Git

Beginner Tutorial

SED, Inria Paris

Schedule

Morning

1. **Introduction**
2. **The basic commands**
3. **Working with branches**

Afternoon

4. **Advanced usage**
5. **Working with GitLab and GitHub**

1. Introduction

1.1. About version control systems (VCS)

1.2. History

1.3. Git features

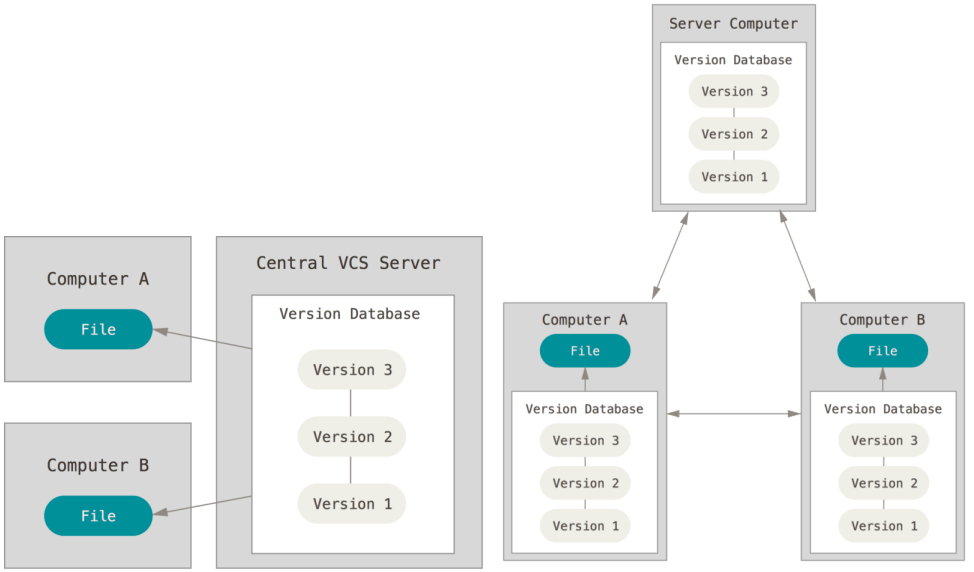
1.4. How it works

1.5. The 3 states

1.6. Structure of a Git repository

About version control

- Maintaining an history of changes
- Different version control modes: **Local, Centralized, Distributed**
- Examples of centralized version control tools: CVS, Subversion
- Examples of distributed version control tools: Mercurial, Git



[Reference: Git Book]

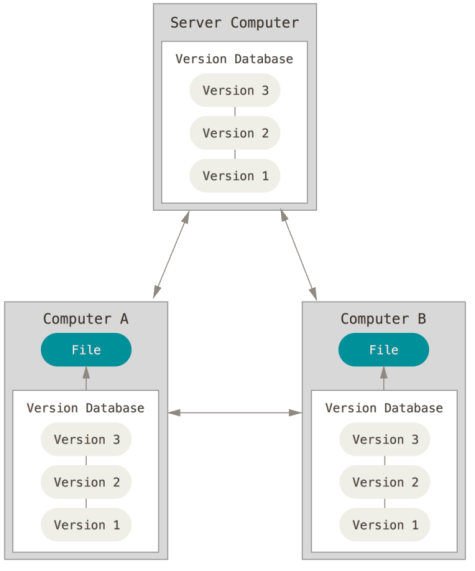
The history of Git

Git was created to solve the maintenance burden of the **Linux** kernel source code

- Before 2002: Contributions made using patches
- 2002: Developers tried BitKeeper
- 2005: Linus Torvalds started to write Git

Git main features

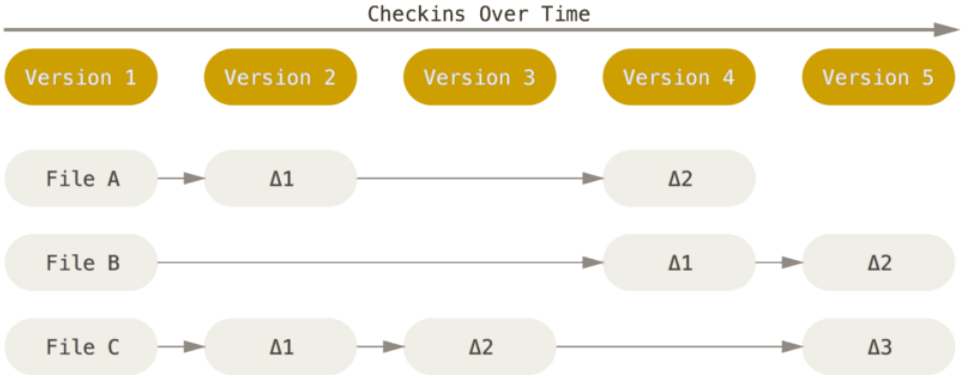
- Extremely **fast**
- Supports **non linear development** (branches)
- **Distributed**: the whole repository is duplicated by each developer
- The version control remains **local**: no network connection required
- Allows to maintain **very large codebases**



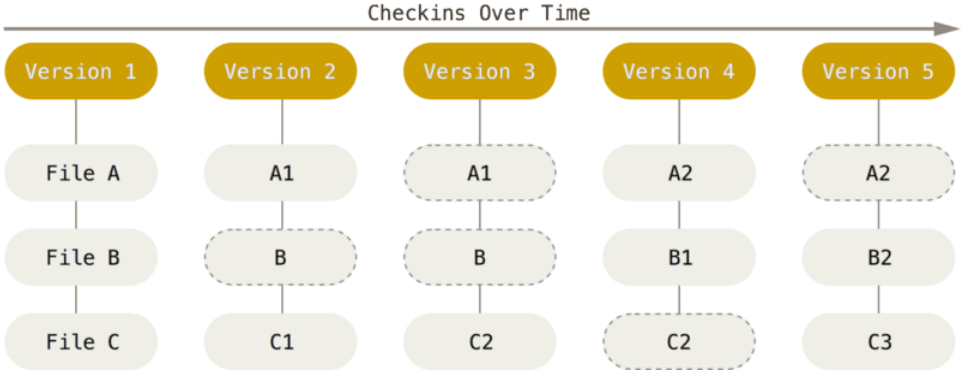
How it works

Git maintains snapshots

Diffs

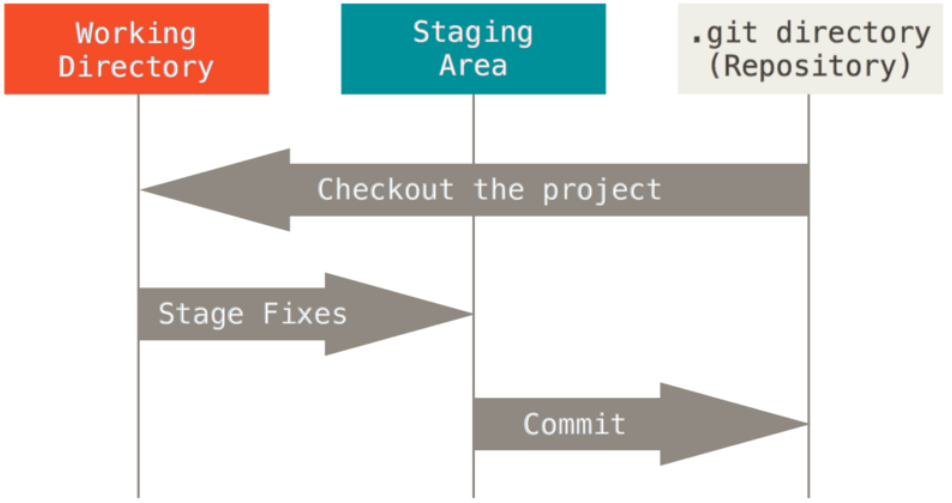


Snapshots



The 3 states

- **Modified** : Working directory
- **Staged** : Staging area
- **Committed** : Integrated to the repository database



Check the current states of the local copy:

```
$ git status
```

Structure of a Git repository

```
<Local copy>
├── .git
│   ├── branches       # Contains the list of branches
│   ├── config         # Local configuration file
│   ├── description    # File containing the repository name
│   ├── HEAD           # Pointer to the current commit
│   ├── hooks
│   ├── index          # File containing the changes in staging
│   ├── info
│   │   └── exclude    # Global file containing ignored patterns
│   ├── objects        # Objects database managed by Git
│   │   ├── info
│   │   └── pack
│   └── refs           # All known references contained in the local repository
│       ├── heads
│       └── tags
└── <working directory>
```

Each object is identified by a SHA-1 checksum

→ **guarantees data integrity**

Different ways to use Git

- Using the command line interface: gives access to **all** features

```
$ git help <command>  # display manual of a particular command
```

- Graphical user interfaces (gitk, git-gui, qgit, ungit, etc)
- Directly integrated to IDEs

→ **We will use the command line interface**

Setup:

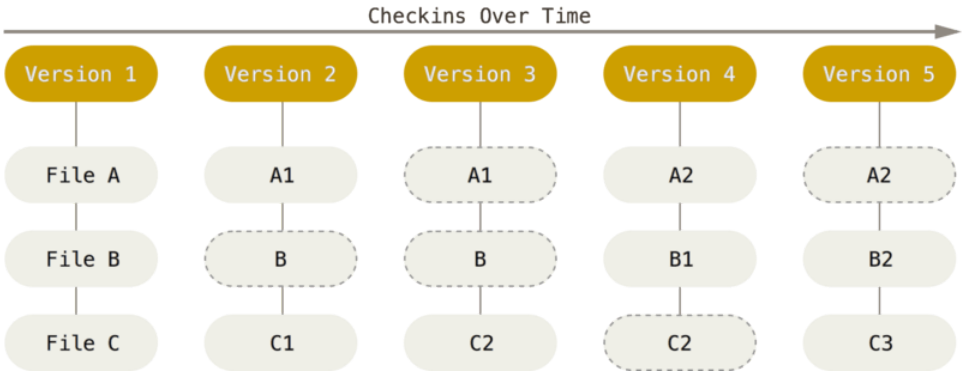
- Linux :

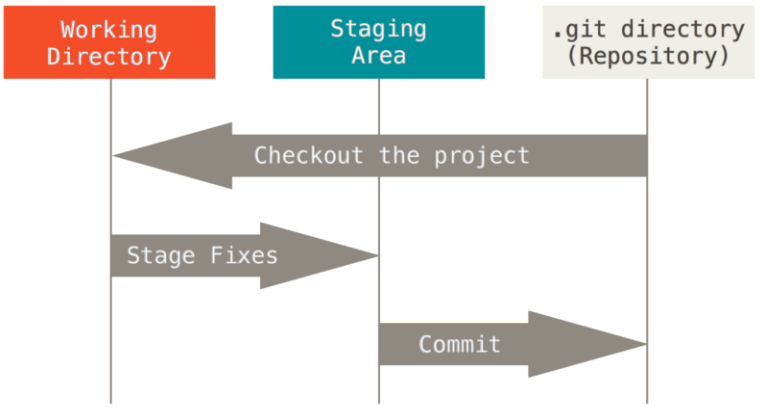
```
sudo apt-get install git-all
sudo yum install git-all
```

- Windows: Git for Windows
- MacOS: Git setup

Summary

How it works, Setup, Different ways to use Git





2. The basic commands

Configuration

- Use simple configuration files
- 3 levels:
  - **local** : `<local copy>/.git/config`, by default
  - **global** : `~/.gitconfig`, option `--global`
  - **system** : `/etc/gitconfig`, option `--system`
- Either edit the files directly, or use `git config` command

```
$ git help config
$ git config --edit            # edit local parameters
$ git config --global --edit   # edit global parameters
```

Always start by filling your user name and email, at the **global** level:

```
$ git config --global user.name "First Last"
$ git config --global user.email your.email@organisation.com
```

Exercises : install et configure git

1. If not done already, install Git
2. Globally configure your preferred editor (option `core.editor`)
3. Globally configure your user name and email
4. Add a few `alias` (examples: alias.st=status, alias.conf=config, etc)

Aide : `git config --global <paramètre> <valeur>`

Initialize a new repository

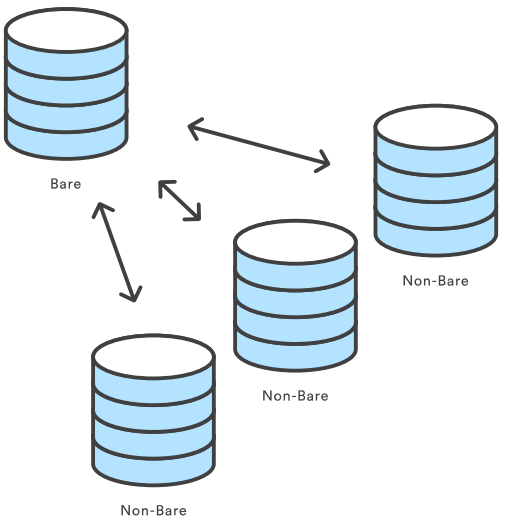
- Start to version code with Git using a single command in the base directory:

  ```
  $ git init
  ```

- When in server mode: **no working directory**

  ```
  $ git init --bare <repository directory>.git
  ```

The **`.git`** extention is used as a convention



Add changes

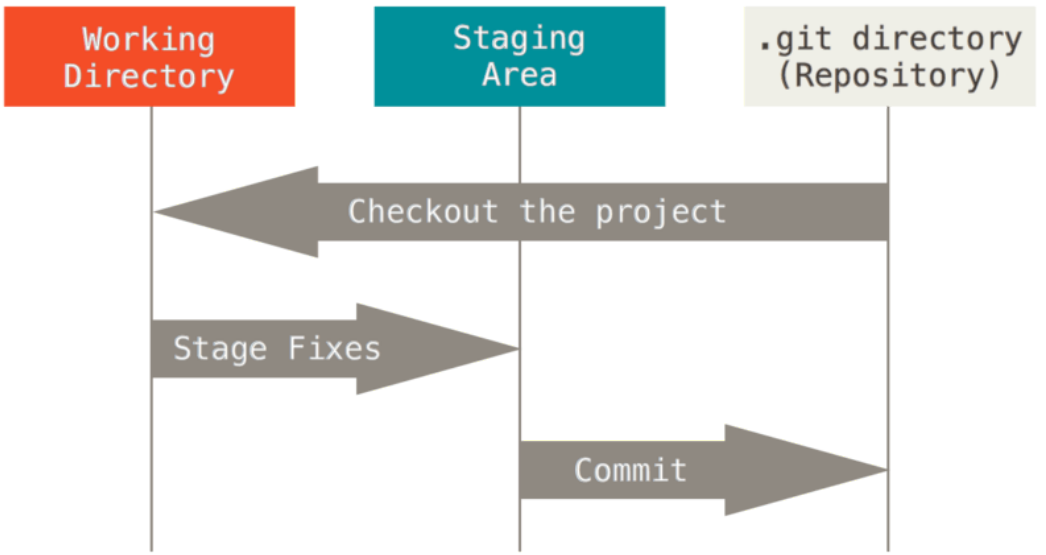
- Add changes to the index ("Stage Fixes") → Staged

  ```
  $ git add <list of files> ou <list of directories>
  ```

- Add changes to the local copy ("Repository") → Committed

  ```
  $ git commit -m "commit message"
  ```

- Remember to use `git status` to check the state of your local copy



Add changes: some tricks

- Skip the staging area step using option `-a`

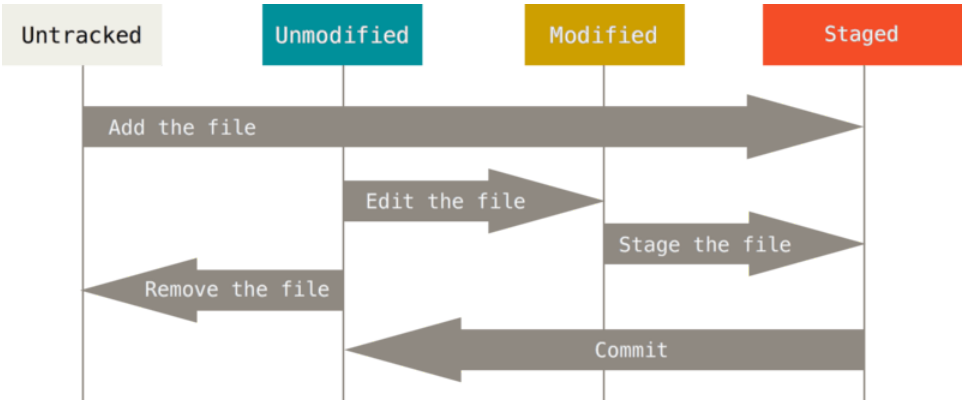
  ```
  $ git commit -a -m "message de commit"
  ```

- Add partial changes to the staging area using option `-p`

  ```
  $ git add -p <list of files>
  ```

  Then type *y* ou *n* to choose which blocks of changes to stage

Managing files



File status life cycle

- Delete a list of files from the working directory and put this change in the staging area: `$ git rm <list of files>`
- Rename/move a file : `git mv <src> <dst>`
- Remember the `.gitignore` file at the base directory of the working directory to ignore file/directory patterns from the *untracked* state
- *untracked* files/directories are treated like regular files/directories

Track differences between the 3 states

This is simply done using `git diff`

- By default, `git diff` → shows the differences between the working directory and the staging area or, when the staging area is empty, the last commit
- `git diff --staged` → shows the differences between the staging area and the last commit
- `git diff <commit>` → shows the différences between the working directory and a commit
- `git diff <commit1> <commit2>` → shows the differences between 2 commits

```
$ git diff
diff --git a/README.txt b/README.txt
index e69de29..8fd7633 100644
--- a/README.txt
+++ b/README.txt
@@ 0 -1,0 +1 @@
-Previous content
+New content
```

Exercises: start a small Python project

Let's discover Git usages by writing a small Python project implementing simple mathematical functions.

This is how the project organization will look like in the end:

```
├── .gitignore
├── myfuncs.py
├── README.md
└── test_myfuncs.py
```

Follow the README.md file in exercises/01-start

Track the history of a local copy

- Using a graphical user interface: `gitk`
- `git log` displays the history of commits
- Some useful `git log` options:

```
-n                # Limit the displayed history to the last n commits
-p                # Display the changes contained in each commit
--pretty=oneline  # Each commit information is displayed in one oneline
--pretty=format:"<rule>" # Finely tune the pattern used to display the information
--graph           # Display the history of commits as a tree (see branches later)
```

Useful: Configuration option `pager.log=less` to activate pagination

Revert changes

3 useful commands:

- `git commit --amend` → modify the last commit
- `git reset`
  - revert the last commit if the staging area is empty → the changes in the commit are kept in the working directory
  - with the `--hard` flag: changes are totally removed and thus lost (**watch out!**)
  - remove changes from the staging area if there are any
  - `-p` flag: select changes to revert
- `git checkout`: reload the last commit in the working directory (**watch out!**)
  - `git checkout <file or directory>`: revert changes on a file/directory
  - `git checkout <commit> <file or directory>`: load in the staging area the diff between the current version and the commit version of the file/directory
  - `-p` flag: select changes to cancel

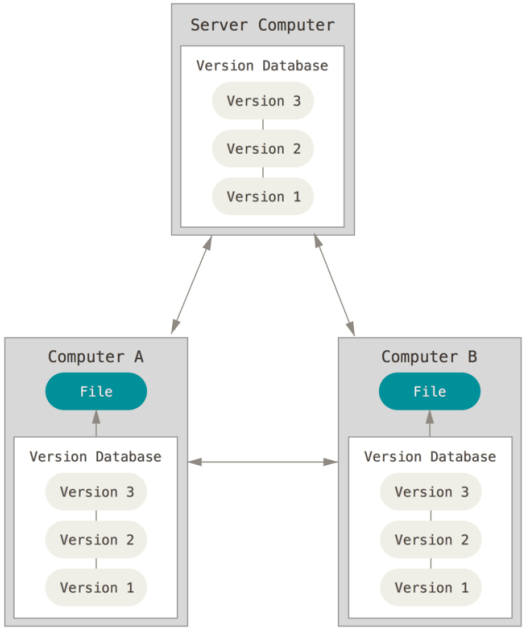
Exercises: manage changes

Follow the README.md file in exercises/02-history.

Working with remote repositories

Principle :

- All developers own more or less synchronized local copies of a Git repository
- Only committed changes can be exchanged
- Information about remote repositories are available using `git remote`



Cloning a remote repository

- **git clone** → one gets **a local copy** of a remote repository

  ```
  $ git clone <repository url>
  ```

- There are several types of **url**:

  - A local directory containing a bare repository (e.g initialized using `--bare`)
  - A read-only repository on a remote host:

    ```
    git://host/<path to the repository>
    ```

  - Read/write access, depending on the rights, from a remote host:

    ```
    ssh://user@host:<path to the repository>
    ```

    ```
    git@host:<path to the repository>
    ```

    ```
    https://host/<path to the repository>
    ```

- By default, Git loads the `main` branch locally (formerly it was `master`)

Introduction to remote repositories

- They are identified by their **url** and **name**
- By default, when cloning a repository, the remote is called **origin**.
- `git remote` displays the list of remote repositories, by name
- `git remote -v` gives more details (such as the url)
- To add a remote repository: `git remote add <name> <url>`
- Other useful commands:
  - `git remote rm <name>`: delete the remote repository (**!locally**)
  - `git remote rename <old> <new>`: rename the remote repository
  - `git remote show <name>`: inspect the remote repository

Synchronize with a remote repository

- Push local changes to a remote repository:

  ```
  $ git push <remote name> <branch>
  ```

- Locally get the updates from a remote repository:

  ```
  $ git fetch <remote name>
  ```

- Fetch changes from remote `<branch>` and integrates them in current branch:

  ```
  $ git pull <remote name> <branch>
  ```

Exercises: some manipulations with remote repositories

Follow the README.md file in exercises/03-remotes.

Summary: the basic commands

So far we have seen how to locally manage changes using Git, in particular:

- How to configure Git

  → `git config`, options `--global`, `--edit`

- How to initialize a local repository and how to interact with remote repositories

  → `git init`, `git clone`, `git pull`, `git push`, `git fetch`

- How to manage changes between the 3 local states: modified, staged and committed

  → `git add`, `git commit`

- How to cancel/revert local changes

  → `git commit --amend`, `git reset`, `git checkout`

- How to track local changes

  → `git status`, `git diff`, option `--staged`, `git log`

3. Working with branches

3.1. What is a branch?

3.2. Starting a new branch

3.3. Switching branches

3.4. Merging branches

3.5. Resolving conflicts

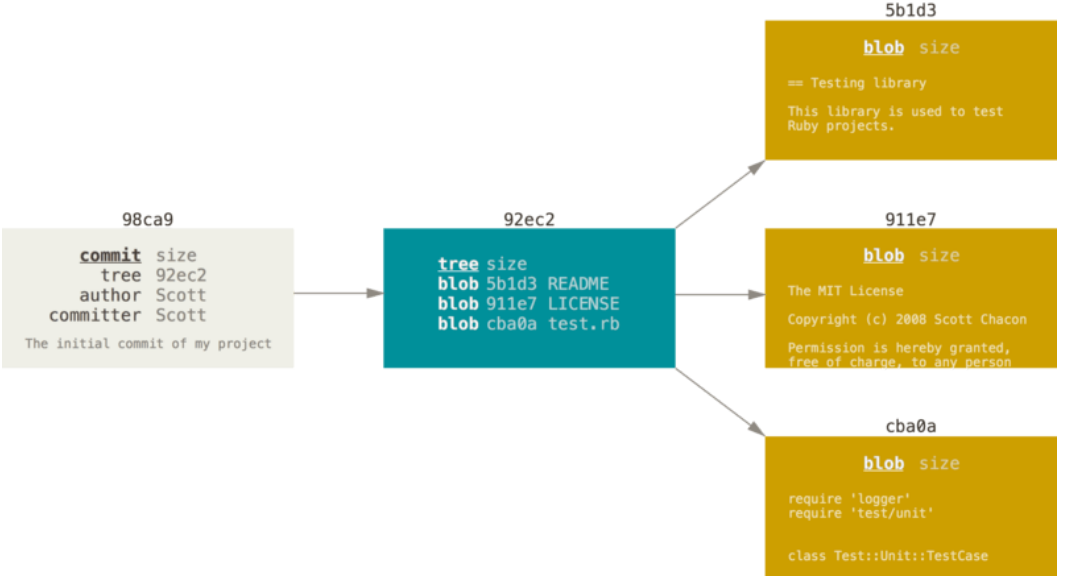
3.6. Remote branches

More insight on commits

Main information stored in a commit is:

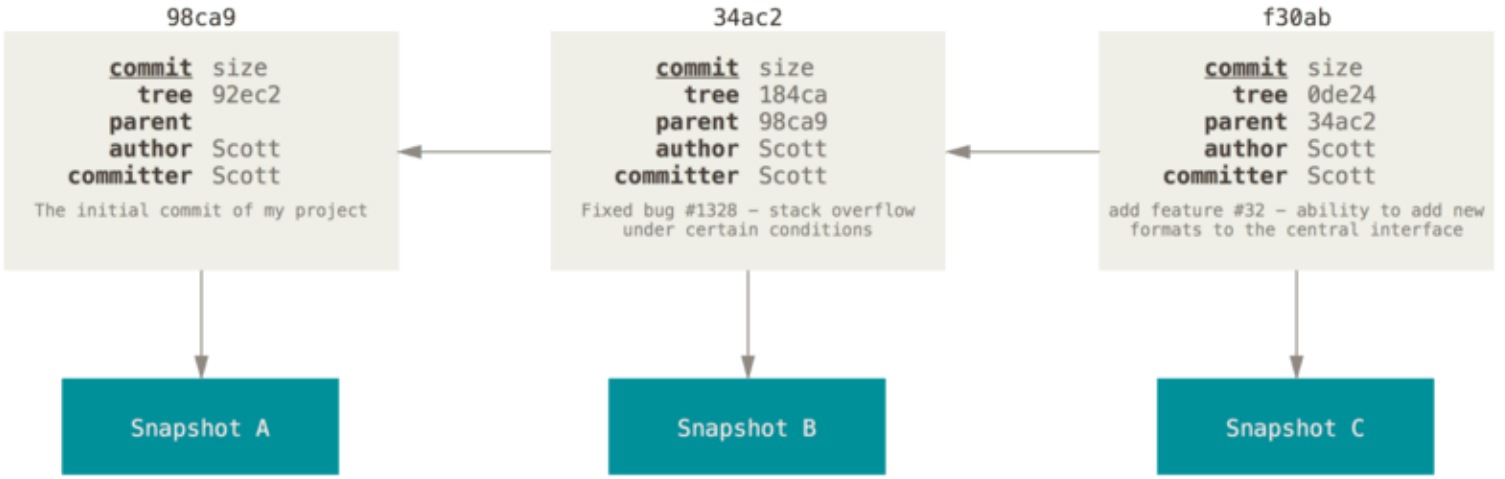
- Metadata: author name, date, message
- Pointer to the content snapshot
- Pointer(s) to ancestor commit(s)

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```



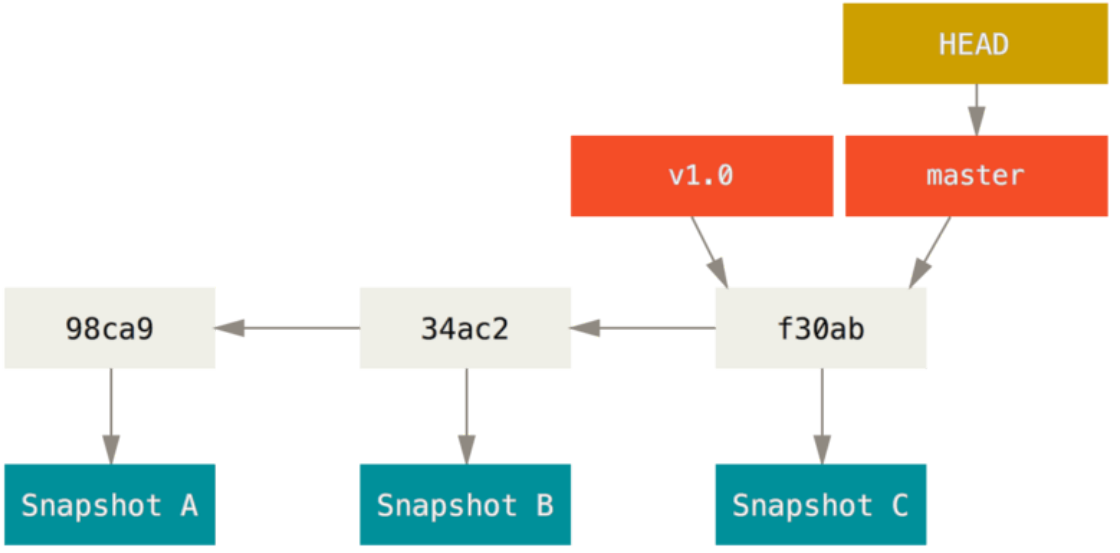
More insight on commits

The history corresponds to a linked list of commits:



What is a branch?

- **One branch** = **one pointer** to a commit
- For each new commit, the pointer automatically moves forward
- More precisely, **a branch** = **a file** which name is the branch name and containing the commit hash (SHA-1)



Reminder: Internal structure of a Git repository

```
<Local copy>
├── .git
│   ├── branches       # Contains the list of branches
│   ├── config         # Local configuration file
│   ├── description    # File containing the repository name
│   ├── HEAD           # Pointer to the current commit
│   ├── hooks
│   ├── index          # File containing the changes in staging
│   ├── info
│   │   └── exclude    # Global file containing ignored patterns
│   ├── objects        # Objects database managed by Git
│   │   ├── info
│   │   └── pack
│   └── refs           # All known references contained in the local repository
│       ├── heads
│       └── tags
└── <working directory>
```

Some notations

- `HEAD` → current version loaded in the working directory
- `HEAD^` ou `HEAD~` → first parent of the `HEAD` commit
- `HEAD~n` → n-th parent of the `HEAD` commit
- `Detached HEAD` → the current position of `HEAD` doesn't correspond to any known branch
- `main` → name of the default branch (formerly, it was `master`)
- List branches:

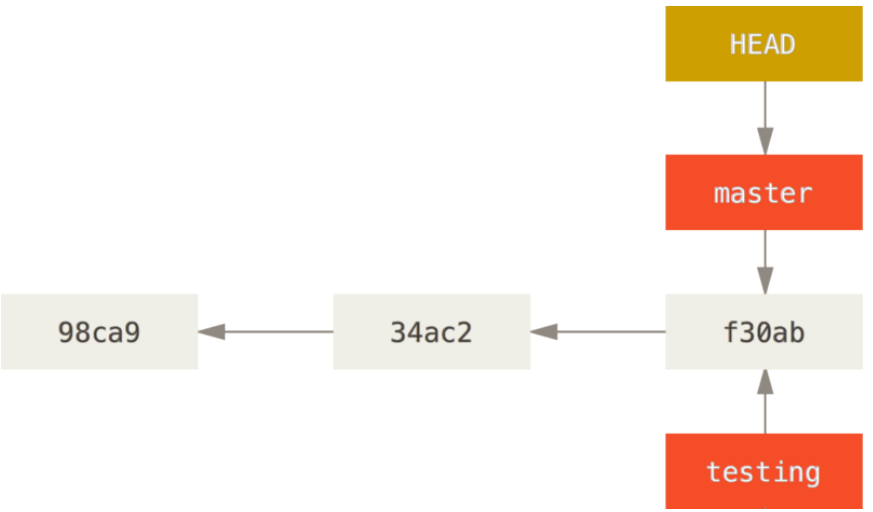
```
$ git branch    # display local branches, the current branch is
                # preceded by *
$ git branch -a # display all branches, local and remote
```

Starting a new branch

- Creating a new branch is very fast: it just creates a single file
- `git branch <new branche>` → create a new pointer to the current commit
- `git log --oneline --decorate` → displays the history with known branches

Example:

```
$ git branch testing
```



Basic operations on branches

`git branch`: base command for managing branches

- Rename a branch:
  `git branch -m <old name> <new name>`
- Delete a branch:
  `git branch -d <branch name>` → No risk. The branch is deleted only if it's up-to-date with its *uptream* version
- Delete a branch:
  `git branch -D <branche>` → Force branch deletion
- Display branches that are merged/unmerged in the current branch:
  `git branch --merged/--no-merged`
- Display the hash and comment of local branches: `git branch -v`

Switching branches

- `git checkout <branch>` → move `HEAD` to the commit pointed by `<branch>`
- `git checkout` also loads the snapshot of the commit in the working directory

Example:

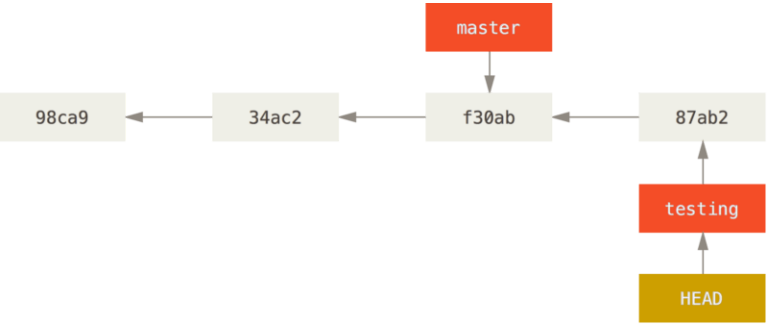
```
$ git checkout testing
```

- use `-b` flag to create and automatically switch to the newly created branch

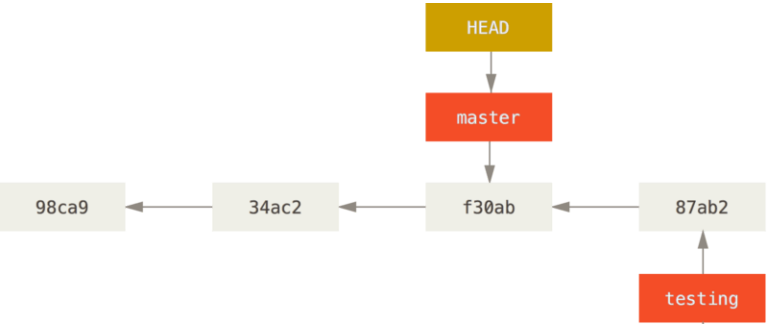
Switching branches

Watch out!: switching branches changes the content of the working directory

```
$ vim test.rb
$ git commit -a -m 'some change'
```



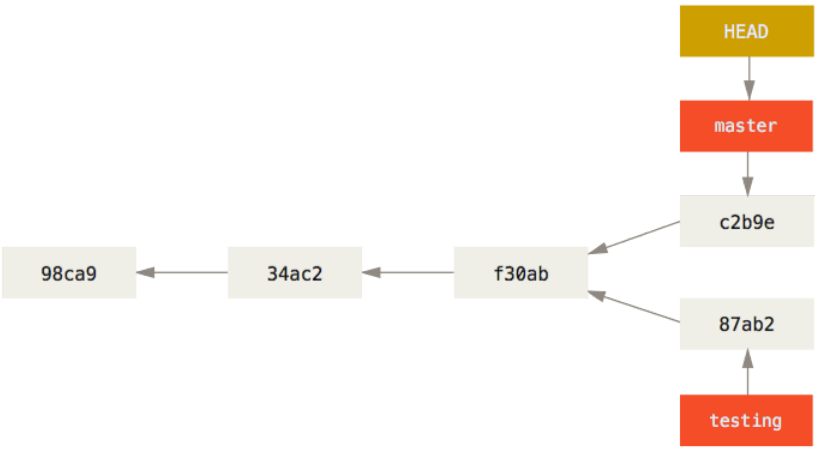

```
$ git checkout main
```



Diverging branches

Let's add a new commit in `main`:

```
$ vim test.rb
$ git commit -a -m 'other changes'
```



- The diverging branch can be seen using `git log --oneline --decorate --graph --all` or with `gitk --all`

Exercises: Manipulating branches

Follow the README.md file in exercises/04-branches

Merging branches

2 cases:

- One branch (`branch1`) is the starting point of another branch (`branch2`)

→ Merging `branch2` in `branch1` = **fast-forward** move of `branch1` to `branch2`

- Branches `branch1` et `branch2` have diverged

→ Merging `branch2` in `branch1` = create a merge commit

Merge commit : commit with 3 ancestors

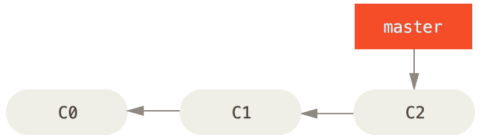
- ancestor of *branch1*
- ancestor de *branch2*
- common ancestor of *branch1* and *branch2*

Let's have a closer look to an example: one wants to hot fix a *bug* in main while working on another branch

Merging branches: workflow example

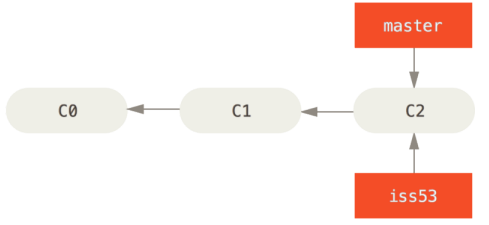
1. One starts from main *main*

```
git checkout main
```



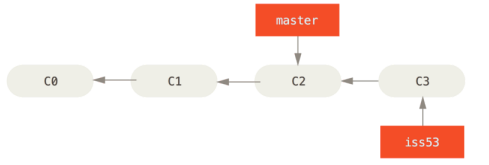
2. One creates and switches to a branch *iss53*

```
git checkout -b iss53
```



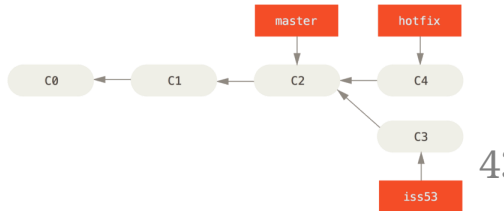
3. One commits changes (C3) in branch *iss53*

```
git commit -a -m "new feature"
```



4. One switches back to *main* and switches to the new *hotfix* branch

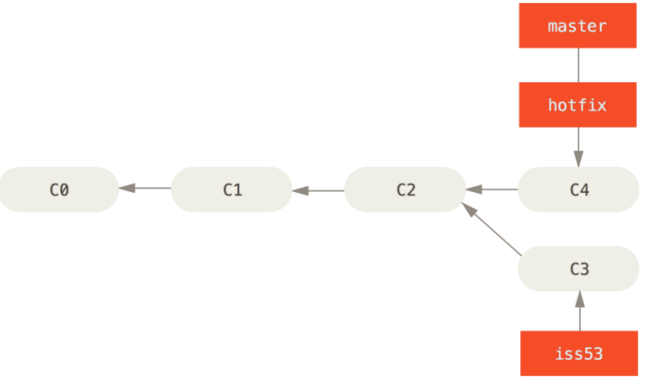
```
git checkout main
git checkout -b hotfix
```



Merging branches: workflow example

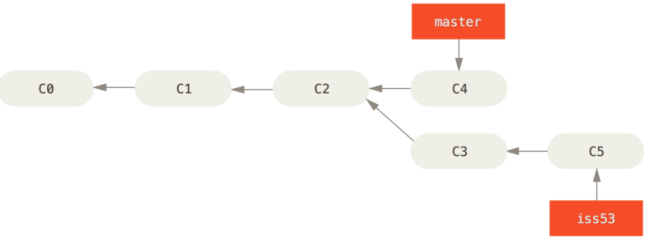
5. When merging *hotfix* into `main` → fast-forward move of `main` towards `hotfix`

```
git checkout main
git merge hotfix
```



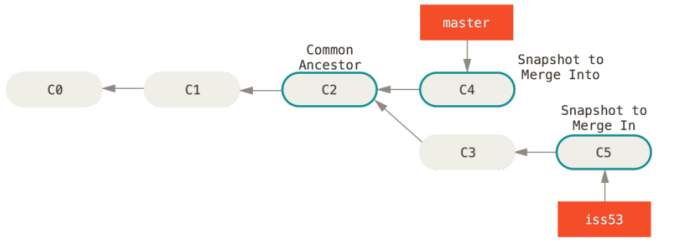
6. One commits a new change in `iss53` branch.

```
git checkout iss53
git commit -a -m "another change"
```



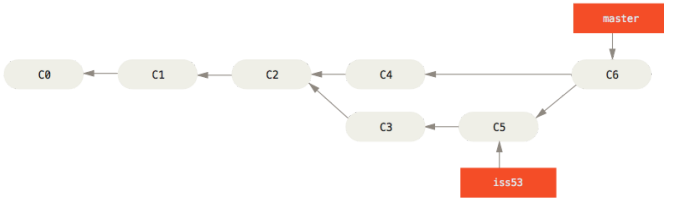
Merging branches: workflow example

7. *iss53* (C5) and *main* (C3) common ancestor is C2.



Merging *iss53* in *main* creates a merge commit C6.

```
git checkout main
git merge iss53
```



Merging branches: how to solve conflicts

- Git is very good at merging things, but *sometimes* it's not able to do it alone Example of merge that produces conflicts:

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

→ 2 ways to solve the conflict:

- edit the files by hand
- use `git mergetool`
  Tips :

```
git config --global merge.tool <your favorite merge conflict tool>
```

  Some existing tools: meld, kdiff3, tortoisegit, vimdiff, etc

→ `git commit` once the conflict is solved

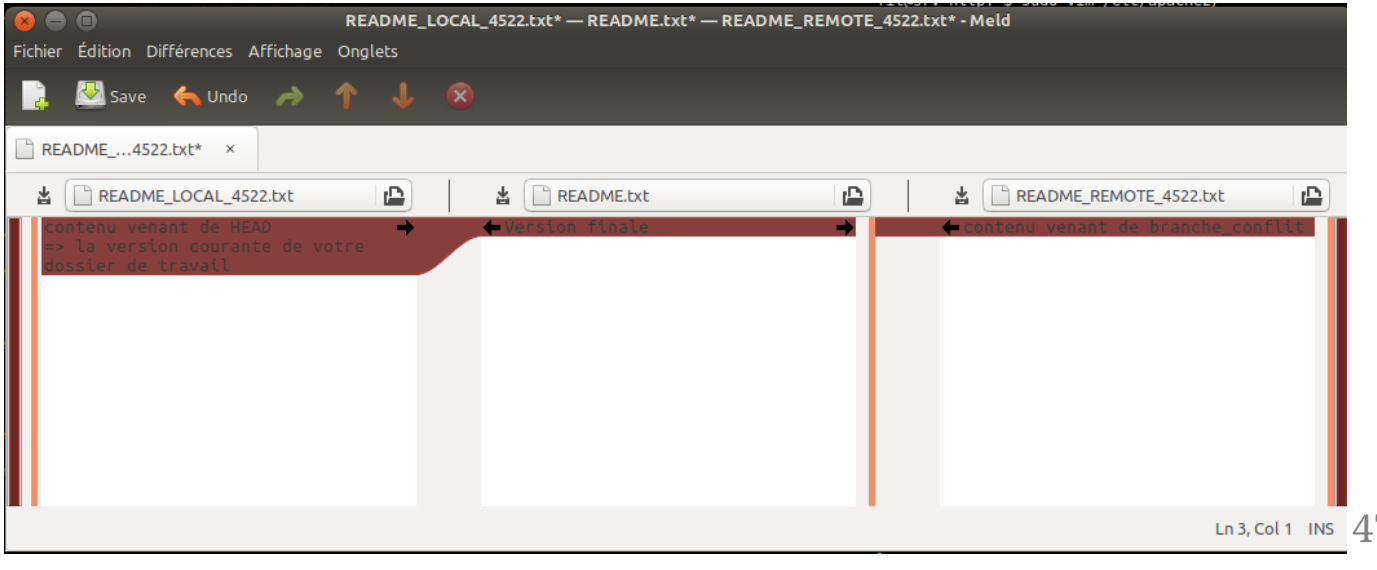
→ `git merge --abort` abort the merge attempt (when things go wrong)

How to solve conflicts

- Manually solving the conflict:

```
<<<<<<< HEAD
Content from HEAD => the current version in the working directory
=======
Content from the conflicting_branch branch
>>>>>>> conflicting_branch
```

- Solving using a graphical tool:



Exercises: merging branches

Follow instructions in the README.md file exercises/05-merging-branches.

Remote branches

- These special branches are pointers to branches on remote repositories
- They are named following the `<repository>/<branch>` name pattern. Example: `origin/main`
- Display the list of remote branches: `git branch -a` or `git remote show <dépôt>`
- These branches are **local** to the local repository → use `git fetch <repository>` to synchronize them (option `--all` will sync all repositories)
- They cannot be modified → they don't follow new commits

Example

Tracking remote branches

- By default, `git clone` automatically sets the `main` branch to track `origin/main`
- But a branch created locally (`git branch <branch>`) and then pushed (`git push <remote> <branch>`) won't *track* its remote branch by default
  → `git pull` without options won't work
  → automatic track must be enabled using the `-u` flag during the first push
- Activate the tracking when creating a branch:
  - `git branch -u origin/<branch>` → creates `<branch>` with tracking of `origin/<branch>` enabled
  - `git checkout --track origin/<branch>` → creates `<branch>` and switches to it
  - `git checkout -b <local branch> origin/<branch>` → creates `<local branch>` from `origin/<branch>` and switches to it
- Example: activate tracking of `origin/main` on local `main` branch:
  `git branch --set-upstream-to=origin/main main`
- *Personal advice* : be **explicit**, limit the use of bare `pull`/`push` commands

Other actions on remote branches

- Delete a branch on a remote repository:
  `git push <remote> --delete <branch>`
  or `git push <remote> :<branch>`
  → but the local branches still exist

- Remove local references (`<remote>/<branch>`) deleted in remote repository
  (! `-n` option for dry-run mode):
  `git remote prune -n <remote>`

Some advice on how to manage your branches

- When starting a new feature, branch from `main`
- Before creating a new branch, sync the base branch with the remote (e.g. fetch)
- Follow the idiom: One branch per feature or fix
- When unsure, before merging, create a backup branch in case things go wrong

Summary

In this (long) section, we learned the principle and usage of branches with Git, in particular:

- A branch is just a file with a commit hash
- `HEAD` is a pointer to the commit loaded in the working directory
- How to start a new branch: `git branch <branch>`
- How to switch on a new branch: `git checkout <branch>`
- How to display the history of all branches using `git log --all` ou `gitk --all`
- How to merge branches
- Some manipulations with remote branches

4. Advanced usage

4.1. Rebasing

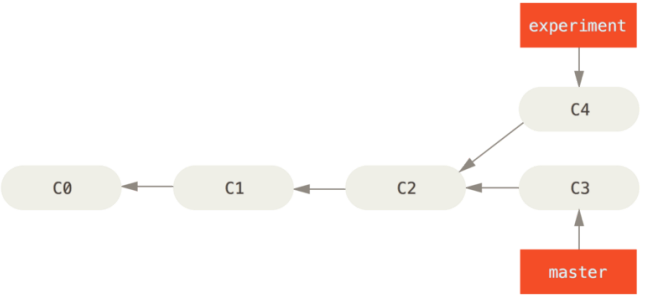
4.2. Debugging using Git

4.3. Worktrees

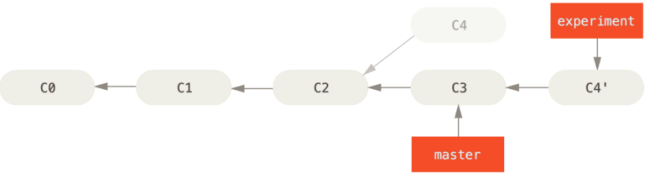
Rebasing

- It is another way merge changes in a branch
- Rebase **applies** successively each commit of a branch on another branch
- To merge a branch in the current branch, use `git rebase <branch>`

Consider this starting point:
experiment et *main* have diverged




```
git checkout experiment
git rebase main
```

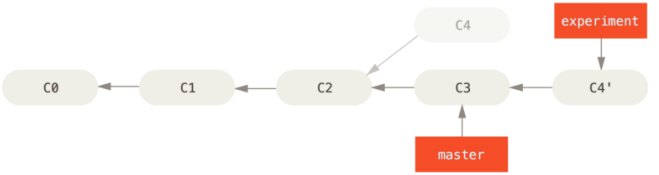


→ The rebase created a new commit, *main* is now the starting point of '*experiment*

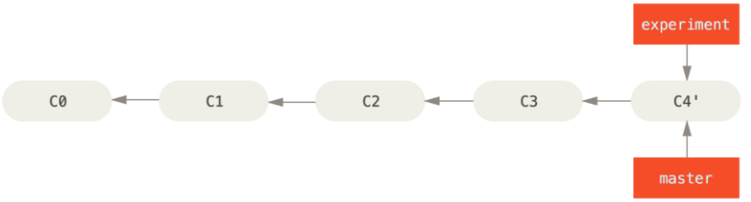
Rebasing principle

→ It's now possible to fast-forward *main* to *experiment*:

experiment rebased on *main*




```
git checkout main
git merge experiment
```



Main interest → we end up with a **linear history** of `main`, which is IMHO cleaner

→ Rebasing is also useful to *cleanup* the history of commits

Drawbacks of rebasing et workarounds

- Rebase *applies* each commit of a branch on the current branch
  → all commits applied have changed
  → we get a **different branch**, as it's now pointing at a different commit
- Compared to its remote branch, the rebased branch has **diverged** and merging it is hard
- To avoid problems:

  Do not rebase commits that exist outside your repository and that people may have based work on (git book advice)

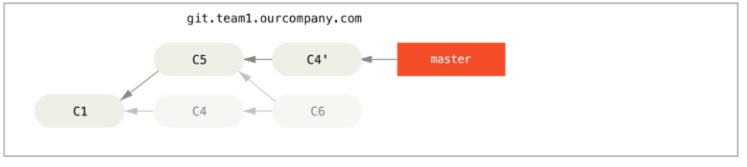
  … unless you are confident and know what you are doing (and nobody added local changes on the same branch)

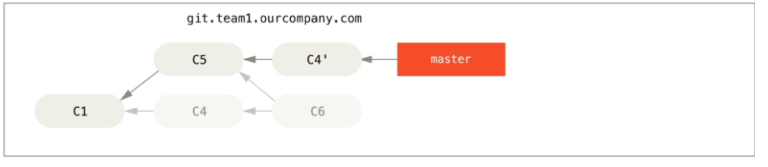
  → *Personal advice* : you can rebase an already pushed branch if you are the only one working on it. But you'll have to overwrite the remote branch

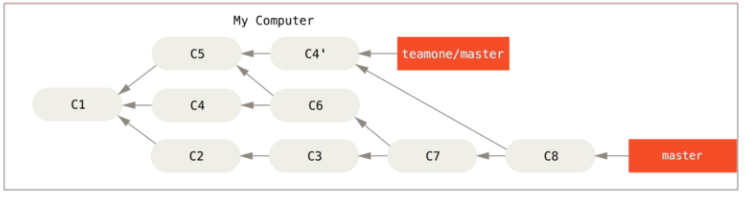
```
$ git push <remote> <rebased branch> -f  # you have to force the push
```

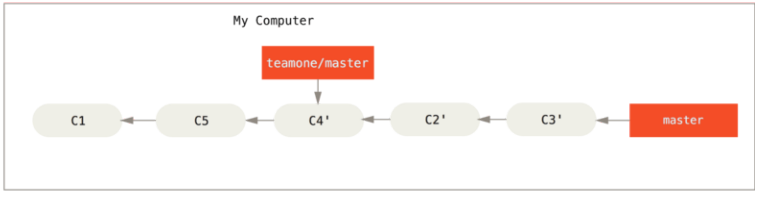
Interest of rebasing

- Improve the history of commits → each commit is meaningful
- Reflect better the global history of the project









History using merge

Linear history using rebase

Typical rebase workflow

Example:

```
$ git checkout main              # current branch is main
$ git pull origin main           # sync main
$ git checkout -b new_feature    # start a new feature branch
.
...    # lots of new commits
...    # in the meantime main also evolved eventually
.
$ git pull origin main --rebase    # automatic rebase on
.                                  # latest main
$ git push origin new_feature      # the new_feature branch
.                                  # is up-to-date with main
.                                  # => allows fast-forward
```

How to solve rebase conflicts

- Rebase successively **applies** each commit of a branch on the current branch → conflicts can happen at each step:

```
$ git rebase conflicting_branch
First, rewinding head to replay your work on top of it...
Applying: modification dans branche_conflit
Using index info to reconstruct a base tree...
M    README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
```

How to proceed:

```
$ git rebase branch  # rebase on branch
. as long as there are conflicts:
$ git mergetool   # fix the conflict
$ git rebase --continue
```

Abort while rebasing: `git rebase --abort`

Change history

- Rebase successively **applies** each commit of a branch on the current branch
  → it is possible to change the order, choose the commits to apply
- Use `-i` flag to use the *interactive* mode
- Then a terminal interface will open which allows to:
  - *edit*: modify the commit message
  - *remove*: remove a commit from the history
  - *squash*: 2 commits are merged together
  - *move*: the order of commits is changed
- `git rebase -i <commit-hash>` applies commits after `<commit-hash>`

Exercise

Following the README.md file in exercises/06-rebasing.

Debug using Git

- Git provides a tool to perform dichotomic search in commits → `git bisect`
- Principle:
  1. Initialization:

```
$ git bisect start        # initialize on the
.                         # current commit
$ git bisect bad          # tell git it is bad
$ git checkout      # load working commit
$ git bisect good   # tell git it is ok
```

  2. Then alternate `git bisect good/bad` depending on the state of the proposed commit
  3. Terminate using `git bisect reset` to switch back to the initial commit
- The whole workflow can be automated:

```
$ git bisect start
$ git bisect run
$ git bisect reset    # once done
```

Exercises: debug your code using git bisect

Follow the README.md file in exercises/07-bisect

Check out multiple branches using worktree

- `git worktree` allows you to manage multiple working trees attached to the same repository
- *worktrees* have additional metadata to differenciate them from other worktrees
- `git init` or `git clone` creates the *main* worktree
- Added worktree are called *linked* worktree
- Adding a worktree:
  - `git worktree add ../<branch>` → creates a worktree in `../<branch>` path with a new branch called `<branch>`
  - `git worktree add <path> <branch>` → creates a worktree from an existing branch
- Once done with a worktree, it can be removed: `git worktree remove <worktree_name>`
- Use `git worktree list` to get the list all worktrees

Summary

In this section, we've seen several advanced but very useful usage of Git:

- The rebasing to avoid merge commit and to keep a linear and clean history
- How to change the history of commits using interactive rebase, e.g. `git rebase -i`
- How to search in the history of commits a change that introduced a bug
- How to checkout multiple branches at the same using `git worktree`

5. Working on GitHub/GitLab

5.1. Possible Git workflows

5.2. Using forks

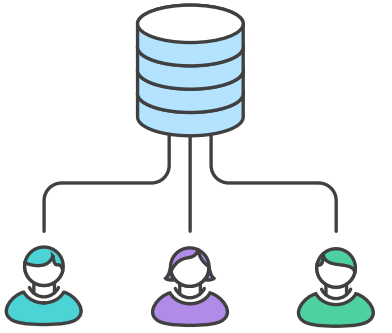
5.3. Overview of GitLab/GitHub

5.4. Opening a Pull-Request/Merge-Request

5.5. Code reviews

Possible Git workflows: NoFlow

All developers are working on the same repository

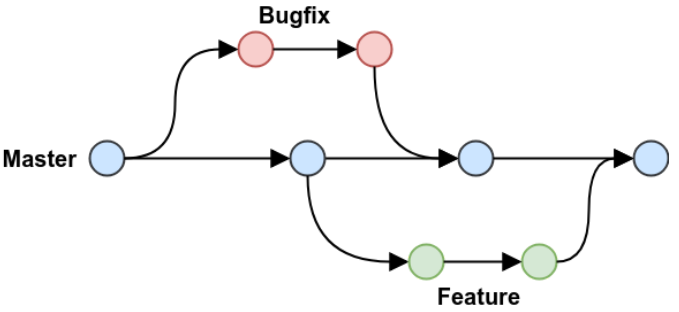


All developpers are working on the same branch

Several problems:

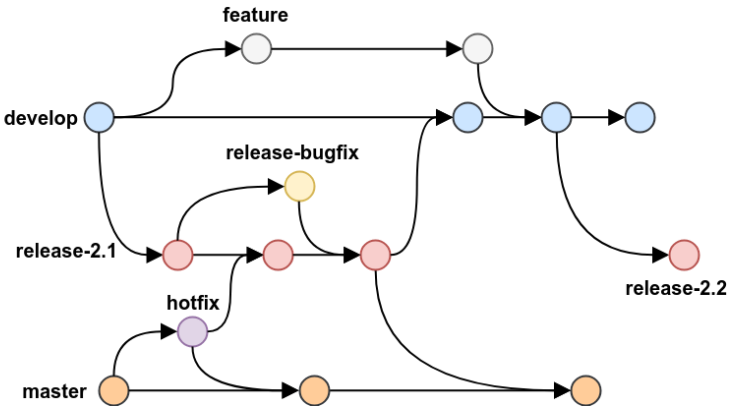
→ All developers need write (e.g. push) access to the main repository
→ Requires conventions on branch name to avoid name clash
→ Doesn't scale to large teams

Possible Git workflows: GitHub Flow



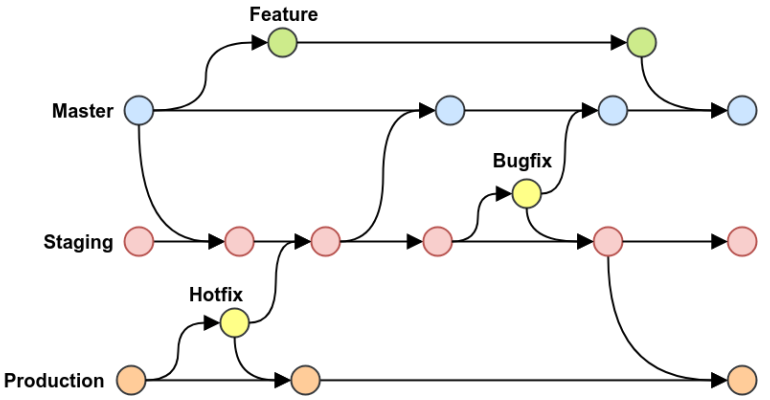
- New work is started by branching from main
- Once done, work is reviewed and tested before merging in main
- Simple and allows to release frequently

Possible Git workflows: GitFlow



- Suited for projects that have a scheduled release cycle
- Main is your rolled out production code with tagged versions
- Only hotfix, and release branches get merged into main
- Feature branches are merged into develop
- Only bugfixes, not new features, are merged into release branches

Possible Git workflows: GitLab Flow



- Ideal workflow for organizations that need to release frequently
- Base workflow similar to GitHub Flow
- `main`: everyone's local development environment line
- `staging`: where `main` is merged into for last minute tests before going to production
- `production`: tagged production code that staging is merged into

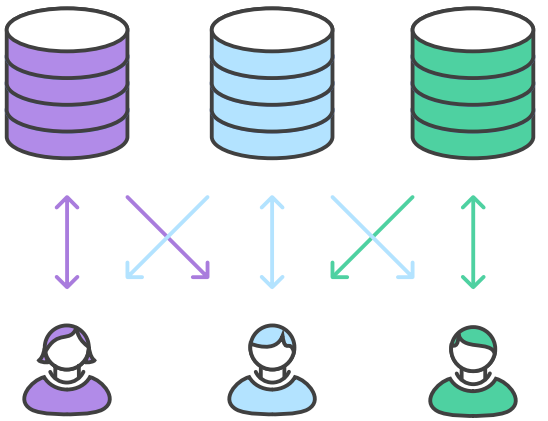
Possible Git workflows

Some useful references:

- https://blog.programster.org/git-workflows
- https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow
- https://betterprogramming.pub/a-simple-git-repository-strategy-93a0c7450f23

Using forks

- A main remote repository where only maintainers have write access
- Each developer owns 2 repositories:
  - A local *private* repository → its local copie
  - A remote repository → the remote server which contains a fork of the main repository
- Each developer publish branches to its remote repository
- Maintainers can push (or merge) branches to the main repository



Advantages of forks

- Provides more flexibility
- Scales better to large teams
- Third-party contributions are possible without push rights
- Each developer manages its own fork at will

Working workflow with forks

Let's go contribute to a new project!

1. I clone the upstream repository locally:
   `git clone <upstream url>.git`

2. I fork the official upstream repository using the web UI (GitHub/GitLab/Gitea)

3. I add my fork as a remote:
   `git remote add <my fork name> <fork url>.git`

4. I create new branches for my various features (`git checkout -b <branch>`). Once done, I check them (self-review, test, iterate).

5. I eventually rebase locally on the latest upstream `main`:
   `git pull upstream main --rebase`. Once done, I check them (self-review, test, iterate).

6. I push my branch(es) to my fork: `git push origin <branch>`

7. I open a *Merge/Pull Request* explaining what my change is about and wait for CI and reviews

Some advice

- You can use the following remote name conventions:
  - the remote of my fork is called `<my username>`
  - the upstream remote is called `origin`
- I almost never use the *merge* command, only *rebase* → **I'm alone on my fork!**

Example:

```
$ git checkout my_branch     # my working branch
$ git fetch origin           # I sync with upstream
$ git rebase origin/main     # make sure my working branch
.                            # is based on latest main
$ git push origin my_branch  # -f if it was already pushed
```

- Never create a pull request using the `main` branch of your fork targetting the `main` branch upstream
- One branch per feature/fix → focused changes are easier to review and this keeps your work clean!

Using GitHub and GitLab

Prerequisites: create an account and add its public SSH key

Demonstration:

- GitHub
- GitLab

Exercises: Working on GitLab

Follow the README.md file in exercises/08-gitlab

Exercises: Working on GitHub

Follow the README.md file in exercises/09-github

Summary

In this last section, we've learned:

- The possible working workflows with Git
- In particular, how to use the forks, with some personal advices
- How to contribute on GitLab and on GitHub
- We opened merge requests on GitLab and briefly went through code reviews

Thank You!

Detailed schedule (1):

1. Introduction

    1. About version control systems (VCS)
    2. History
    3. Git features
    4. How it works
    5. The 3 states
    6. Structure of a Git repository

2. The basic commands

    1. Configuring Git
    2. Initializing a new repository
    3. Adding changes
    4. Tracking the history of changes
    5. Reverting changes
    6. Working with remotes

Detailed schedule (2):

1. Working with branches
    1. What is a branch?
    2. Starting a new branch
    3. Switching branches
    4. Merging branches
    5. Resolving conflicts
    6. Remote branches
2. Advanced usage
    1. Rebasing
    2. Debugging using Git
    3. Worktrees
3. Working on GitHub/GitLab
    1. Team working: possible workflows
    2. Using forks
    3. Overview of GitLab/GitHub
    4. Opening a Pull-Request/Merge-Request
    5. Code reviews