Git for developers

Source code management with *git*

Agenda

1. Git overview and base concepts
2. Getting started with git
3. Working with git
4. Organising the workflow
5. References
6. One last thing...

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Git presentation and history

- Git is a *distributed revision control* and *source code management* system
- Git is a free software created by *Linus Torvalds*
- Git is available under the *GNU GPL V2* license
- Git was initialy created to host the *Linux* kernel sources after the hosting on BitKeeper was abandonned
- Key dates:
  - 3 April 2005: Linus starts development
  - 16 June 2005: release 2.6.12 of the linux kernel managed by git
  - 26 July 2005: handover of maintenance to a major contributor (Junio Hamano)
  - 21 December 2005: release of v1.0 of git
  - 28 May 2014: release of v2.0 of git

Content

1 Git overview and base concepts
- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Git design criteria

- Strong support for non-linear development:
  - rapid branching and merging
  - *core* assumption: a change will be merged more often than it is written
- *Distributed* development and workflow
- Strong safeguard against corruption (accidental or malicious)
- Compatibility with existing systems/protocols (http, ftp, rsync, ssh)
- Efficient handling of large projects
- *Object Oriented* model
- *Plumbing* vs. *Porcelain*

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Other SCMs

- Centralized (client/server):
  - Concurrent Version System (CVS)
  - Subversion (svn)

Distributed:

  - Mercurial
  - GNU Bazaar
  - BitKeeper

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Distributed system benefits and drawbacks

- Benefits:
  - Full history available locally
  - No need for network connection for usual operations
  - No need for manager approval for day to day usage (branch, tag, merge ...)
  - There is still a reference repository
  - You can clone the reference repository or any existing clone
  - Merge system (vs. lock one)
- Drawbacks:
  - Initial cloning can be long (full history)
  - No lock system (can be a problem for binaries)

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Differences with Subversion

- Distributed (vs. centralised)
- Use of meta data (vs. files)
  - only one `.git` folder at root level (this point has changed with last versions of svn)
- Better data integrity (usage of *SHA-1* hash to identify each object)
- No global revision in git
- Better branch management
- “Real” tags which can be signed with *GPG* keys if needed
- The system stores full contents as opposed to delta (diff)

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Data integrity control

- Usage of *SHA-1* hashes (same result on any machine/system for the same content)
- 160 bits footprint (40 chars.)
- All files are stored in `.git/objects`
- Files are identified by an ID: fast and efficient

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

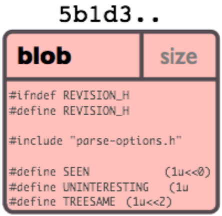
Git Object Model

There are 4 types of objects in Git:

- Blob
- Tree
- Commit
- Tag

Git Object Model: Blob

A blob object stores the content of a file. It's a piece of binary data which does not reference anything, not even a file name.



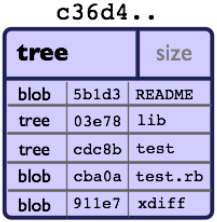
```
#ifndef REVISION_H
#define REVISION_H

#include "parse-options.h"

#define SEEN              (1u<<0)
#define UNINTERESTING   (1u
#define TREESAME  (1u<<2)
```

You can
inspect the content of a blob if you know its hash with the command:

```
git show <hash> / git cat-file -p <hash>
```

Git Object Model: Tree

A tree object contains a list of pointers to *blobs* and other *trees*. It represents the content of a directory or sub-directory.



c36d4..

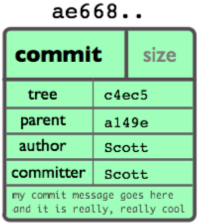
tree		size
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

You can use **git show** to inspect a tree but **git ls-tree** will give more info:

```
git ls-tree <hash> / git show <hash>
```

Git Object Model: Commit

A commit links the physical state of a *tree* with a comment explaining why it came to that state. It is defined by a tree, one or several parents, an author, a committer and a comment.



ae668..

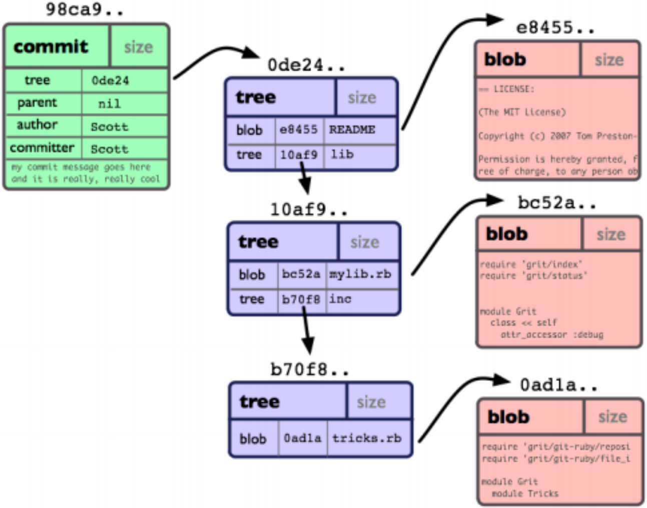
commit	size
tree	c4ec5
parent	a149e
author	Scott
committer	Scott

my commit message goes here
and it is really, really cool

You can use **git show** or **git log** with the **–pretty=raw** option to inspect a commit:

```
git show --pretty=raw <hash> / git cat-file -p <hash>
```

Git Object Model: Commit/Tree/Blob relations



98ca9..

commit	size
tree	0de24
parent	nil
author	Scott
committer	Scott

```
my commit message goes here
and it is really, really cool
```

0de24..

tree	size	
blob	e8455	README
tree	10af9	lib

10af9..

tree	size	
blob	bc52a	mylib.rb
tree	b70f8	inc

b70f8..

tree	size	
blob	0ad1a	tricks.rb

e8455..

blob	size

```
== LICENSE:

(The MIT License)

Copyright (c) 2007 Tom Preston-

Permission is hereby granted, f
ree of charge, to any person ob
```

bc52a..

blob	size

```
require 'grit/index'
require 'grit/status'


module Grit
  class << self
    attr_accessor :debug
```

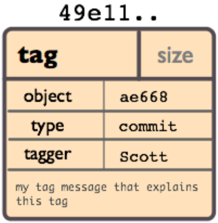
0ad1a..

blob	size

```
require 'grit/git-ruby/reposi
require 'grit/git-ruby/file_i

module Grit
  module Tricks
```

Git Object Model: Tag

A tag object points to a specific state of the code. It contains an object id, an object type, the name of the tagger and a message with an eventual signature.



You can use **git cat-file** to view the content of a tag:

```
git cat-file tag <TAG_NAME>
```

Content

1 Git overview and base concepts

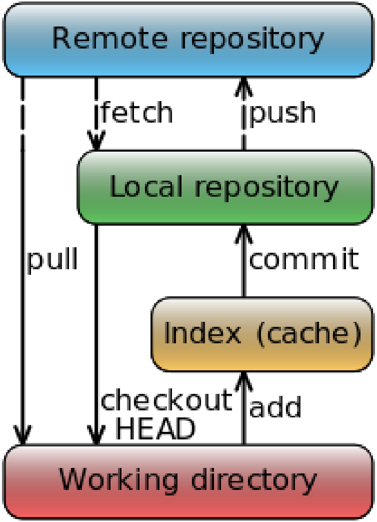
- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Understanding git architecture: Working areas

There are basically 4 areas you have to work with:

- The *working directory*: it holds your project files ; this is were you make changes.
- The *staging area*: this is were you **add** the snapshot of the files you want to commit to your repository.
- The *local repository*: this is were you **commit** your changes ; this is held in the `.git` directory.
- The *remote repository*: this is the reference that you use to **pull** the changes made by other developpers and **push** your own changes ; this repository is *bare* (it contains no working copy) ; in Smile context, this will be a GitLab repository on `https://git.smile.fr/` that you access over http or ssh.

Understanding git architecture: Operations overview



Understanding git architecture: Anatomy of the .git folder

Every working directory contains a single `.git` file at its root containing the local git repository.

`.git`

`config`	Your local preferences for this repository
`description`	Description of your project
`HEAD`	Pointer to your current branch
`hooks/`	Pre/Post action hooks
`index`	Staging area content
`info/`	Additional info on the repo
`logs/`	History of your branch
`objects/`	Your objects (commits, trees, blobs, tags)
`refs/`	References to your branches and tags

To go further see:

```
git help repository-layout
```

Understanding git architecture: Review of git hooks

Hooks samples are embedded in any git repository in `.git/hooks`. A hook is basically a shell script in any language (*shabang* / *shebang* usage).

`.git/hooks`

`applypatch-msg.sample`	local
`commit-msg.sample`	local
`post-update.sample`	server
`pre-applypatch.sample`	local
`pre-commit.sample`	local
`pre-push.sample`	local
`pre-rebase.sample`	local
`prepare-commit-msg.sample`	local
`update.sample`	server

Many more hooks exist. To go further see:

```
git help hooks
```

Content

1 Git overview and base concepts

- Git presentation and history
- Git design criteria
- Other SCMs
- Distributed system benefits and drawbacks
- Differences with Subversion
- Data integrity control
- Git Object Model
- Understanding git architecture
- Plumbing vs. Porcelain

Plumbing vs. Porcelain

Git was first made as a VCS toolkit which embeds a large set of *low-level* commands to interact with the `.git` directory contents (store contents, hash objects, create an index, see git objects contents ...). These commands are called *plumbing commands* as they mostly concern the low-level internal processes.

- **git cat-file** / **git symbolic-ref** / **git rev-parse**/ **git update-index** / **git read-tree** / **git write-tree** ...

Git also comes with a user-friendly command line interface for common usage, which is a set of *user-level* commands. They are mostly shortcuts for *plumbing* ones to facilitate a day-to-day usage. These commands are called *porcelain commands* as they represent the user-friendly couch of git.

- **git push** / **git pull** / **git status** / **git log** / **git add** / **git commit** ...

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Installing git and needed tools

- Linux debian/ubuntu

  ```
  apt-get install git git-svn gitk meld
  ```

- Windows
  - Git for window (https://git-for-windows.github.io/)
  - (optionnally) tortoiseGit (https://tortoisegit.org/)
- Mac
  - http://code.google.com/p/git-osx-installer
  - http://www.macports.org

    ```
    sudo port install git-core +svn +doc \
        +bash_completion +gitweb
    ```

- Eclipse
  - egit

Installing git and needed tools: Practice

- Install **git** on your machine
- Make sure the **git** command is available in your terminal
- Check which **git** version is in use

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Getting help

- For all sub-commands, git integrates a detailed help accessible with:

```
git help <command>
```

- All configuration variables can be inspected with a good detail level with command:

```
git help config
```

- Most commands and references can be autocompleted on the commande line, so do not hesitate to hit the TAB key:

```
git [TAB]             # list of commands
git cmd [TAB]         # list of options
git cmd -o [TAB]      # list of remotes/branches
```

Getting help: Practice

- Look at global help
- Use autocompletion with the **help** command (type the first letter of a git subcommand)

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Configuration

- Git can be configured via the command line:

```
git config (--local | --global) scope.var "value"
```

- You can also edit configuration files (`INI` like format):
  - file `~/.gitconfig` for global per user configurations (like using option –**global**)
  - file `.git/config` for local repository configurations (like using option –**local**)

Configuration: Practice

- Have a look at available config variables
- Have a look at your *global* and *local* config values

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Telling git who you are

Before you start any work in git, you must set your identity correctly. It will be used to identify your as an author or committer:

```
git config --global user.name "Olivier Clavel"
git config --global user.email \
    "olivier.clavel@smile.fr"
```

Telling git who you are: Practice

- Configure your full name and email

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Setting editors

- You can choose your favorite editor as you wish. Default is **$EDITOR** (most probably **vi**). Examples:

```
git config --global core.editor nano
git config --global core.editor gedit
```

- You should as well define your diff editor. The recommended one is **meld** on linux:

```
git config --global merge.tool meld
```

- For other diff editor possibilities see:

```
git help mergetool
```

Setting editors: Practice

- Set your favorite text editor
- Set your favorite diff editor ; if you don't have one, install *Meld* and set it as default

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Colors in your terminal

- Git can colorize most of its output in the terminal for better readability. To globaly enable colors:

  ```
  git config --global color.ui auto
  ```

- For fined grained color tuning see all **color.*** config variables.

Colors in your terminal: Practice

- Have a look at color variables in config help
- Set default auto color for output

Content

2 Getting started with git

- Installing git and needed tools
- Getting help
- Configuration
- Telling git who you are
- Setting editors
- Colors in your terminal
- Know where you are in your bash prompt

Know where you are in your bash prompt

The git package comes with **PS1** macro your can add to your prompt. It will display the current branch if your are inside a git repository. Here is an example customized from a default `.bashrc` under Ubuntu. Adapt it to your needs.

```
# Find this line in your .bashrc
if [ "$color_prompt" = yes ]; then
    PS1='${debian_chroot:+($debian_chroot)}
        \[\033[01;32m\]
        \u@\h\[\033[00m\]:\[\033[01;34m\]
        \w$(__git_ps1 " (%s)")\[\033[00m\]\$ '
else
    PS1='${debian_chroot:+($debian_chroot)}
        \u@\h:\w$(__git_ps1)\$ '
fi
unset color_prompt force_color_prompt
#.bashrc continues
```

Know where you are in your bash prompt: Practice

- Make sure the **PS1** macro is available for your platform (hint: see `/etc/bash_completion.d/` directory)
- Locate your **PS1** prompt definition (hint: have a look at `~/.bashrc`)
- Make a backup copy of that file
- Try to modify your **PS1** prompt
- Run **mkdir test && cd $_ && git init**
- See what happens on your prompt

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Creating a local repository

- Initialise an empty local repository

```
mkdir my_project
cd my_project
git init
```

- Clone an existing remote repository

```
git clone \
    git@git.smile.fr/my_group/my_project.git \
    my_project_local_dir
```

Creating a local repository: Practice

- Basic set up of your Gitlab account (preferences, ssh key ...)
- Create a folder for today's training
- Move to that folder
- Initialise a git dir from scratch and copy some files in there
- Create another repo by cloning the example for today's training

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Adding files to the staging area

Once you have modified some files in your workdir, you must add the changes to the staging to prepare them for commit.

- First have a look at what has changed:

```
git status
git diff
```

- You should *always over-use these commands* (with options eventually)
- Add some files to the staging area:

```
git add [file | folder | ...] [file2 | folder2 | ...]
```

- For partial add operations, use the patch mode:

```
git add -p / git add --patch
```

- For fine tuning of add operation, use the interactive mode:

```
git add -i / git add --interactive
```

Adding files to the staging area: Practice

- Work on your own empty created repo
- Examine the state of your dir
- Add some more files or make some changes if needed
- Add some changes to your staging area
- Add an external library
- Use the **git add** *interactive* and *partial* modes to add the library to the index

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- **Listing differences**
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Listing differences

- See differences between your working copy and the staging area:

```
git diff
```

- See differences between your working copy + staging area and the last commit on your branch:

```
git diff HEAD
```

- See differences between your staging area and the last commit:

```
git diff --staged
```

Listing differences: Practice

- List differences for files you did not yet add
- List differences already added to your staging area

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Commiting changes and viewing history

To create a new commit with the content of the staging area, use **git commit**.

- With no option, it will launch your external editor to let you write commit's message:

  ```
  git commit
  ```

- Specify the message on the commmand line:

  ```
  git commit -m "my commit message"
  ```

- You can add all changes to the staging and commit them at once:

  ```
  git commit -a -m  "my commit message"
  ```

- You can modify a commit:

  ```
  git commit --amend
  ```

Commiting changes and viewing history

To look at history of commits, use **git log**.

- Full default history:

```
git log
```

- Show history since one minute ago:

```
git log -p --since="1 minute ago"
```

- Show history of last 2 commits:

```
git log -p HEAD~2..HEAD
```

Commiting changes and viewing history: Practice

- Commit your staged changes
- Add some more changes and commit them
- Have a look at the history
- Get help on git show and examine some commits with it

Commiting changes and viewing history: Use case

Problem: how to update a commit with new files or modifications.

```
# initial commit with forgotten modifications
git commit -m "commit message"
> [master 1234567] commit message
> ...

# add missing new files or modifications
# to the staging area
git add <path>

# redraw the last HEAD commit
# with last commit message
git commit --amend -C HEAD
> [master 7654321] commit message
> ...
```

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

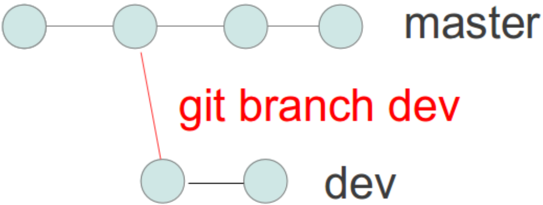
Creating branches

- By default the repository contains the master branch
- To create a dev branch based on the latest commit of the current one:

```
git branch dev
```

- To switch on the newly create branch:

```
git checkout dev
```



Creating branches: Practice

- Create a **dev** branch from **master**
- Switch to **dev** branch
- Make some changes, add, commit them
- Switch back to **master**
- List differences between **master** and **dev** branches (hint: get help on diff)
- Get help on the checkout command and figure out how to create and switch to a branch named **experimental** in a single command
- Make some changes in branch **experimental**, commit them

Content

3 Working with git

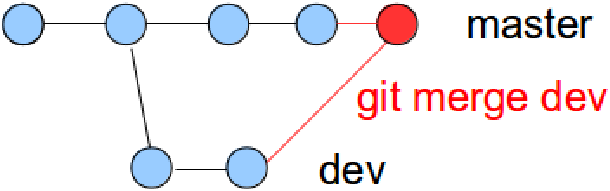
- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- **Merging branches**
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Merging branches

- To merge the changes from one branch into another (e.g. dev into master):

```
git checkout master
git merge dev
```

- Branches can be merged multiple times. Git keeps trace of the merge history.



Content

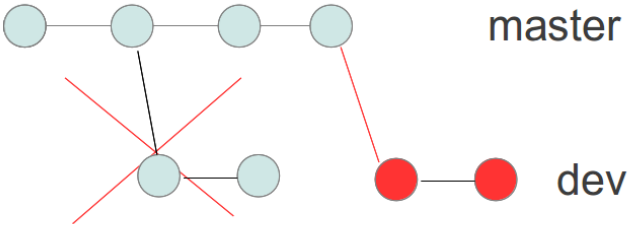
3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Rebasing

- You can rebase one branch on the work done on another.
- When you rebase, you are basically adding your work after the one done on the other branch

```
git pull --rebase
git rebase master
```



Merging and Rebasing: Practice

- Merge **dev** into **master** and show logs
- Rebase **experimental** on **master** and show logs
- Merge **experimental** into **master** and show logs
- Delete branch **experimental** (hint: git help on branch)

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Cherry-picking

- You can pick a single commit from one branch and insert it in another:

```
git cherry-pick <hash>
```

- When you *cherry-pick*, the original commit is patched in current history and a new commit is done
- You can compare commits between branches:

```
git cherry -v master dev
```

Cherry-picking: Practice

- Checkout the **dev** branch, make some updates and commit them
- Checkout **master** and list commits diff
- Cherry-pick your commit from **dev** to **master**
- List commits diff again

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Sharing work with the remote repository

- If you have cloned a remote, git already has the master as a remote tracking branch. Else you need to add the remote, publish the branch and set it as upstream:

```
git remote add origin \
    git@git.smile.fr:my_group/my_project.git
git push --set-upstream origin master
```

- You proceed the same way with new branches you want to publish and track:

```
git push -u origin dev
```

- To retrieve changes pushed by others on the remote, you must fetch them and merge them in your branch:

```
git pull
# or in two steps
git fetch && git merge
```

Sharing work with the remote repository

- In most cases, on long living branches, you will want to rebase your work on the remote branch:

```
git pull --rebase
```

- To rebase automatically for a specific branch each time you pull:

```
git config branch.master.rebase true
git pull
```

- To rebase automatically for all new branches (add **–global** to have it permanent on the machine):

```
git config branch.autosetuprebase always
```

Remote interaction: Practice

- Create a **training** repository under your name on gitlab. *Please remember to clean it up after the training. Thanks :)*
- Add this new remote as **origin** on your local home made repository
- Push **master** on origin and set origin/master as remote tracking branch
- Move to the example clone repository
- Config **master** to always rebase on upstream on pull
- Push your changes (if you can)
- Pull changes from others

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- **Working with tags**
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Working with tags

- Tags are used to identify a specific version of a specific branch, typically to identify a version pushed to production
- Tags can be organised with prefixes like for folders on a disk
- You can create a lightweight tag (default):

```
git tag production/20140116_0902 master
```

- Or a full tag object, even with a *GPG* signature:

```
git tag -s -m "pushing release XXX to \
    uat for testing" uat/20140112_1735 master
```

- Signed tags can be verified:

```
git tag -v uat/20140112_1735
```

Working with tags: Practice

- Create some tags
- Push those tags to the **origin** remote (hint: help push)

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Moving and deleting files

- Delete files (must commit after to send to repo):

```
git rm file1 file2
```

- Rename or move a file (must commit after to send to repo):

```
git mv old_file new_file
```

Moving and deleting files: Practice

- Delete and move some files arround
- Commit and push your changes

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Ignoring files

- Personnal ignore file globally applied on your machine:

```
git config --global core.excludesfile \
    /home/<USER>/.gitignore
vim /home/<USER>/.gitignore
```

- Local and personnal ignore file for a specific repository:

```
vim .git/info/exclude
```

- Local shared ignore file commited in the repository:

```
cd my_project
vim .gitignore
git add .gitignore
git commit -m "Ignoring y and z in project"
```

- Those files all contain list of *GLOBS* file patterns to exclude

Ignoring files: Practice

- Exclude your favourite backup extension globally for all your git repositories
- Exclude `.phps` locally for your current repositories
- Create a `cache` and `log` directories and ignore all their contents for anyone working on the project

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Going back to a previous state

- Remove a change in staging keeping the working file unchanged (opposite of **add**):

```
git reset -- path/to/file
```

- Discard all changes come back to the HEAD revision in repository:

```
git reset --hard HEAD
```

- Recover a file accidentaly deleted:

```
# recover from HEAD revision in repository
git checkout HEAD path/to/file
# recover from staging area
git checkout -- path/to/file
```

- Repair an error that was commited to the repository. This creates a new commit undoing the change. In this example only undo last commit:

```
git revert HEAD
```

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- **Include submodules**
- Git attributes

Include submodules

Git comes with the ability to include another git repository in a project treating it as a git clone. It only stores the remote URL of the module and the commit you want to use in your project:

```
# add a new submodule
git submodule add /URL/of/git/submodule/ \
    local/submodule/path/
# update the submodule
cd local/submodule/path/
git pull --rebase origin master
# commit the state you want for your submodule
cd -
git add local/submodule/path/
git commit -m "update submodule to XXX"
```

Content

3 Working with git

- Creating a local repository
- Adding files to the staging area
- Listing differences
- Commiting changes and viewing history
- Creating branches
- Merging branches
- Rebasing
- Cherry-picking
- Sharing work with the remote repository
- Working with tags
- Moving and deleting files
- Ignoring files
- Going back to a previous state
- Include submodules
- Git attributes

Git attributes

Git comes with the ability to define special behaviors:

- Behavior for a file type (such as identation, binary contents)
- Exclude some files or directories from tarballs
- Define a merge strategy for specific files or directories

```
# get help about available attributes
git help attributes
# setup a local .gitattributes file
vim .gitattributes
# setup a global .gitattributes file
vim ~/.gitattributes
```

Git attributes: Practice

- Add a submodule to project and commit and push it
- Have a look at it at GitLab
- Setup git to consider any `*.html` file as a text file
- Exclude any `*.tmp` file from tarball exports

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Workflow basics

- Git branches are light and easy to manage: *use them !*
- You do not have to push all branches you create locally
- Type of branches on the remote:
  - *Long living* or Environment branches
  - *Short living* or Topic/Feature branches
- Topic branches are usually derived from **master** and eventually rebased against it (dependencies, long term work)
- Topic branches are merged in Environment branches. Merge to higher level, never backwards.
- Topic branches are created from *development* branch, and rebased against it.
- Git is only the repository tool. *It does not replace procedures, planning and communication !*

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Environment branches

- A *master* branch should always exist:
  - It must be the *production-ready* sources
  - You should always be able to deploy it on production (no development or unstable stuff)
  - Best practice: any commit on master should create a new tag (version) or deployment
- A *development* branch should always exist:
  - It must centralize all developments to be tested before including them on **master**
  - It must always be ahead **master** (or at least at the same point)
  - It receives merges from Topic/Feature branches
- Any other environment branches can exist

Environment branches: examples

- master = production
- dev (=continuous integration ?)
- uat
- staging
- preprod
- ...

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Feature branches

- Topic or Feature branches have a short life
- They should be up-to-date with current **master** (at least) by merging it regularly
- They must be merged in the *development* branch when the work is done
- Once fully integrated in environment branches, they can be deleted
- Remember: you don't have to push all your local branches to the remote

Feature branches: examples

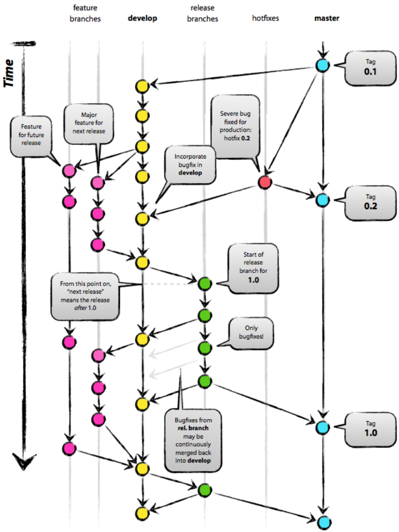
- feature-12345 (redmine ticket)
- checkout_redesignV2 (project)
- local_cutting_edge_newfeat
- hotfix-654677 (redmine ticket)

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Branches organisation overview


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- **Tips & tricks**
- Git-flow tool
- Workflow practice

Tips & tricks

- When merging a short-living branch, use the **–no-ff** option to force git to make a merge commit:

  ```
  git merge --no-ff my-feature-branch
  ```

- Use the “fork” and “pull-request” concepts of GitLab (and GitHub)

Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Git-flow tool

- Git-flow provides some function to simplify branching and merging

- Start a new feature:

```
git flow feature start my-feature
```

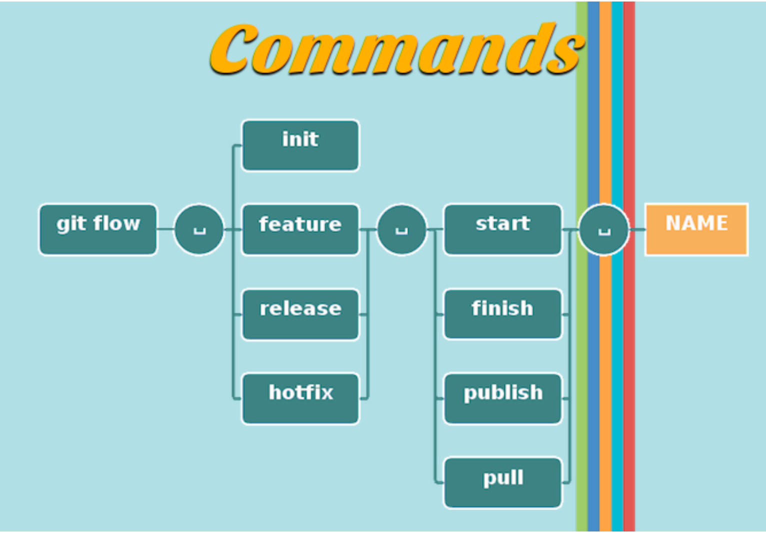
- Finish up a new feature:

```
git flow feature finish my-feature
```

- Publish a new feature:

```
git flow feature publish my-feature
```

Git-flow tool



Content

4 Organising the workflow

- Basis
- Environment branches
- Feature branches
- Branches organisation overview
- Tips & tricks
- Git-flow tool
- Workflow practice

Workflow practice

Examples:

- Teams constitution: define roles and tasks
- Create long living branches
- Create feature branches (one for each team)
- Work separately on features
- Deliver individual developements up to production in different orders
- Clean up local and remote references (hint: git help push/branch)
- ... and any other operations you want to try and discuss :)

Content

5 References

References

- Learn git online in 15 minutes on github: http://try.github.io/levels/1/challenges/1
- Visual git Cheatsheet: http://ndpsoftware.com/git-cheatsheet.html
- Pro Git book: http://git-scm.com/book
- Git community book: http://alx.github.io/gitbook/
- Git - the simple guide: http://rogerdudler.github.io/git-guide/
- git-workflows for agilist: https://github.com/stevenharman/git-workflows
- git-flow: http://danielkummer.github.io/git-flow-cheatsheet/

Content

6 One last thing...

One last thing...

- How to "stash": `https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning`
- Rewriting history: `https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History`
- Using bisect debugging: https://git-scm.com/book/en/v2/Git-Tools-Debugging-with-Git
- Mono/many repos: https://speakerdeck.com/fabpot/a-monorepo-vs-manyrepos