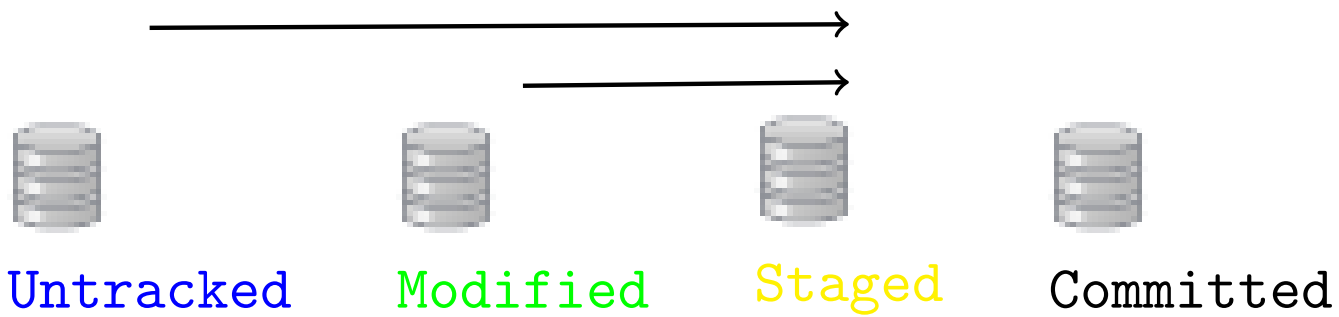
- `$> git add <file,dir>`,
- `$> git add -u`, *(only from Modified)*



Adding and Staging

Move files from Untracked/Modified to Staged:

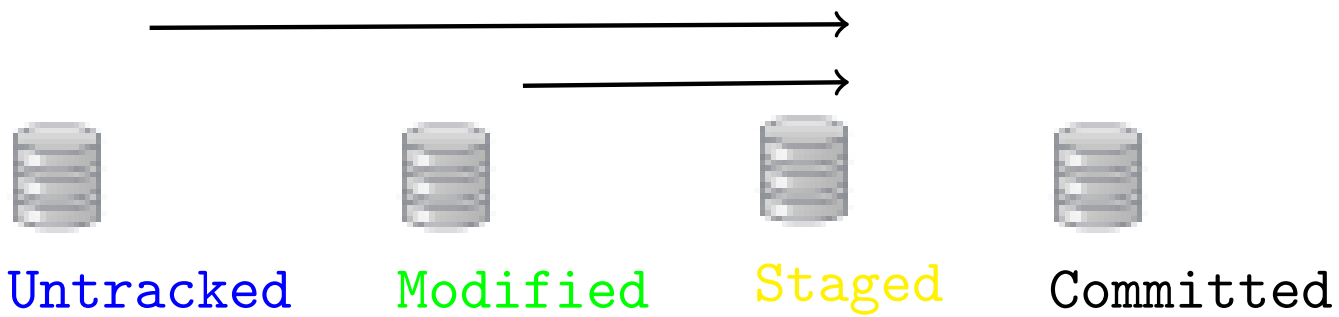
- `$> git add <file,dir>`,
- `$> git add -u`, *(only from Modified)*
- `$> git add -p <file,dir>`



Adding and Staging

Move files from Untracked/Modified to Staged:

- `$> git add <file,dir>`,
- `$> git add -u`, *(only from Modified)*
- `$> git add -p <file,dir>`
- `$> git add -A` *(considered to be bad practice)*



Committing

After doing some work on the project:

- check all changes via `$> git diff` or `$> git diff --staged`

Committing

After doing some work on the project:

- check all changes via `$> git diff` or
  `$> git diff --staged`
- `$> git commit -m "<commit message>"`

git then adds a commit to your *log* and stores a *snapshot* of modified objects.

Committing

After doing some work on the project:

- check all changes via `$> git diff` or `$> git diff --staged`
- `$> git commit -m "<commit message>"`
- Better: `$> git commit`

git then adds a commit to your *log* and stores a *snapshot* of modified objects.

Status

- `$> git status`
- Shows you the current status of you working directory.
- Which files are *staged*, *modified*, etc.

Logs

- `$> git log`
- *By default*, shows you the commit history of you current branch.

Logs

- `$> git log`
- *By default*, shows you the commit history of you current branch. IMHO in a rather clumsy way...
- Since git is all about versioning and information ... we want this to be prettier!

Logs

- `$> git log`
- *By default*, shows you the commit history of you current branch. IMHO in a rather clumsy way...
- Since git is all about versioning and information ... we want this to be prettier!
- `$> git log --oneline`

Logs

- `$> git log`
- *By default*, shows you the commit history of you current branch. IMHO in a rather clumsy way...
- Since git is all about versioning and information ... we want this to be prettier!
- `$> git log --oneline`
- `$> git log --graph`

Logs

- `$> git log`
- *By default*, shows you the commit history of you current branch. IMHO in a rather clumsy way...
- Since git is all about versioning and information ... we want this to be prettier!
- `$> git log --oneline`
- `$> git log --graph`
- ... and combined: `$> git log --oneline --graph`

Logs

- `$> git log`
- *By default*, shows you the commit history of you current branch. IMHO in a rather clumsy way...
- Since git is all about versioning and information ... we want this to be prettier!
- `$> git log --oneline`
- `$> git log --graph`
- ... and combined: `$> git log --oneline --graph`
- A bit more advanced:

```
$> git log --graph --pretty=format:'...'  ...
```

How to undo things

- git generally only adds data and information, but it is possible to undo, amend and even change the history!
- Obviously changing commited history is not considered to be best practive.
- You can also drop changes you have made completely.

Amending the last commit

- `$> git commit --amend`
- This amends your last commit by adding the currently staged files.

Amending the last commit

- `$> git commit --amend`
- This amends your last commit by adding the currently staged files.
- Two important use cases:
  1. Fixing typos in the commit message and
  2. if you have forgotten to add files:

```
$> git commit -m "some commit"
$> git add ...
$> git commit --amend -m "some commit and xyz"
```

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.
- `--soft` Undo all commits back until `<commit>` and keeps all files staged.

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.
- `--soft` Undo all commits back until `<commit>` and keeps all files staged.
- This discards `$> git commit`.

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.
- `--mixed` Undo all commits back until `<commit>` and unstages files.

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.
- `--mixed` Undo all commits back until `<commit>` and unstages files.
- This discards `$> git commit` and `$> git add`

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.
- `--hard` Undo all commits back until `<commit>` and **discards** all changes made.

git reset

- `$> git reset (--soft|--mixed|--hard) <commit>`
- Three different options, *–mixed* being the default.
- `--hard` Undo all commits back until `<commit>` and **discards** all changes made.
- This discards `$> git commit`, `$> git add` and any changes to the files.

Unstaging Files

- `$> git status` tells you how to do this:
- `(use "git reset HEAD <file>..." to unstage)`
- `$> git reset --mixed <file>` undos `$> git commit` and `$> git add`, if the file is not commited it will just do the latter!

Untracking Files

- In case you have accidentally added a file:
- Use `$> git rm --cached` to untrack it.
- The `--cached` option will keep it in your working directory.

Untracking Files

- In case you have accidentally added a file:
- Use `$> git rm --cached` to untrack it.
- The `--cached` option will keep it in your working directory.
- If you want to permanently ignore the file:
  - `$> touch .gitignore`
  - `$> echo <filename> >> .gitignore`

Untracking Files

- In case you have accidentally added a file:
- Use `$> git rm --cached` to untrack it.
- The `--cached` option will keep it in your working directory.
- If you want to permanently ignore the file:
  - `$> touch .gitignore`
  - `$> echo <filename> >> .gitignore`
- Add `*.<filetype>` to exclude all files of the given type.

Undo Changes

- If you want to discard changes you have made to a single file:
- `$> git checkout <file>`

Undo Changes

- If you want to discard changes you have made to a single file:
- `$> git checkout <file>`
  - Useful if you are experimenting or
  - you went a bit to far and want to go back to a safe state.

Undo Changes

- If you want to discard changes you have made to a single file:
- `$> git checkout <file>`
  - Useful if you are experimenting or
  - you went a bit to far and want to go back to a safe state.
- **Caution:** Both `$> git checkout <file>` and `$> git reset --hard <commit>` make changes to your working directory by discarding changes.

Summary

- What is version control?
- What is a distributed system?
- What is a local repository?
- What are the different stages a file may be in?
- What is a commit?
- What is a snapshot?
- How can we move files between stages?
- What is the git log?
- How can we fix errors?
- What does the `.gitignore` file do?