

Formation GIT

Mettre en œuvre le contrôle de versions avec GIT



Bienvenue !

■ Présentation :

- Du formateur
- De DocDoku

■ A votre propos :

- Expérience avec les outils de versioning (VCS)
- Ce que vous attendez de ce cours
- Les projets sur lesquels vous allez travailler avec GIT

Informations pratiques

■ Horaires :

- Heure de début
- Heure de fin

■ Pauses :

- Matin : 15 minutes
- Repas : 1h30
- Après-midi : 15 minutes

Vous allez apprendre à...

- Comprendre les concepts de base de la gestion des versions et des apports de la décentralisation
- Installer et configurer l'outil Git
- Créer et initialiser un dépôt avec Git
- Manipuler les commandes de Git pour gérer les fichiers et les branches
- Mettre en œuvre un projet en mode collaboratif avec Git

Contenu de la formation



- 01 – Présentation de GIT
- 02 – Installation et configuration
- 03 – Utilisation de GIT, les fondamentaux
- 04 – Gestion locale des fichiers
- 05 – Gestion des branches
- 06 – Partage du travail et collaboration
- 07 – Mise en œuvre des outils GIT
- 08 – Impacts organisationnels de git

Présentation de GIT

Un peu de théorie

Présentation de GIT

- Concepts de base du contrôle de version
- La gestion centralisée ou distribuée
- Les différentes solutions de gestion de versions
- Principes et apports la décentralisation

Concepts de base du contrôle de version

Utilisé dans tout projet logiciel sérieux

- **Partage** du code
- **Communication**
- **Sécurité**

Totalement **indispensable** en entreprise

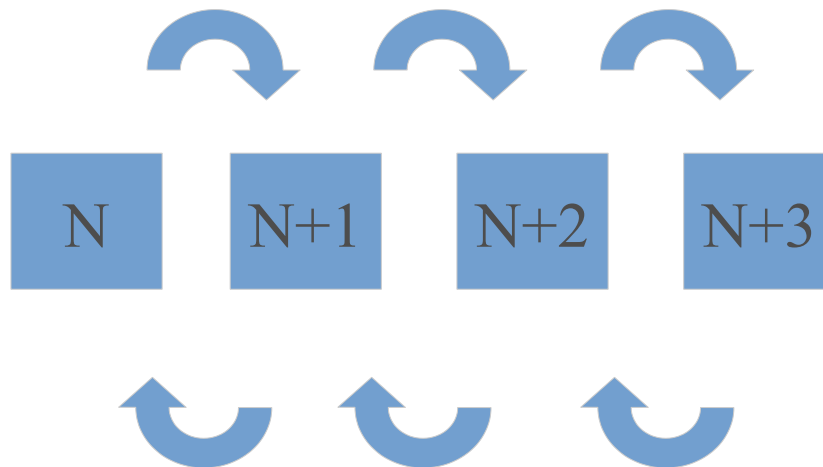
Concerne principalement la gestion du code source ...

.. mais pas seulement : fichiers binaires, base de données, systèmes de sauvegarde, etc...

Concepts de base du contrôle de version

Modification d'un fichier

- A **appliquer** ou à **retirer**
- Obtention de la **version désirée**
- **Ensemble** de modifications



Concepts de base du contrôle de version

Ces outils sont indispensables mais nécessitent une certaine rigueur.

Importance de la « **propreté** » de l'historique

- Quel que soit l'outil utilisé
- Facilite la **revue**
- Facilite les **échanges**
- Facilite la **compréhension** du projet

Présentation de GIT

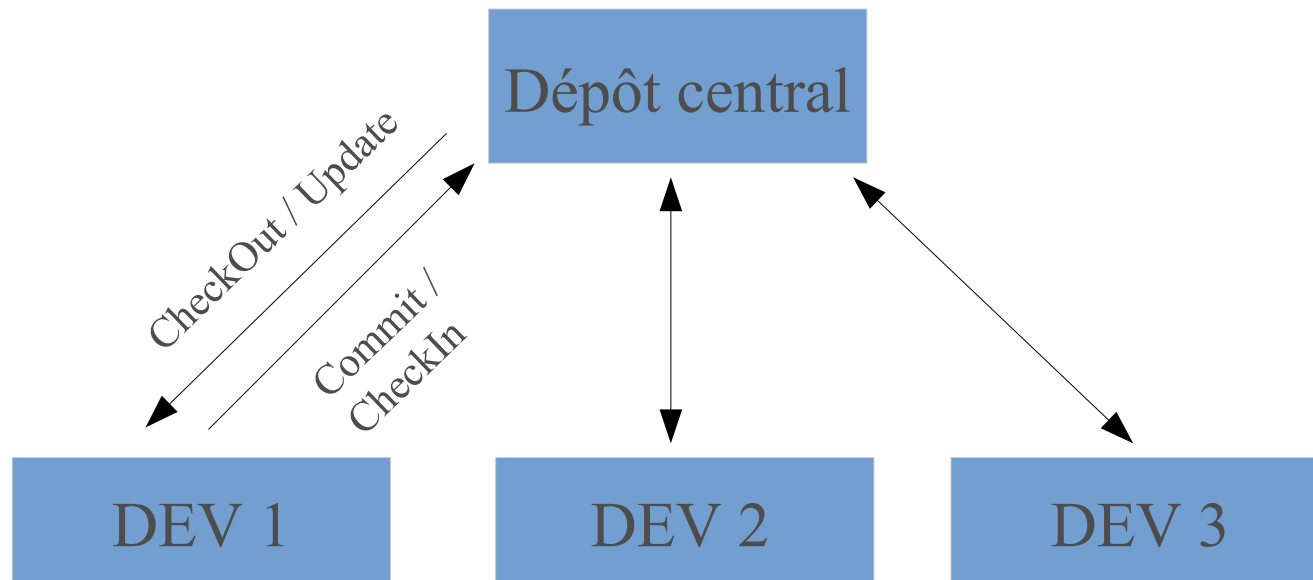
- Concepts de base du contrôle de version
- La gestion centralisée ou distribuée
- Les différentes solutions de gestion de versions
- Principes et apports la décentralisation

La gestion centralisée ou distribuée

Gestion centralisée : **dépôt unique sur un serveur**

S'y trouve :

- l'ensemble des fichiers et de leurs versions
- l'ensemble des branches



La gestion centralisée ou distribuée

Avantages et inconvénients de la gestion **centralisée**

Facilité de gestion, coût de mise en place

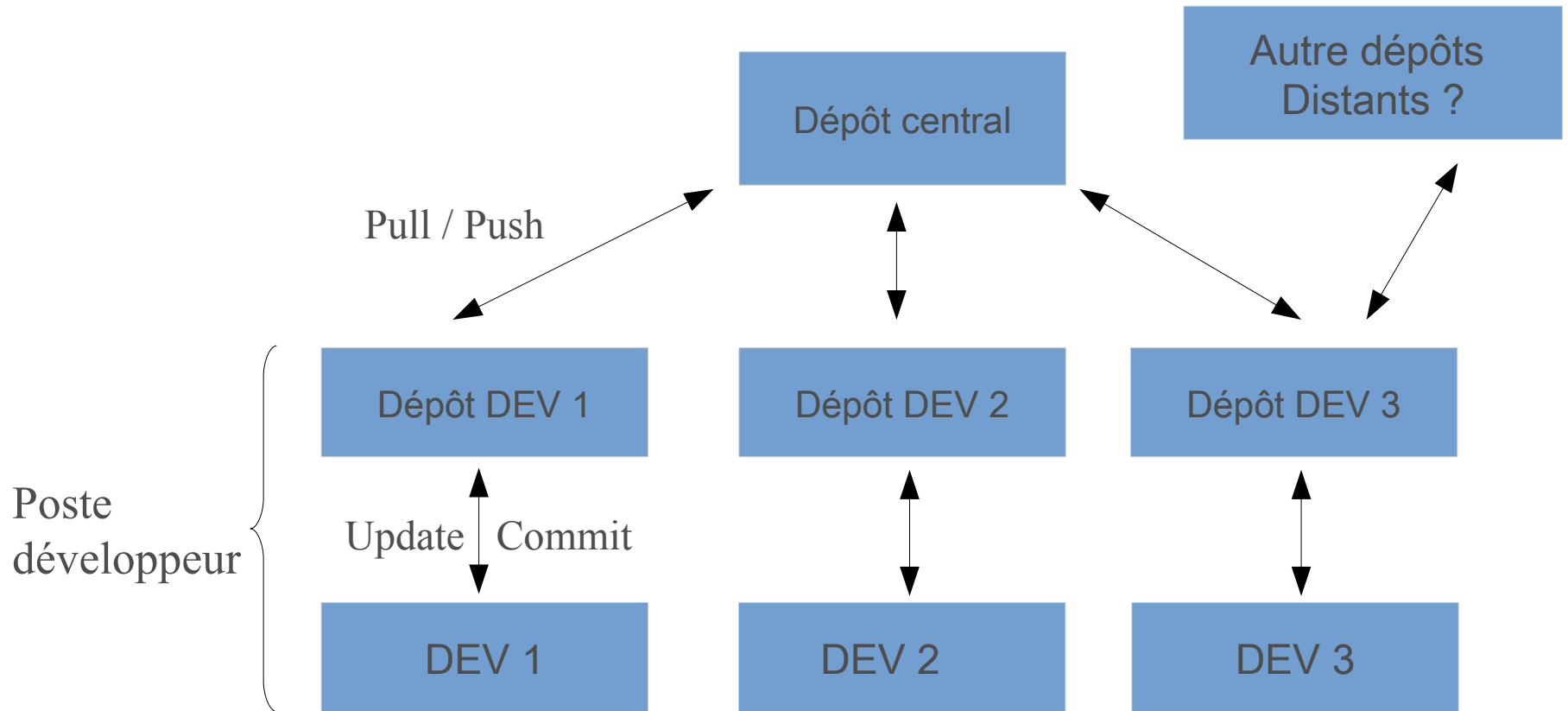
Mais ...

Si le dépôt tombe ou est inaccessible

- Pannes
 - Piratage
- Besoin d'une stratégie de sauvegarde et de restauration

La gestion centralisée ou distribuée

Gestion **distribuée** : chaque développeur a son dépôt qui contient toutes les versions



La gestion centralisée ou distribuée

Avantages et inconvénients de la gestion **distribuée**

Chaque dépôt peut « survivre » **seul**

Meilleure gestion de l'historique

Permet une utilisation « **offline** »

Mais ...

Complexité

Cela a un **coût** de mise en œuvre plus élevé

Montée en compétence des développeurs

Présentation de GIT

- Concepts de base du contrôle de version
- La gestion centralisée ou distribuée
- Les différentes solutions de gestion de versions
- Principes et apports la décentralisation

Les différentes solutions de gestion de version

Les principaux logiciels de gestion de version

Centralisés

- SVN (Libre - Apache - 2000)
- CVS (Libre - The CVS Team - 1990)
- TFS (Propriétaire - Microsoft - 2005)

Distribués

- Mercurial (Libre - Matt Mackall - 2005)
- GIT (Libre – Linus Torvalds – 2005)
- Bazaar (Libre – Canonical - 2005)

Les différentes solutions de gestion de version

CVS – Concurrent Versions System – 1990



Principalement utilisé dans des projets de logiciels **libres**

Modèle centralisé

Plusieurs **implémentations**

- Cervisia, WinCVS, linCVS, CVSNT, TortoiseCVS

Les différentes solutions de gestion de version

SVN – Apache Subversion – 2000



Modèle centralisé

Conçu pour **succéder** à CVS

Modèle similaire à CVS, implémentations **ré-écrites**

Pas de distinction entre : label, branche et répertoire

- Permet des comparaisons faciles
- Laisse le choix à l'utilisateur

Les différentes solutions de gestion de version

Mercurial - Matt Mackall – 2005

Ecrit en python

Modèle complètement distribué



Rapidité, robustesse, gestion avancée des fusions

Interface Web intégrée

Les différentes solutions de gestion de version

Bazaar - Canonical Ltd – 2005

Ecrit en python

Modèle décentralisé



Jeu de commandes restreint

- Facilité d'apprentissage
- Facilité d'utilisation

Sait lire les historiques **d'autres gestionnaires de version** (svn, Git, etc ...)

<https://code.launchpad.net/projects>

Présentation de GIT

- Concepts de base du contrôle de version
- La gestion centralisée ou distribuée
- Les différentes solutions de gestion de versions
- Principes et apports la décentralisation

Principes et apports de la décentralisation

Dépôt de fichiers **non unique** sur un serveur

Chaque développeur possède **son propre dépôt**

Chaque dépôt contient **l'ensemble** des versions des fichiers et des branches

En découle de nombreuses fonctionnalités ...

Principes et apports de la décentralisation

Travailler **hors connexion**

Chaque développeur peut travailler dans son propre dépôt, sans publier tout de suite

Vous laisse le choix de **quand et où** travailler !



MERCI GIT !

Principes et apports de la décentralisation

Plus facile pour le développeur de **versionner ses modifications**

Moins de **crainte d'impacts** pour les autres développeurs

Partage du travail (publication) seulement lorsqu'il a terminé de développer la fonctionnalité

Possibilité de **séparer le travail** (correction bug par bug dans différents commits)

Principes et apports de la décentralisation

Permet au développeur de créer **ses propres branches** privées

Pas d'obligation de les partager

Fusion des branches une fois le travail terminé, puis publication

Principes et apports de la décentralisation

Néanmoins ...

Avoir seulement des dépôts locaux est acceptable dans une certaine mesure.

- Très peu de développeurs, voire un seul !
- Petit projet

Comment savoir qui a la dernière version ???

Un serveur (ou plus) ayant le dépôt est indispensable pour tout projet de taille conséquente.

Principes et apports de la décentralisation

Un seul serveur de dépôt ?

Non, il est possible de publier des **copies d'un dépôt**, et ainsi, de **partager** des patches entre eux.

Cela est bien plus facile à faire qu'avec un système centralisé... mais au prix d'une certaine **complexité**.

Ce réseau de dépôts et son fonctionnement devront être bien définis et **documentés** pour le bien être du projet

Principes et apports de la décentralisation

Pourquoi tout le monde n'utilise pas un système distribué ?

- Outils « jeunes » (même si stables)
- Ecosystème des outils encore en **maturation**
- Subversion s'intègre avec une multitude d'outils (IDE, bug tracker ...)
- Complexité plus importante (organisation de **processus** et **conventions**)

Installation et configuration

Un peu de pratique

Installation et configuration

■ Installation sous différents systèmes

- Sous Windows : msysgit

- Le fichier .gitconfig

- Analyse de différentiel de versions

Installation sous différents systèmes

Git est disponible sur

- Mac OS X
- Windows
- Linux
- Solaris

<https://git-scm.com/download>

Il est aussi possible de compiler les sources

<https://github.com/git/git>

Installation sous différents systèmes

Sous Linux (pour debian-like)

Directement dans les dépôts

- Ajouter tout de même le dépôt à votre gestionnaire pour bénéficier de la dernière version

```
$ sudo apt-get install git
```

Installation sous différents systèmes

Sous Mac OS X

Un installeur existe :

<https://sourceforge.net/projects/git-osx-installer/>

Ou bien, view homebrew

```
$ brew update  
$ brew install git
```

Installation sous différents systèmes

Sous Windows

Via l'installateur

<https://git-scm.com/download/win>

Installation et configuration

- Installation sous différents systèmes
- Sous Windows : msysgit
- Le fichier .gitconfig
- Analyse de différentiel de versions

Sous Windows : msysgit

=> **msysgit** : environnement **msys** + **mingw** + tout le nécessaire pour compiler Git sous Windows.

=> **Git for windows** : Git compilé pour Windows

Adaptation des **outils** de développement **GNU** pour Windows

Un shell : **Git BASH**

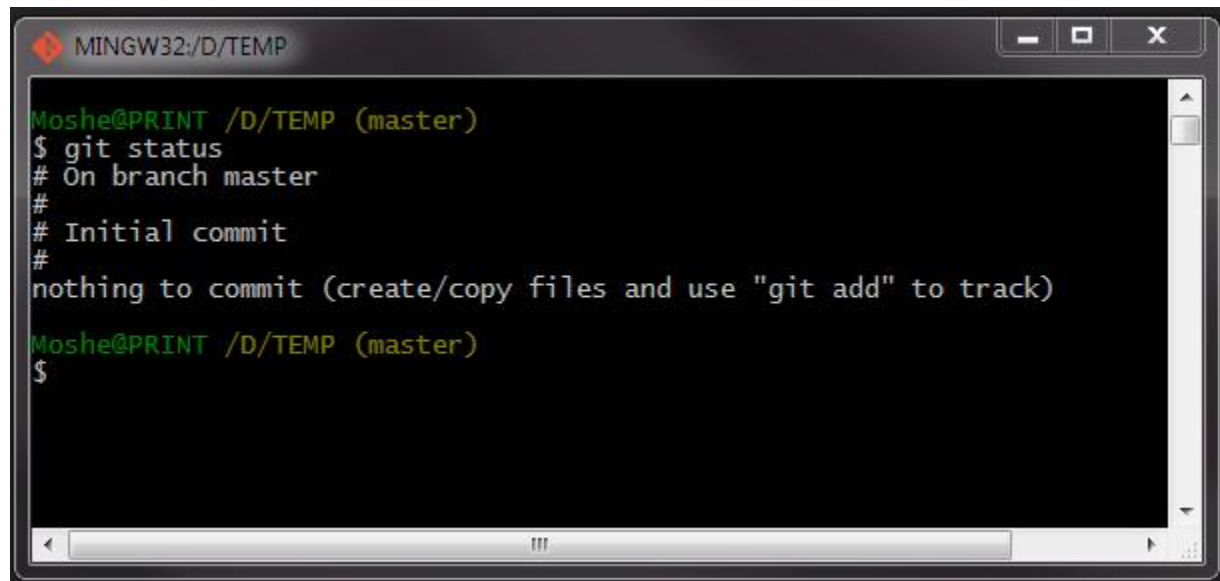
Une interface graphique : **Git GUI**

Et bien sûr ... **Git**

Sous Windows : msysgit

Git BASH : Console Shell avec les commandes de base de Linux: ls, mv, curl, etc.

Utilisée en remplacement du terminal par défaut de Windows



```
MINGW32:/D/TEMP
Moshe@PRINT /D/TEMP (master)
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
Moshe@PRINT /D/TEMP (master)
$
```

Installation et configuration

- Installation sous différents systèmes
- Sous Windows : msysgit
- Le fichier .gitconfig
- Analyse de différentiel de versions

Le fichier .gitconfig

Pour personnaliser son environnement Git

Réglages **persistants**

Possibilité de l'éditer en **ligne de commande**

Plusieurs niveaux de configuration possible

- Niveau système
- Niveau utilisateur
- Niveau dépôt

Le fichier .gitconfig

Pour changer un paramètre :

```
$ git config [<option>] [<valeur>]  
$ git config [<option>] user.name "Robert Plant"
```

Niveau système (option --system)

- /etc/gitconfig

Niveau utilisateur (option --global)

- ~/.gitconfig

Niveau dépôt (option --local, option par défaut)

- [répertoire du dépôt]/.git/config

Le fichier .gitconfig

Premier paramètre à renseigner : **son identité** (nom et adresse e-mail)

```
$ git config --global user.name "Robert Plant"  
$ git config --global user.email "rob.plant@led.zep"
```

Deuxième paramètre : **son éditeur de texte** (utilisé quand Git demande de saisir un message)

```
$ git config --global core.editor vim
```

Le fichier .gitconfig

Vérifier ses paramètres

```
$ git config --list  
user.name=Robert Plant  
user.email=rob.plant@led.zep  
color.status=auto  
color.status=auto  
...
```

Certains paramètres peuvent apparaître plusieurs fois selon le **niveau de configuration**

Pour n'afficher qu'un seul paramètre :

```
$ git config user.name  
Robert Plant
```

Installation et configuration

- Installation sous différents systèmes
- Sous Windows : msysgit
- Le fichier .gitconfig
- Analyse de différentiel de versions

Analyse de différentiel de versions

Il existe plusieurs outils d'analyse compatibles avec Git :

kdifff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, opendiff

Un outil d'analyse de différentiel permet de

- Visualiser les différences entre différentes versions d'un fichier
- Reporter les modifications d'une version à une autre

Comme pour le choix de l'éditeur de texte, on peut définir son outil d'analyse

```
$ git config --global diff.tool <diff tool to use>
```

Analyse de différentiel de versions

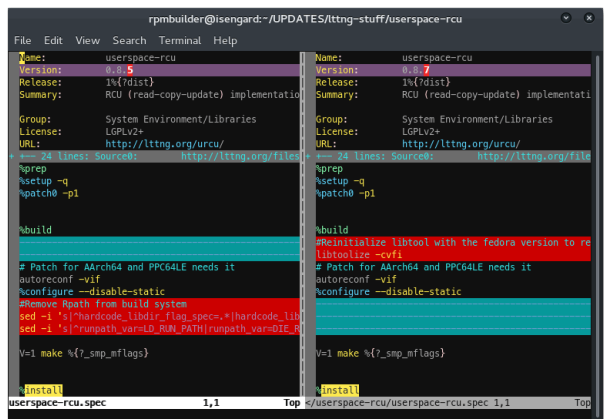
Vimdiff (Embarqué avec vim)

Permet de comparer jusqu'à 3 versions simultanément.

Les zones en surbrillance sont affectées par des modifications

```
$ git config --global diff.tool vimdiff
```

```
$ git difftool branche1:FICHER branche2:FICHER
```



```
rpmbuilder@isengard:~/UPDATES/lttng-stuff/userspace-rcu
File Edit View Search Terminal Help
Name: userspace-rcu
Version: 1.0.0
Release: 1%{dist}
Summary: RCU (read-copy-update) implementation
Group: System Environment/Libraries
License: LGPLv2+
URL: http://lttng.org/urcu/
Source: http://lttng.org/files
%prep
%setup -q
%patch0 -p1
%build
# Patch for AArch64 and PPC64LE needs it
autoreconf -vif
%configure --disable-static
# Patch for AArch64 and PPC64LE needs it
autoreconf -vif
%install
V=1 make %{?_smp_mflags}
%install
V=1 make %{?_smp_mflags}
```

Analyse de différentiel de versions

Affichage d'un diff entre le HEAD et la version locale

```
$ git difftool FICHIER_A_ANALYSER
```

Par défaut, Git va comparer la version locale du fichier, avec la **dernière version de la branche courante**.

Pour comparer des versions de branches/tags différentes :

```
$ git difftool branche1:FICHIER v2.2.10:FICHIER
```

Documentation de difftool

<https://git-scm.com/docs/git-difftool>

Analyse de différentiel de versions

Meld

Difftool avec une interface graphique.

Configuration de meld comme difftool et lancement

```
$ git config --global diff.tool meld
```

```
$ git difftool FICHIER
```



Utilisation de GIT, les fondamentaux

git help

Utilisation de Git, les fondamentaux

- Le modèle objet Git
- Le répertoire Git
- L'index ou staging area
- Création d'un dépôt
- Branches, tags, dépôts
- Outil graphique : gitk

Le modèle objet Git

Git est comme un système de fichiers au dessus du système de fichiers du poste utilisateur.

Le dossier « **.git** » situé à la racine du projet contient tous les fichiers nécessaires pour que Git fonctionne.

Toutes les données de l'historique sont stockées dans ces fichiers.

On parle **d'objets**.

Le modèle objet Git

Ces fichiers ont comme nom un hash **SHA1** de 40 caractères :

Ce hash est calculé avec le contenu de l'objet

- Contenu d'un fichier
- Liste de dossiers / fichiers
- Méta données (auteurs, dates, messages...)

Pour comparer 2 objets, il suffit pour Git de comparer deux signatures

```
92abfb8e70d090c9f90280ac4f9b546da527bcf1
```

Le modèle objet Git

Chaque objet est composé

- D'un **type**
- D'une **taille** (taille du contenu)
- Du **contenu**

4 types d'objets :

- **Blob** (données d'un fichier)
- **Tree** (Répertoire, référence d'autres **tree** et **blob**)
- **Commit** (Pointe vers un **tree**, snapshot de l'état du projet dans le temps)
- **Tag** (Pointe vers commit spécifique, permet de tagger une version)

Le modèle objet Git

Pas de système de « deltas » comme SVN, Mercurial ou d'autres.

Git stocke une **vue instantanée du projet** dans ses fichiers à chaque commit

```
TAG =>  
  COMMIT =>  
    TREE =>  
      TREE =>  
        BLOB  
        BLOB  
      TREE =>  
        BLOB  
      BLOB  
      BLOB
```

Le modèle objet Git

Le type **blob**

- Stocke le **contenu** d'un fichier
- Ne référence pas d'autres objets
- Renommer un fichier ne change pas son **blob**

Pour afficher le contenu d'un blob :

```
$ git show 92abfb8e70d090c9f90280ac4f9b546da527bcf1
```

```
Ceci est le contenu de mon fichier ...  
Lorem ipsum dolor sit amet,  
consectetur adipiscing elit ...
```

Le modèle objet Git

Le type **tree**

- Contient une liste de pointeurs vers des **blobs** et d'autres **trees**

Pour afficher la liste des sous trees et blobs

```
$ git ls-tree 92abfb8e70d090c9f90280ac4f9b546da527bcf1
```

```
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c fichier1
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d fichier2
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3 fichier3
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745 dossier1
```


Le modèle objet Git

Le type **commit**

- Pointe vers un **tree**
- Référence le(s) **commit(s) parent(s)**
- Contient des **métadonnées** (auteurs, dates, message)

Pour examiner un commit

```
$ git show 92abfb8e70d090c9f90280ac4f9b546da527bcf1
```

```
commit 92abfb8e70d090c9f90280ac4f9b546da527bcf1
```

```
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
```

```
parent 257a84d9d02e90447b149af58b271c19405edb6a
```

```
author Robert Plant <rob.plant@led.zep> 1187576872 -0400
```

```
committer Someone Else <someone@domain.com> 1187591163 -0700
```

```
My commit message ...
```

Utilisation de Git, les fondamentaux

- Le modèle objet Git
- Le répertoire Git
- L'index ou staging area
- Création d'un dépôt
- Branches, tags, dépôts
- Outil graphique : gitk

Le répertoire Git

Tous les objets sont stockés dans le répertoire **.git**, présent à la **racine du projet**

```
$ ls -al .git
```

```
drwxrwxr-x  2 user user 4096 mai   9 10:33 branches/
-rw-rw-r--  1 user user  15 mai  11 12:13 COMMIT_EDITMSG
-rw-rw-r--  1 user user 261 mai   9 10:33 config
-rw-rw-r--  1 user user  73 mai   9 10:33 description
-rw-rw-r--  1 user user  90 mai  10 10:53 FETCH_HEAD
-rw-rw-r--  1 user user  23 mai   9 10:33 HEAD
drwxrwxr-x  2 user user 4096 mai   9 10:33 hooks/
-rw-rw-r--  1 user user 1519 mai  11 12:13 index
drwxrwxr-x  2 user user 4096 mai   9 10:33 info/
drwxrwxr-x  3 user user 4096 mai   9 10:34 logs/
drwxrwxr-x 35 user user 4096 mai  11 12:13 objects/
-rw-rw-r--  1 user user  41 mai  10 10:53 ORIG_HEAD
drwxrwxr-x  5 user user 4096 mai   9 10:34 refs/
```

Le répertoire Git

Liste des fichiers/dossiers :

- **description** : fichier utilisé uniquement par le programme GitWeb
- **config** : fichier contenant les options de configuration spécifiques à votre projet
- **info** : répertoire contenant un fichier d'exclusions que vous ne voulez pas mettre dans un fichier **.gitignore** du projet
- **hooks** : répertoire contenant les scripts de procédures automatiques côté client ou serveur
- **objects** : répertoire contenant les objets Git
- **refs** : fichier stockant les pointeurs vers les objets **commit**
- **HEAD** : fichier pointant sur la branche en cours
- **FETCH_HEAD** : fichier pointant sur le dernier commit téléchargé
- **index** : fichier contenant les informations sur la zone d'attente

Le répertoire Git

Partage du dossier « .git »

Ce dossier peut être partagé, sauvegardé... il suffit d'avoir ce dossier pour re-créeer un espace de travail.

```
# Liste toutes les branches  
$ git branch
```

```
# Se place sur une branche (espace de travail restauré)  
$ git checkout <une-branche>
```

Utilisation de Git, les fondamentaux

- Le modèle objet Git
- Le répertoire Git
- L'index ou staging area
- Création d'un dépôt
- Branches, tags, dépôts
- Outil graphique : gitk

L'index ou staging area

Appelée aussi **zone de transit**

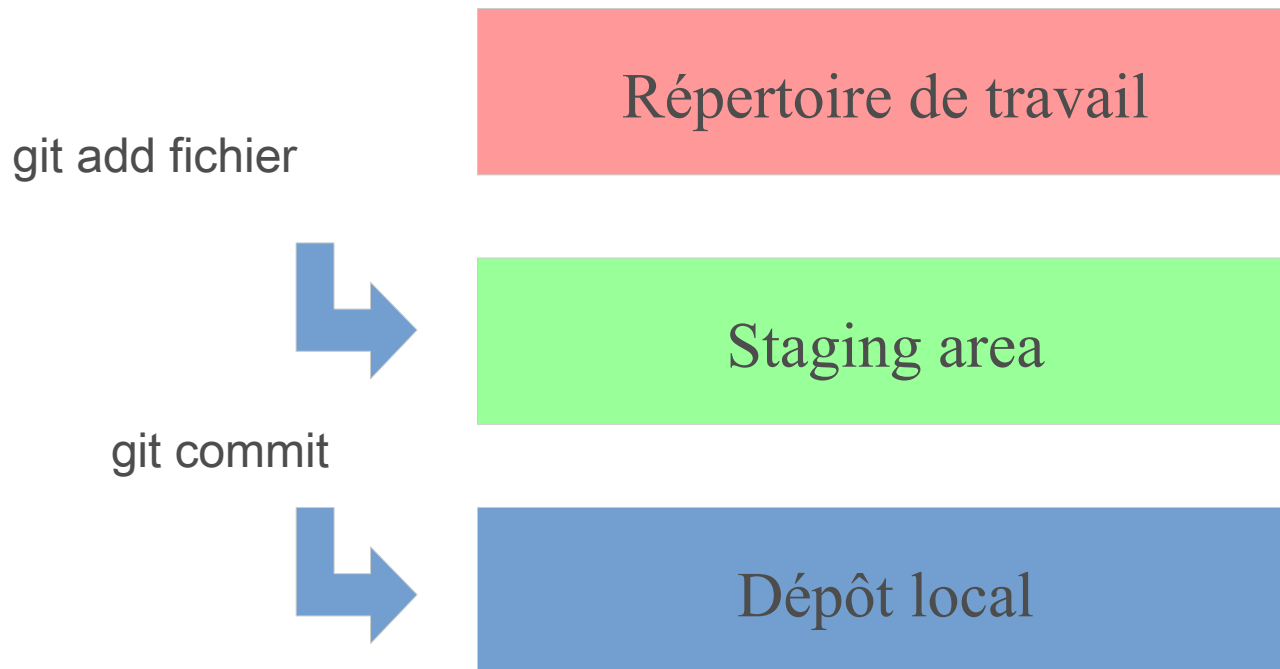
Zone **temporaire** avant écriture dans le dossier .git

Le développeur décide quels changements vont être effectivement commités

Permet une plus grande flexibilité et un meilleur contrôle

L'index ou staging area

Avant de pouvoir commiter certains fichiers, il faut les **ajouter** à cette zone temporaire.



L'index ou staging area

Lors de nombreuses modifications, il peut être utile de **séparer** les **commits**.

Il suffit alors d'ajouter les **fichiers concernés** à l'index, puis effectuer le commit :

```
$ git add fichier1  
$ git commit -m "Fonctionnalité A – ticket #345"  
$ git add fichier2  
$ git commit -m "Fonctionnalité B - ticket #58"
```

L'index ou staging area

Possibilité d'ajouter plusieurs fichiers/dossiers

```
$ git add chemin/vers/un/fichier  
$ git add chemin/vers/un/dossier  
$ git add .  
$ git add --all
```

L'index ou staging area

Si un fichier a été ajouté à l'index par erreur, il est possible de **l'enlever** de cette zone.

Deux solutions

```
$ git reset -- fichier
```

=> Enlève tout changement ajouté à l'index pour le fichier donné. Le contenu du fichier n'est pas modifié

```
$ git rm --cached fichier
```

=> Ajoute la suppression de ce fichier à l'index . Le fichier devient non tracké.

Utilisation de Git, les fondamentaux

- Le modèle objet Git
- Le répertoire Git
- L'index ou staging area
- Création d'un dépôt
- Branches, tags, dépôts
- Outil graphique : gitk

Création d'un dépôt

La commande « git init » permet d'initialiser un dépôt dans le **répertoire courant**

Le répertoire peut être vide ou déjà contenir des fichiers

Cela crée automatiquement le répertoire **.git** à la racine de ce répertoire

Si le répertoire contient déjà des fichiers, il peut être utile de les ajouter à l'index, puis créer le **premier commit** (commit racine)

```
$ git add --all  
$ git commit -m "First commit"
```

Création d'un dépôt

Il est possible d'initialiser un dépôt en copiant un autre **dépôt existant**

Avec subversion, on utilise la commande checkout, avec Git, c'est la commande **clone**

```
$ git clone [url]
```

Exemple : cloner le dépôt de subversion !

```
$ git clone https://github.com/apache/subversion.git
```

Cela crée un répertoire « subversion » dans le répertoire courant

Création d'un dépôt

Options de git clone

Cloner en spécifiant un dossier local

```
$ git clone [url] chemin/vers/dossier
```

Cloner une branche seulement

```
$ git clone --single-branch -b dev [url]
```

Sans historique

```
$ git clone --single-branch -b dev --depth 1 [url]
```

Création d'un dépôt

Création d'un dépôt sur un **serveur**

On ne veut pas stocker les fichiers, **seulement les objets**

On utilise l'option **--bare**

```
$ mkdir /home/git/mon-depot  
$ cd /home/git/mon-depot  
$ git init --bare  
Initialized empty Git repository in /home/git/mon-depot/
```


Utilisation de Git, les fondamentaux

- Le modèle objet Git
- Le répertoire Git
- L'index ou staging area
- Création d'un dépôt
- Branches, tags, dépôts
- Outil graphique : gitk

Branches, tags et dépôts

Les branches

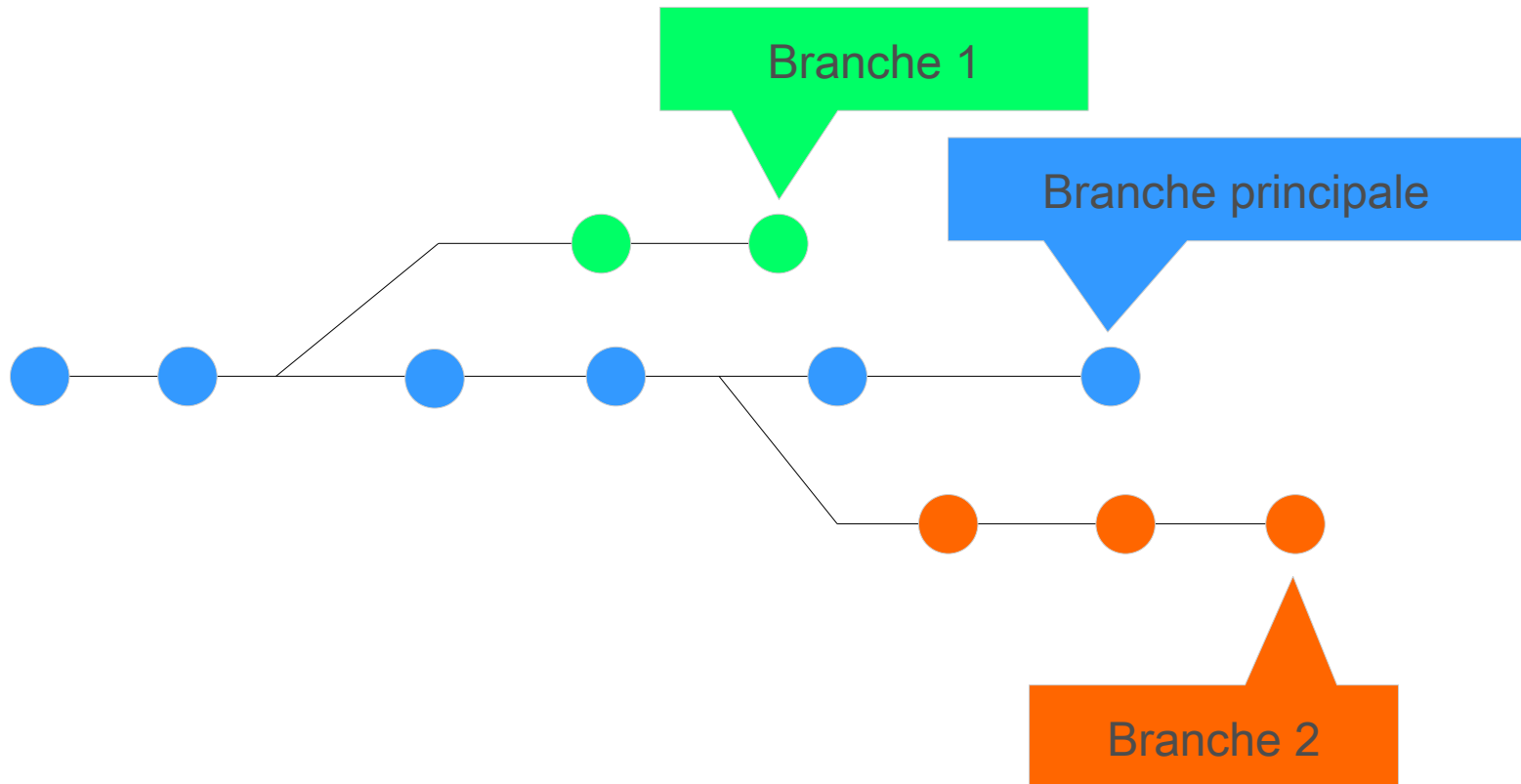
Presque tout VCS propose une **gestion de branches**, avec plus ou moins de rapidité lors de la création.

Il est indispensable dans tout projet de taille conséquente de maintenir **plusieurs branches**.

Une branche est simplement une **divergence** de la ligne principale, afin d'implémenter une fonctionnalité

Branches, tags et dépôts

Dans Git, une branche n'est qu'un **pointeur** vers un **objet commit**



Branches, tags et dépôts

Lister toutes les branches

```
$ git branch -a
```

Créer une **nouvelle branche** (depuis la branche courante)

```
$ git checkout -b ma-nouvelle-branche
```

Supprimer une branche

```
$ git branch -D la-branche-a-supprimer
```

Branches, tags et dépôts

Les tags

Un tag, comme une branche, est un **pointeur** vers un **objet commit**

Quelle différence ???

Un tag ne **bouge pas** alors que la branche évolue

Branches, tags et dépôts

2 **types** de tags

« **Légers** » et « **annotés** »

Le type léger, n'apporte **rien de spécial** (plutôt utilisé en local pour aider le développeur)

Le type annoté permet d'ajouter des **informations** (plutôt utilisé pour publier)

- Nom et email de l'auteur, date
- Peuvent être signées avec GNU Privacy Guard

Branches, tags et dépôts

Lister les tags

```
$ git tag
```

```
V0.0.1
```

```
V0.0.2
```

```
V0.0.3
```

```
...
```

Lister les tags avec filtre

```
$ git tag -l '1.2.*'
```

```
V1.2.0
```

```
V1.2.1
```

```
...
```

Branches, tags et dépôts

Créer un tag léger

```
$ git tag v5.0.1
```

Créer un tag annoté

```
$ git tag -a v5.0.1 -m "5.0.1 release"
```

Si le message n'est pas spécifié avec l'option « -a », Git lance votre éditeur de texte pour le saisir

Branches, tags et dépôts

Les dépôts

Par dessus les branches et les tags

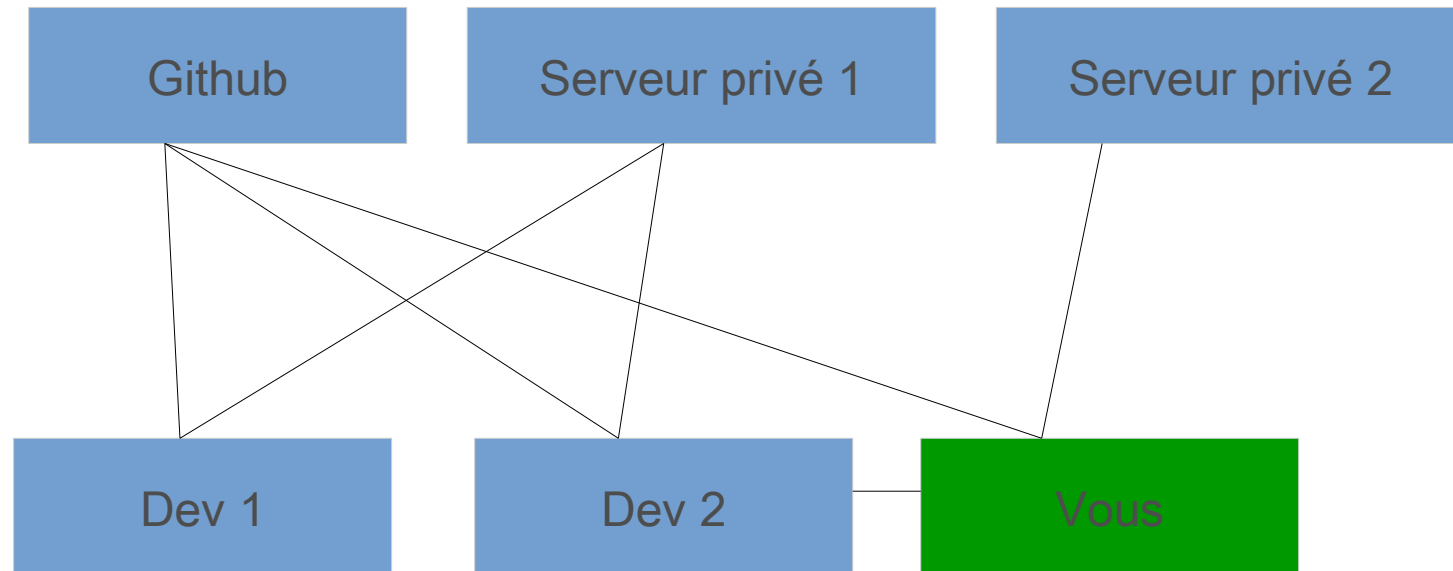
Un même projet peut avoir **plusieurs dépôts distants** (« remotes »)

Public ? Privé ?

On peut très bien mélanger les 2

Branches, tags et dépôts

Dépôts **publics**, dépôts **privés**



Branches, tags et dépôts

Lister les « remotes » de son dépôt

```
$ git remote  
Origin  
Github  
Serveur1
```

Ajouter un « remote » à son dépôt

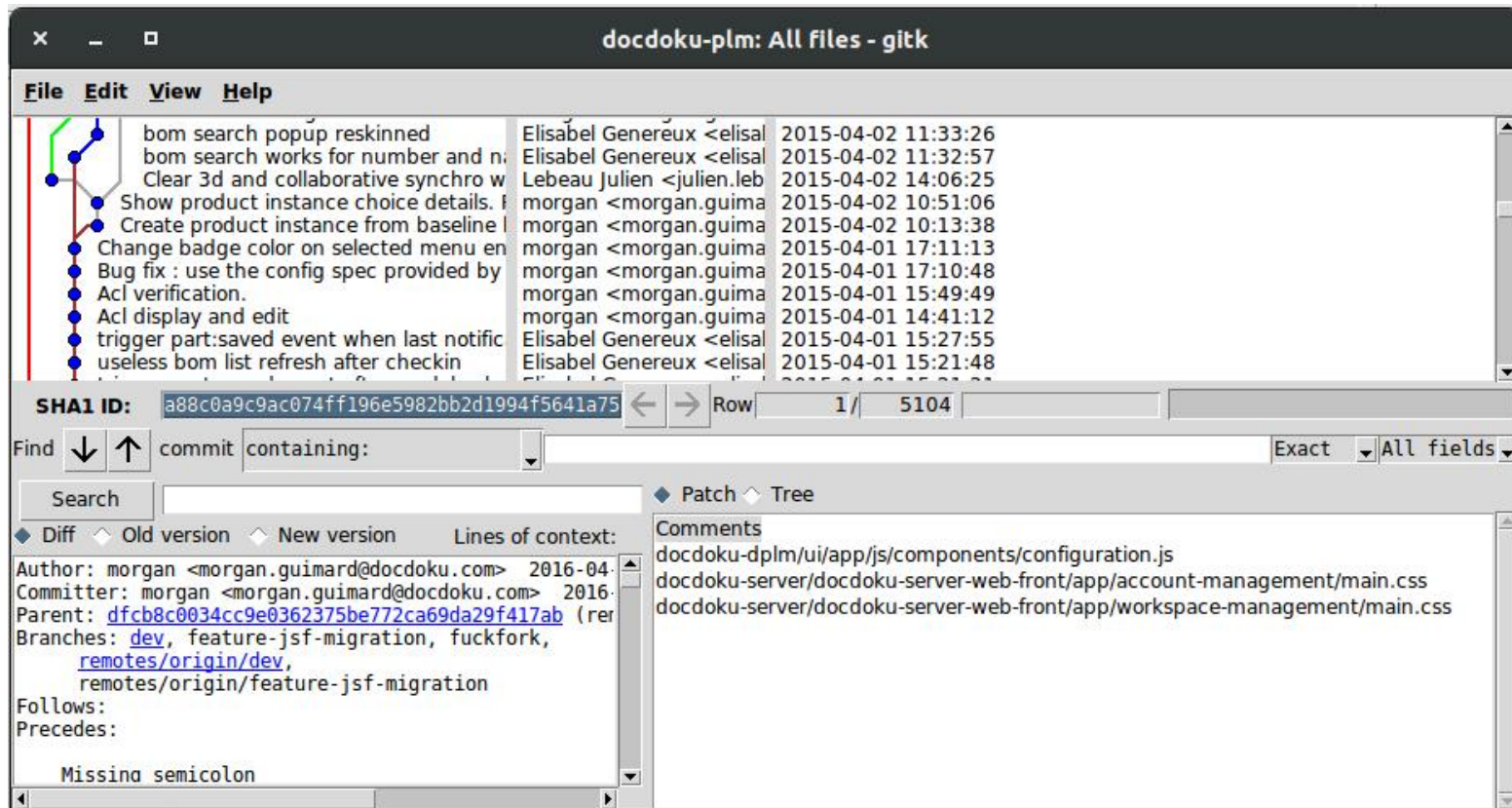
```
$ git remote add nom-du-remote [url]
```

Utilisation de Git, les fondamentaux

- Le modèle objet Git
- Le répertoire Git
- L'index ou staging area
- Création d'un dépôt
- Branches, tags, dépôts
- Outil graphique : gitk

Outil graphique : gitk

Permet de visualiser les différentes branches, leur divergences, les commits, les modifications associées



Outil graphique : gitk

Possibilité de filtrer la vue en ligne de commande

```
$ gitk [<options>] [<revision range>] [--] [<path>...]
```

--branches[=<pattern>]

--tags[=<pattern>]

--remotes[=<pattern>]

--since=<date>

--until=<date>

--all

Gestion locale des fichiers

git status

Gestion locale des fichiers

- Etat de l'arbre de travail
- Ajouter, ignorer, modifier, supprimer, rechercher
- Annulation et visualisation des modifications
- Parcours de l'historique, les logs

Etat de l'arbre de travail

Afficher l'état de l'arbre de travail

```
$ git status
```

Git affiche les chemins des fichiers :

- Qui diffèrent entre l'index et le HEAD
- Qui diffèrent entre le répertoire de travail et l'index
- Qui ne sont pas « trackés »

Etat de l'arbre de travail

Principales options de git status

- s (**short**) : Affiche le résultat dans un formatage simple
- b (**branch**) : Affiche les détails de la branche dans un formatage simple
- porcelain : Affiche le résultat dans un format facilement parsable
- long (par défaut) : Affiche le résultat dans un formatage long

Etat de l'arbre de travail

Format de sortie

```
$ XY PATH1 → PATH2
```

X prend comme valeurs :

- ' ' = non modifié
- M = modifié
- A = ajouté
- D = supprimé
- R = renommé
- C = copié
- U = Mis à jour, non fusionné

Y : Lors de conflits de fusion (voir chapitre « Fusionner des branches, gérer les conflits »)

PATH1 : Chemin du fichier (PATH2 s'affiche si le chemin a changé)

Gestion locale des fichiers

- Etat de l'arbre de travail
- Ajouter, ignorer, modifier, supprimer, rechercher
- Annulation et visualisation des modifications
- Parcours de l'historique, les logs

Ajouter, ignorer, modifier, supprimer, rechercher

Les fichiers **non trackés**, ou dans la **staging area** doivent être ajoutés pour être commités

Tout ajouter

```
$ git add --all
```

Ajouter un fichier (ou répertoire)

```
$ git add chemin/vers/fichier
```

Ajouter, ignorer, modifier, supprimer, rechercher

Les fichiers susceptibles d'être modifiés mais qu'on **ne veut pas commiter** (configuration par exemple), peuvent être **ignorés de l'index**

```
$ git update-index --assume-unchanged chemin/vers/fichier
```

Pour ne plus ignorer le fichier

```
$ git update-index --no-assume-unchanged chemin/vers/fichier
```

Ajouter, ignorer, modifier, supprimer, rechercher

Plus généralement, la commande **update-index** permet plusieurs choses

- add : ajoute à l'index (comme git add)
- remove : enlève de l'index (comme git rm)
- refresh : vérifie si des fusions ou mises à jour sont nécessaires

Voir toutes les options :

<https://git-scm.com/docs/git-update-index>

Ajouter, ignorer, modifier, supprimer, rechercher

Rechercher et filtrer dans les **noms** des fichiers

```
$ git ls-files '*Test.java'
```

Sans le pattern, Git liste tous les fichiers de l'index.

Possibilité de filtrer sur **l'état** des fichiers

```
$ git ls-files --[cached|deleted|others|stage|unmerged|killed|modified]
```


Ajouter, ignorer, modifier, supprimer, rechercher

Rechercher et filtrer dans le **contenu** des fichiers

```
$ git grep "contenu recherché"
```

Configuration

- --line-number affiche le numéro de ligne
- --threads=[nombre de threads à utiliser]
- --full-name chemins relatifs à la **racine du projet** ou du **répertoire courant**

Choisir où chercher :

- --cached : dans l'index seulement
- --no-index : dans les fichiers non trackés seulement
- --untracked : dans les fichiers non trackés (en plus des fichiers trackés)

Gestion locale des fichiers

- Etat de l'arbre de travail
- Ajouter, ignorer, modifier, supprimer, rechercher
- Annulation et visualisation des modifications
- Parcours de l'historique, les logs

Annulation et visualisation des modifications

Pour savoir quel changements ont été effectués dans tout le projet : **git diff**

```
$ git diff
diff --git a/fichier.txt b/fichier.txt
index e69de29..dc013ec 100644
--- a/fichier.txt
+++ b/fichier.txt
@@ -0,0 +1 @@
+# HELLO WORLD #
```

Ou en spécifiant un chemin (fichiers, dossiers, branches)

```
$ git diff [fichier|dossier|branche]
```

Annulation et visualisation des modifications

Pour plus de flexibilité, comparer entre deux commits

```
$ git diff 45cd543 5465784 ./fichier.txt
```

Chercher seulement dans l'index

```
$ git diff --cached
```

Avoir une vue d'ensemble

```
$ git diff --stat
```

<https://git-scm.com/docs/git-diff>

Annulation et visualisation des modifications

Annuler un ajout

```
$ git reset
```

Annuler un commit

```
$ git reset HEAD~1
```

Remettre le commit supprimé

```
$ git reset 'HEAD@{1}'
```

<https://git-scm.com/docs/git-reset>

Gestion locale des fichiers

- Etat de l'arbre de travail
- Ajouter, ignorer, modifier, supprimer, rechercher
- Annulation et visualisation des modifications
- Parcours de l'historique, les logs

Parcours de l'historique, les logs

Afficher tout l'historique et le parcourir

```
$ git log
```

Pour afficher de façon résumée

```
$ git log --pretty=oneline --abbrev-commit
```

Entre deux commits

```
$ git log a5e3af4...b123ff1
```

<https://git-scm.com/docs/git-log>

Parcours de l'historique, les logs

Afficher avec un peu de style (penser à faire des alias!)

```
$ git log --graph --pretty=format:  
'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)  
%C(bold blue)<%an>%Creset' --abbrev-commit
```

```
* 7015c90 - (HEAD, origin/master, origin/HEAD, master) Updated iPhones graphic with new screenshot (11 days ago) <Oinutter>  
* 9bfa321 - Updated README (2 weeks ago) <Oinutter>  
* dd18305 - Updated to new design (2 weeks ago) <Oinutter>  
* b2e36c6 - Added fallback for missing avatar (2 weeks ago) <Oinutter>  
* 8678978 - Removed Disk caching of avatar (6 weeks ago) <Oinutter>  
* 39d2c3c - Preparing for App Store submission (9 weeks ago) <Oinutter>  
* 4fd7261 - Fixed cell bg when user only has 1 badge. (9 weeks ago) <Oinutter>  
* 13bf363 - Fix image (10 weeks ago) <Oinutter>  
* 705363e - Try screenshots with transparent background (10 weeks ago) <Oinutter>  
* bd4602e - Correct image location in README (10 weeks ago) <Will McKenzie>  
* 483d05b - Added iPhone screenshots to README (10 weeks ago) <Oinutter>  
* 2d7a6d7 - Adding license (10 weeks ago) <Oinutter>  
* d6c6660 - Adding to .gitignore (10 weeks ago) <Oinutter>
```


Parcours de l'historique, les logs

Les alias pratiques pour l'historique

Affichage condensé :

```
git config --global alias.hist "log --pretty=format:'%C(yellow)%ad%C(reset) %C(green)%h%C(reset) %s (%an) %C(yellow)%d%C(reset)' --date=short"
```

Afficher un graph :

```
git config --global alias.graph "log --pretty=format:'%C(yellow)[%ad]%C(reset) %C(green)[%h]%C(reset) | %C(red)%s %C(bold red){%an}}%C(reset) %C(blue)%d%C(reset)' --graph --date=short"
```

Utilisation

git hist / git hist -10 / git hist une-branche / git hist un-tag

Gestion des branches

git branch

Gestion des branches

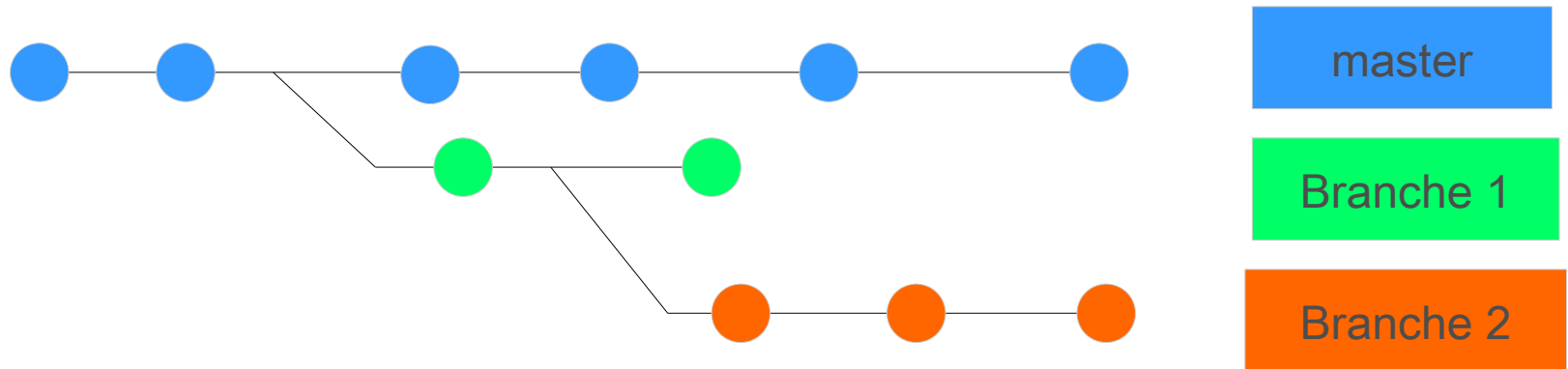
- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

La branche « master »

Pourquoi « **master** » ?

Juste une convention, mais on peut s'en passer

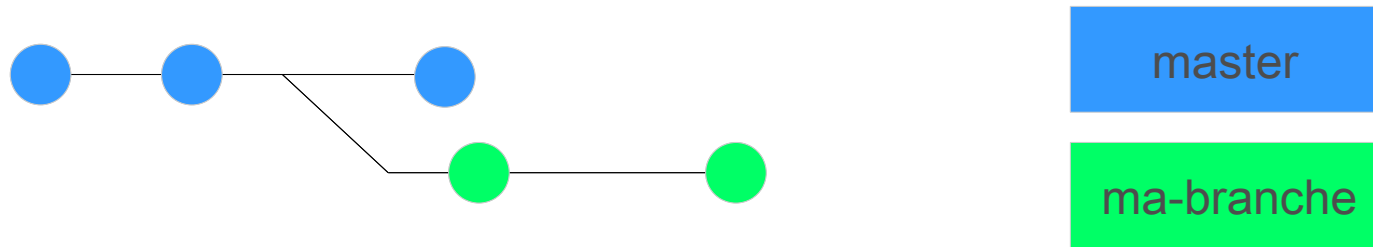
En général, cette branche est considérée comme la **dernière version** stable du projet



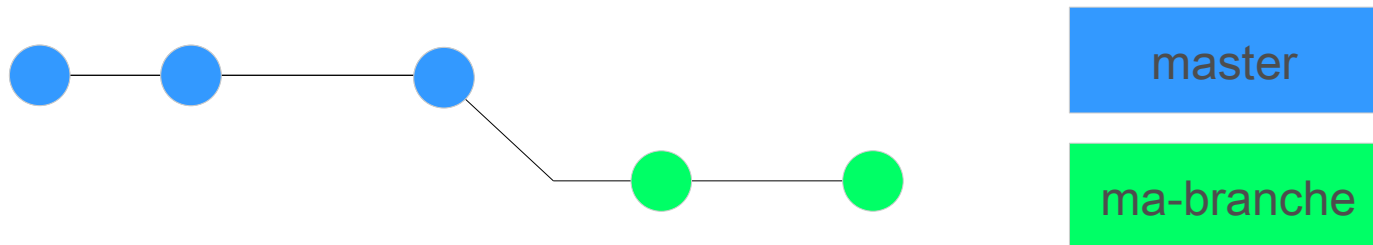
Récupération du travail de l'équipe

Pour se maintenir à jour d'une branche parent (souvent utile si on a du retard)

Supposons que «ma-branche» dérive de «master », et qu'elle est en retard



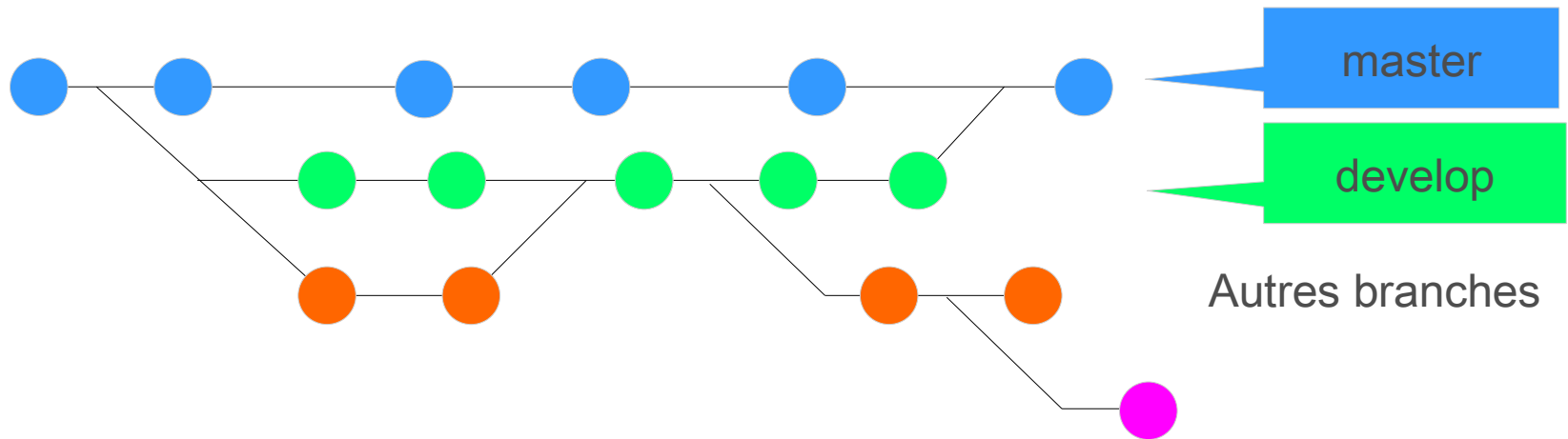
```
$ git checkout ma-branche  
$ git rebase master
```



La branche « master »

Plusieurs branches dérivent de cette branche racine, et peuvent fusionner, ou vivre leur vie.

On obtient généralement des schémas ressemblants



Gestion des branches

- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

Création de branches, sous-branches, naviguer

Pour créer une branche depuis la branche courante

```
$ git branch ma-nouvelle-branche
```

Pour se placer sur la branche une fois créée

```
$ git checkout ma-nouvelle-branche
```

Pour se placer sur une autre branche

```
$ git checkout autre-branche
```

Git change alors le contenu du répertoire de travail avec **l'état du projet** correspondant au **commit pointé par la branche**

Création de branches, sous-branches, naviguer

Pourquoi créer des branches ?

- Le coût est minime (pointeurs)
- Création quasi **instantané**
- Aide à **séparer** les features
- Basculement entre branches quasi **instantané**

Git encourage à travailler avec cette méthode

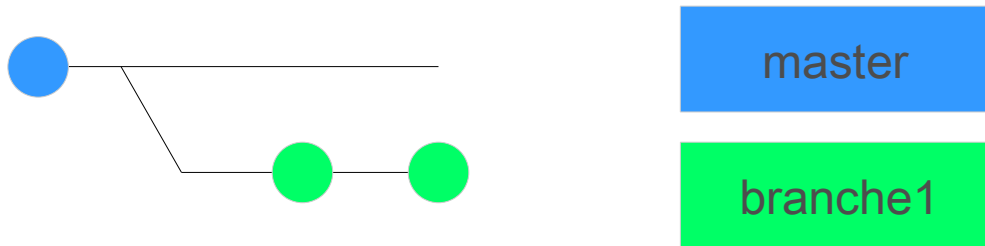
La façon de développer peut changer !

Gestion des branches

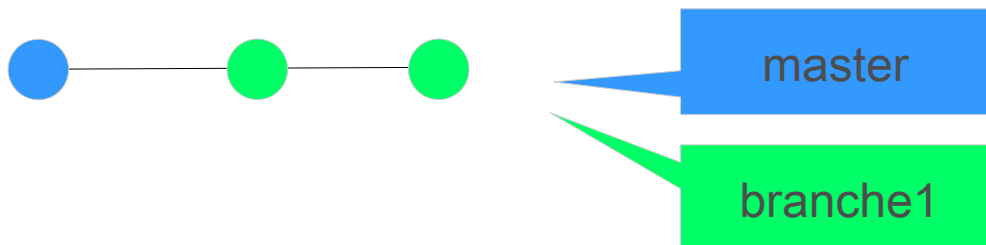
- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

Fusionner des branches, gérer les conflits

Pour « merger » deux branches, il faut se placer sur la branche sur laquelle on souhaite ajouter les commits de l'autre



```
$ git checkout master  
$ git merge branche1
```



Fusionner des branches, gérer les conflits

Si les 2 branches ont une modification sur un même fichier, et que Git **n'arrive pas à les résoudre**, des conflits apparaissent, les fichiers concernés sont listés par l'opération de merge

```
$ git merge branche1
Auto-merging fichier.js
CONFLICT (content): Merge conflict in fichier.js
```

```
$ cat fichier.js
<<<<<< HEAD:fichier.js
$(function(){alert('hello')})
=====
$(function welcome(){
    alert('ola');
});
>>>>>> branche1:fichier.js
```

Fusionner des branches, gérer les conflits

Résoudre ce conflit manuellement en supprimant les lignes de conflits et garder une des deux versions (ou en éditer une)

```
$ cat fichier.js
$(function welcome(){
  alert('miaou');
});
```

Une fois tous les conflits résolus dans un fichier, il faut lancer git add pour le marquer comme résolu

```
$ git add fichier.js
```

Il faudra ensuite commiter ces modifications

```
$ git commit
```

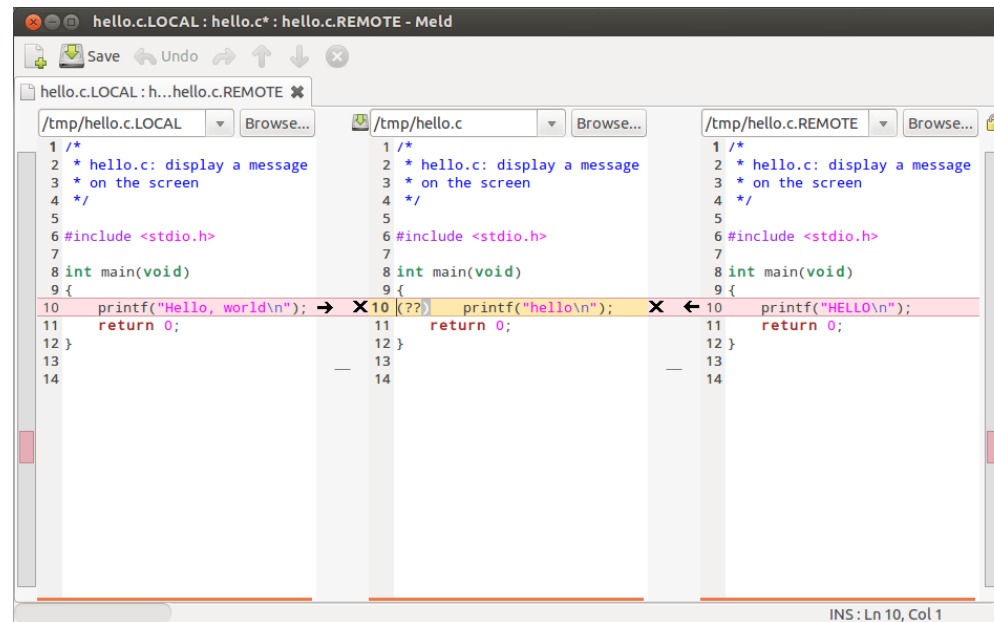
Fusionner des branches, gérer les conflits

Fusionner et résoudre les conflits avec un **outil graphique**

- Configurer un outil de merge (par exemple meld)
- Lancer git mergetool

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

```
$ git mergetool
```



Fusionner des branches, gérer les conflits

Une vue en 3 colonnes est souvent présentée avec des raccourcis pour choisir quelle version prendre pour la ou les lignes en conflits

Branche courante	Résultat	L'autre branche
Du texte	Contenu original	Autre chose
...

Gestion des branches

- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

Comparaison de deux branches

Comparer deux branches avec **git diff**

Afficher les changements effectués sur branche2 depuis leur divergence

```
$ git diff branche1...branche2
```

Afficher les changements entre les HEAD de chaque branche

```
$ git diff branche1 branche2
```

Attention à l'ordre branche1 / branche2 !

Gestion des branches

- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

Le remisage

Le « couper/coller » dans Git

Cas d'usage : changer de branche sans commiter les modifications en cours, mais tout en les gardant de côté.

```
$ // travail en cours...  
$ git stash  
$ // Changement de branche possible  
$ // une fois revenu sur notre branche  
$ git stash apply
```

Possibilité d'ajouter à l'index après remisage

Le remisage

Afficher la pile de modifications remisées

```
$ git stash list  
stash@{0}: WIP on master: 049d078 added the index file  
stash@{1}: WIP on master: c264051... Revert "added file_size"  
stash@{2}: WIP on master: 21d80a5... added number to log
```

Réappliquer une remise sans indexation

```
$ git stash apply  
$ git stash apply stash@{1}
```

Réappliquer une remise avec indexation

```
$ git stash apply --index
```

Le remisage

Supprimer une remise de la pile

```
$ git stash drop <optionel : stash@{0}>
```

Gestion des branches

- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

Rebase interactif

Ré-écrire l'historique :

- Fusionner les commits « WIP » en un seul
- Diviser un commit
- Modifier le message d'un commit
- Supprimer un commit
- Ré-ordonner des commits

Rebase interactif

```
$ git hist
* [2018-10-19] [0809768] | Almost last commit    (HEAD -> feature1)
* [2018-10-19] [58d5d47] | Last commit
* [2018-10-19] [64d7764] | Add title
* [2018-10-19] [f4a4e77] | WIP: add hello message
* [2018-10-19] [e1f5854] | WIP: add hello message
* [2018-10-18] [067466c] | Add .gitignore    (master)
* [2018-10-18] [ae213ff] | First commit
$ git rebase -i 067466c    ⇔    git rebase -i HEAD~5
```


Rebase interactif

L'éditeur s'ouvre et permet d'indiquer les commandes à appliquer

```
pick e1f5854 WIP: add hello message
pick f4a4e77 WIP: add hello message
pick 64d7764 Add titile
pick 58d5d47 Last commit
pick 0809768 Almost last commit

# Rebase 067466c..0809768 onto 067466c (5 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Rebase interactif

En commentaires les commandes disponibles :

```
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Rebase interactif

Application des commandes

pick e1f5854 WIP: add hello message
pick f4a4e77 WIP: add hello message
pick 64d7764 Add titile
pick 58d5d47 Last commit
pick 0809768 Almost last commit



pick e1f5854 WIP: add hello message
squash f4a4e77 WIP: add hello message
reword 64d7764 Add titile
pick 0809768 Almost last commit
pick 58d5d47 Last commit

Puis sauvegarde éditeur et fermeture

Rebase interactif

Squash des 2 commits :

This is a combination of 2 commits.

This is the 1st commit message:

WIP: add hello message

This is the commit message #2:

WIP: add hello message



Add hello message

Add message in body to say hello.

Rebase interactif

Ré-écriture du message du commit

Add title

Please enter the commit message
for your changes.
Lines starting with '#' will be ignored,
and an empty message aborts the commit.



Add title

Add title in the header

Please enter the commit message
for your changes.
Lines starting with '#' will be ignored,
and an empty message aborts the commit.

Rebase interactif

Application des commandes :

```
$ git rebase -i 067466c
[detached HEAD abc59f1] Add hello message
Date: Fri Oct 19 10:12:34 2018 +0200
1 file changed, 1 insertion(+)
[detached HEAD 4101818] Add title
1 file changed, 1 insertion(+)
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: could not apply 1c89576... Almost last commit
```

Conflit lors du
ré-ordonnement
des 2 derniers commits

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".

Could not apply 1c89576... Almost last commit

Rebase interactif

Résolution des conflits :

- Édition du fichier
- git add
- git rebase --continue

```
$ git add index.html
$ git rebase --continue
Successfully rebased and updated refs/heads/feature1.
$ git hist
* [2018-10-19] [a0f4729] | Last commit      (HEAD -> feature1)
* [2018-10-19] [12b126f] | Almost last commit
* [2018-10-19] [4101818] | Add title
* [2018-10-19] [abc59f1] | Add hello message
* [2018-10-18] [067466c] | Add .gitignore   (master)
* [2018-10-18] [ae213ff] | First commit
```

Gestion des branches

- La branche « master »
- Création de branches, sous-branches, naviguer
- Fusionner des branches, gérer les conflits
- Comparaison de deux branches
- Le remisage
- Rebase interactif
- Cherry-pick

Cherry-pick

Copier/coller un commit d'une autre branche

À reporter sur master

```
* [2018-10-19] [0a5354e] | Bug: #1324: login broken    (HEAD -> feature1)
* [2018-10-19] [a0f4729] | Last commit
* [2018-10-19] [12b126f] | Almost last commit
* [2018-10-19] [4101818] | Add title
* [2018-10-19] [abc59f1] | Add hello message
* [2018-10-18] [067466c] | Add .gitignore      (master)
* [2018-10-18] [ae213ff] | First commit
```

Cherry-pick

```
$ git checkout master
$ git cherry-pick 0a5354e
[master 0a32b79] Bug: #1324: login broken
Date: Fri Oct 19 11:35:41 2018 +0200
1 file changed, 1 insertion(+)
```

Cherry-pick

```
$ git hist master feature1
* [2018-10-19] [0a32b79] | Bug: #1324: login broken      (HEAD -> master)
| * [2018-10-19] [0a5354e] | Bug: #1324: login broken      (feature1)
| * [2018-10-19] [a0f4729] | Last commit
| * [2018-10-19] [12b126f] | Almost last commit
| * [2018-10-19] [4101818] | Add title
| * [2018-10-19] [abc59f1] | Add hello message
| /
* [2018-10-18] [067466c] | Add .gitignore
* [2018-10-18] [ae213ff] | First commit
```

Partage du travail et collaboration

git merge

Partage du travail et collaboration

- Mise en place d'un dépôt distant
- Publier ses modifications
- Récupération du travail de l'équipe
- Les branches de suivi. Gestion des échecs
- Les submodules

Mise en place d'un dépôt distant

Nécessaire pour partager du code avec des collaborateurs distants de manière privée.

Exemple avec une distribution Linux

- Créer un utilisateur **git**
- Sécuriser les accès par **ssh** : ajouter les clés publiques de chaque collaborateur au fichier « `authorized_keys` »
- Initialiser un dépôt

```
$ sudo adduser git
$ su git
$ cd
$ vim .ssh/authorized_keys
$ mkdir mon-depot.git && cd $_
$ git init --bare
```

Mise en place d'un dépôt distant

A l'initialisation, on crée un **premier commit**, et nos **branches** sur le dépôt distant

```
$ git clone git@mon-serveur.com:mon-depot.git && cd mon-depot  
$ git add README.md  
$ git commit -m 'initial commit'  
$ git push origin master  
$ git checkout -b r0.0.1  
$ git push -u origin r0.0.1
```

Chaque développeur autorisé pourra ensuite récupérer ce dépôt

```
$ git clone git@mon-serveur.com:mon-depot.git
```

Partage du travail et collaboration

- Mise en place d'un dépôt distant
- Publier ses modifications
- Récupération du travail de l'équipe
- Les branches de suivi. Gestion des échecs
- Les submodules

Publier ses modifications

« *On commit souvent, on optimise plus tard, et enfin on publie* »

Une fois la fonctionnalité finalisée on peut la publier (pousse la modification de la **branche courante** vers le serveur)

On doit spécifier le nom du **remote** et la **branche** si la branche courante n'est pas trackée

```
$ git push  
$ git push <dépôt> <branche>  
$ git push github dev
```

Push force. Ré-écriture d'historique distant.

```
$ git push -f (--force)
```

Publier ses modifications

Conventions et bonne pratiques

- Est-ce que ça **compile** ?
- Lancer les **tests avant**
- Vérifier que les messages des commits sont **compréhensibles**
- **Relire** ses modifications
- Des **hooks** peuvent aider à se prémunir de ce genre de situations (pre-commit, pre-push)
- Le **push force**, à utiliser seulement si vous savez ce que vous faites !



Partage du travail et collaboration

- Mise en place d'un dépôt distant
- Publier ses modifications
- Récupération du travail de l'équipe
- Les branches de suivi. Gestion des échecs
- Les submodules

Récupération du travail de l'équipe

Toute **branche publiée** sur le dépôt peut être rapatriée sur le poste d'un autre développeur

Pour se mettre à jour, puis lister les branches

```
$ git fetch origin -p  
$ git branch -a
```

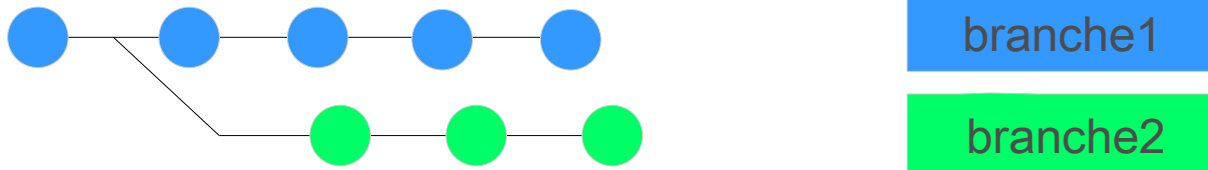
On peut alors se placer sur la branche que l'on souhaite récupérer

```
$ git checkout la-branche-souhaitee
```

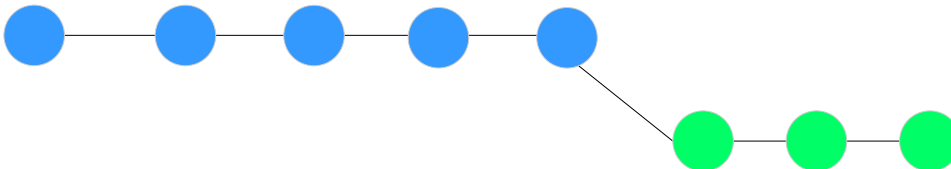
Récupération du travail de l'équipe

Pour se maintenir à jour d'une branche parent (souvent utile si on a du retard)
Supposons que « branche2 » dérive de « branche1 », et que la branche courante est « branche2 »

```
$ git rebase branche1
```



Les commits de « branche1 » seront appliqués à « branche2 » juste après la divergence



Récupération du travail de l'équipe

On peut utiliser ce système de rebase pour se mettre à jour de la même branche distante (branche de suivi)

```
$ git pull --rebase
```

De la même façon, tous les **commits publiés** que vous n'avez pas rapatriés seront appliqués **avant vos modifications**

Les commits sont appliqués **un par un**

Attention : comme pour le merge, des **conflits** peuvent arriver

Partage du travail et collaboration

- Mise en place d'un dépôt distant
- Publier ses modifications
- Récupération du travail de l'équipe
- Les branches de suivi. Gestion des échecs
- Les submodules

Les branches de suivi. Gestion des échecs

Une branche de suivi est simplement une **branche locale** connectée à une **branche distante**

Pour des raisons de simplicité, on garde le même nom en local et distant

Différents développeurs peuvent publier sur la **même branche**, il peut y avoir des conflits

Attention : commit de merge ? Push force ?

Le push force est à éviter, sauf si ...

- Vous êtes très peu sur cette branche et communiquez avec vos collaborateurs
- Vous souhaitez ré-écrire l'historique

Les branches de suivi. Gestion des échecs

Avoir un **schéma de branches des conventions**, et un **processus** bien défini est très important pour s'y retrouver

Noms des branches

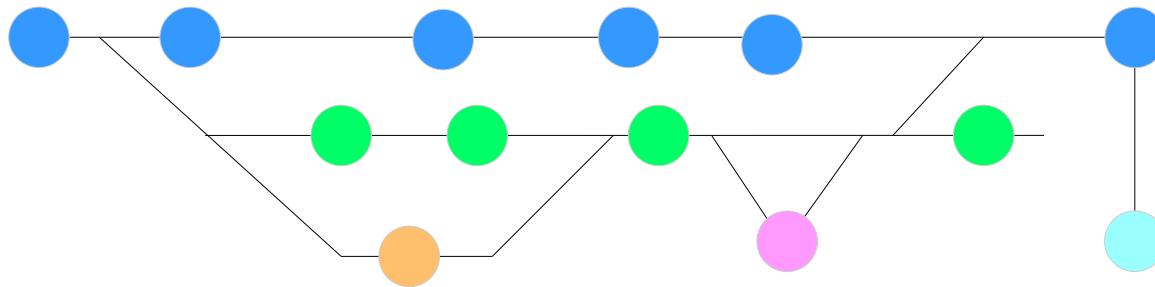
- feature/ma-fonctionnalité
- hotfix/bug-2234
- release/r-0.0.1

Préfixes, numéro du ticket, mots clés...

A mettre en place et à documenter pour l'arrivée de nouveaux collaborateurs

Les branches de suivi. Gestion des échecs

Schéma et processus



master

dev

bugfix-023

bugfix-024

Release-1.0

Une fois fusionnées, certaines branches peuvent être supprimées du dépôt distant

```
$ git push origin --delete bugfix-023
```

Partage du travail et collaboration

- Mise en place d'un dépôt distant
- Publier ses modifications
- Récupération du travail de l'équipe
- Les branches de suivi. Gestion des échecs
- Les submodules

Les submodules

Pour les projets de taille conséquente

C'est finalement un dépôt dans un autre dépôt

Ré-utilisation d'un autre projet à l'intérieur d'un autre projet

- Pas de duplication du code
- Bénéficie des dernières modifications du projet
- Choix de la version
- Séparation des messages de commit

Est-ce la seule solution ?

- Dépend du langage et de son gestionnaire de paquets.

Les submodules

Exemple typique de gestion de multiples submodules:

Branch: master ▾		New pull request	Create new
digitalLumberjack committed on GitHub Merge pull request #1183 from lorecast162/patch-1 ...			
📁 .github	Create ISSUE_TEMPLATE		
📁 recalbox-buildroot @ 2db76d4	prepare 4.0.0 release		
📁 recalbox-configgen @ 9e38ef3	prepare 4.0.0 release		
📁 recalbox-emulationstation @ 00755aa	prepare 4.0.0 release		
📁 recalbox-rescue @ b7b10b6	prepare 4.0.0 release		
📁 wiki	added fba-libretro 0.2.97.37 gamelist file		
📄 .gitmodules	prepare 4.0.0 release		

Les submodules

Initialisation d'un submodule

- Ajout initial : `git submodule add <url> <chemin>`

```
$ git submodule add https://uri/to/repo.git mySubModule
```

Clone d'un dépôt contenant des submodules

- Option `--recursive`

```
$ git clone --recursive <url>
```

Les submodules

Synchronisation d'un submodule dans son parent et récupération du travail des autres collaborateurs

```
# On se place dans le projet parent
$ git pull
$ git submodule sync --recursive
$ git submodule update --init --recursive
```

Les submodules

Mise à jour de la version d'un submodule dans son parent

```
# On se place dans le submodule
$ cd /path/to/submodule
$ git fetch
$ git checkout -q <commit-sha1>

# On se replace dans le projet parent
$ git add mySubModule
$ git commit -am "Bump submodule version"
$ git push
```


Les submodules

Pour résumer, les submodules sont intéressants pour

- Pallier au manque d'un gestionnaire de dépendances (npm, maven, etc.)
- Gérer un gros projet utilisant plusieurs composants de nature différente
- Séparer les historiques

Mais...

- Plus de travail pour le développeur
- Plus de commandes à gérer
- Compréhension des mécanismes des submodules par toute l'équipe de dev

Mise en oeuvre des outils GIT

S'équiper

Mise en oeuvre des outils GIT

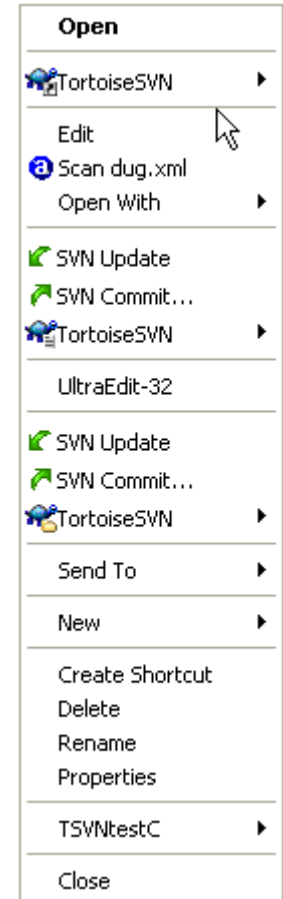
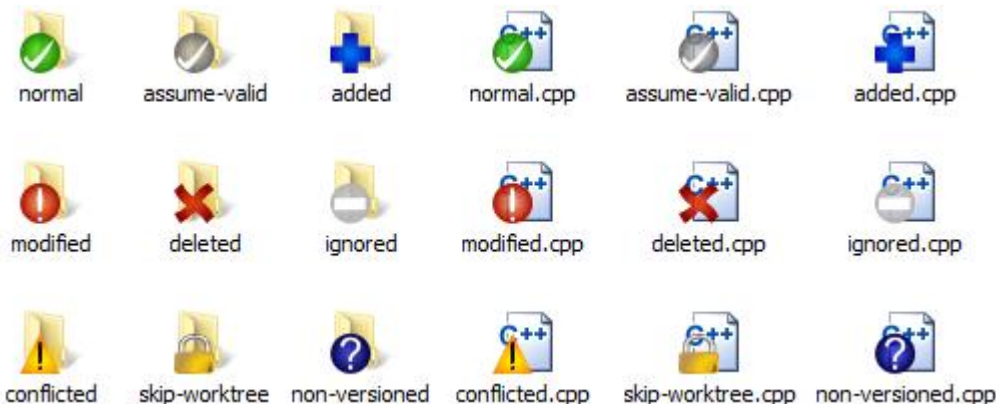
- Git-gui et TortoiseGIT – navigation graphique
- GITWeb – navigation graphique
- GitHub – service d'hébergement
- Gerrit – revue de code

Git-gui et TortoiseGIT

Plusieurs **outils graphiques** peuvent être utilisés selon le système

TortoiseGIT est disponible sur **windows**

- Intégration des **commandes Git** dans l'**explorateur** et les **menus contextuels**



Git-gui et TortoiseGIT

Avantages de TortoiseGIT

- Bien intégré dans Windows
- Facile d'accès
- Rapide
- Open source (GPL)

En revanche

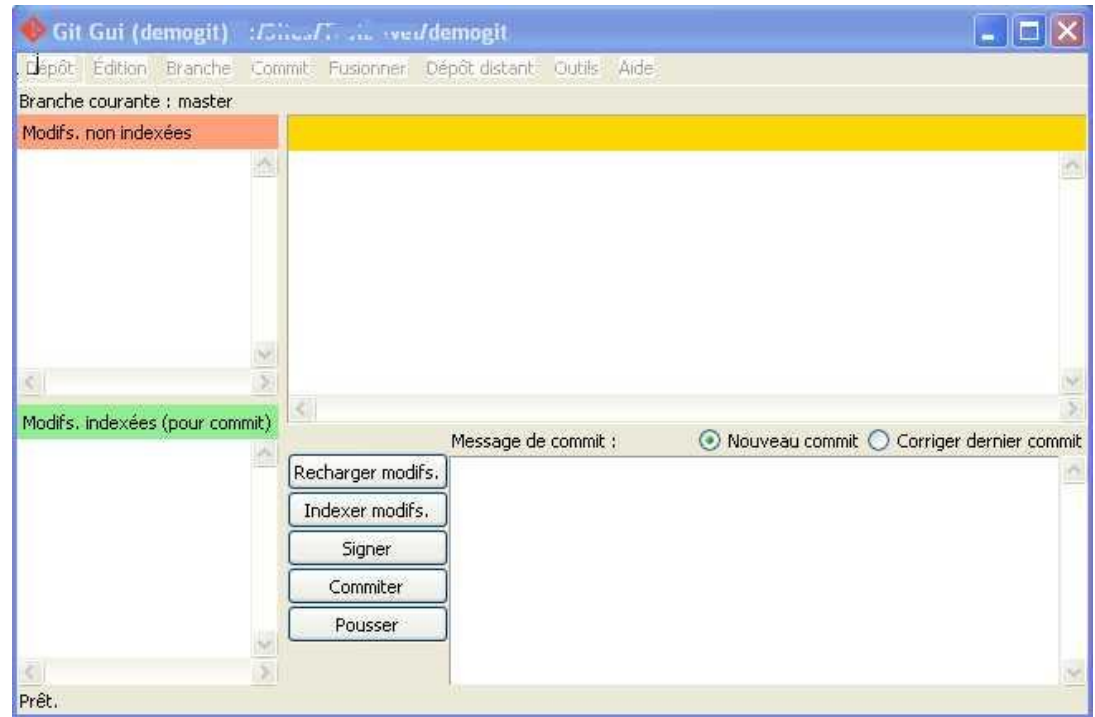
- Windows ...
- Erreurs parfois difficiles à comprendre ?

Git-gui et TortoiseGIT

Git Gui

Un peu plus
« roots » mais fait
le job

Disponible sur tout
système



Permet de créer des dépôts, branches, naviguer, rechercher, commiter, publier ...

Git-gui et TortoiseGIT

Avantages de Git Gui

- Très portable
- Rapide
- Simple

En revanche

- Un peu roots en terme de design

Git-gui et TortoiseGIT

Les outils graphiques apportent un certain **confort**

La liste est assez longue

GitHub Desktop, Tower, Gitbox, etc...

<https://git-scm.com/downloads/guis>

Et n'oubliez pas pour autant
la **ligne de commandes** !



Mise en oeuvre des outils GIT

- Git-gui et TortoiseGIT – navigation graphique
- GiTWeb, GitLab, Gitorious – navigation graphique
- GiTHub, Bitbucket – services d'hébergement
- Gerrit – revue de code

GitWeb

« Git web interface »

Une **interface web** pour naviguer dans votre dépôt

- Navigation entre branches
- Contenu des fichiers
- Logs
- RSS
- Recherche

<https://git-scm.com/docs/gitweb>

GitWeb

Nécessite un serveur http : Apache, nginx, etc.

Mise en place selon les systèmes :

- Sur Windows, avec msysgit
<https://git.wiki.kernel.org/index.php/MSysGit:GitWeb>

- Sur les Debian-like

```
$ sudo apt-get install gitweb
```

Puis configuration d'un serveur http local

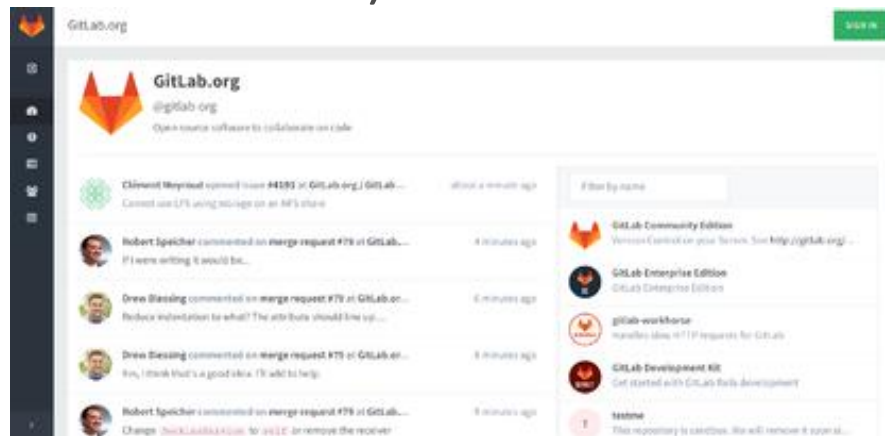
GitLab

Scindé en deux :

- GitLab Community Edition
- GitLab Enterprise Edition

Écrit en ruby

Simple à installer car intègre les éléments d'infrastructure (base de données, serveur web...)



Gitorious

Forge complète autour de git

L'offre hébergée est aujourd'hui arrêtée

Racheté en 2015 par Gitlab

Mise en oeuvre des outils GIT

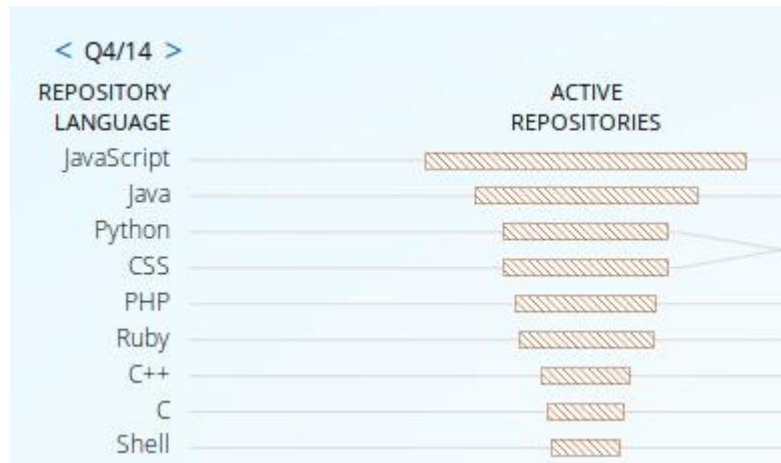
- Git-gui et TortoiseGIT – navigation graphique
- GiTWeb, GitLab, Gitorious – navigation graphique
- GiTHub, Bitbucket – service d'hébergement
- Gerrit – revue de code

GitHub

Utilise Git évidemment

Propose **l'hébergement** gratuit de vos dépôts

Utilisé massivement (2.2 millions de dépôts)



GitHub

Apporte des **fonctionnalités** supplémentaires

- Bug tracker
- Wiki
- Site web pour le projet (sous domaine)
- **Discussions** autour d'une ligne de code en particulier
- Recherche dans le code ou les « issues »
- Gestion de « pull requests »
- Des stats
- Et des plugins navigateurs pour GitHub ...

Ce qu'on n'aime pas ...

- Publier son dépôt chez un **tiers**
- Si GitHub tombe demain ?

Bitbucket

Concurrent de Github

- Orienté « repo » privés
- Reprend le concept de pull request
- Droits d'accès fins (sur branches par ex)
- Et bien sûr s'intègre avec les logiciels Atlassian (JIRA...)

Github reste la référence pour l'open source

Mixer services hébergés et internes

Git est décentralisé

- Chaque utilisateur possède son dépôt
- En conséquence plusieurs « bare repos » peuvent cohabiter sans difficulté
- Cas usage : branche open source sur Github et branches privées en interne
- Pour ajouter un dépôt distant

```
git remote add {name} {url}
```

- Pour lister les dépôts enregistrés

```
git remote -v
```

Mise en oeuvre des outils GIT

- Git-gui et TortoiseGIT – navigation graphique
- GiTWeb, GitLab, Gitorious – navigation graphique
- GiTHub, Bitbucket – service d'hébergement
- Gerrit – revue de code

Gerrit

Plateforme utilisant Git et y apportant des fonctionnalités

« Zone distante temporaire »

- Discussion à propos du code publié
- Acception de ce code par l'équipe et « vrai publication »

Enrichi les processus de développement

Gerrit

La différence avec GitHub réside dans la **modélisation des changements**
Moins utilisé cependant

Pour gerrit, chaque commit est un changement, généralement ré-écrits après discussions, puis « réellement publiés »

Dans GitHub, le système de pull requests consiste à fusionner un ensemble de modifications

<https://github.com/blog/1124-how-we-use-pull-requests-to-build-github>

Impacts organisationnels de git

Modalité de travail

Evolution ou révolution ?

Une révolution !

L'impact n'est pas que technique mais organisationnel

Changement profond de notre façon de travailler

- Délinéarisation des développements
- Intégration facilitée des contributions externes
- L'ère du « social coding » a démarré

Délinéarisation des développements

Il est enfin facile de créer des branches, ce qui permet de :

- D'explorer des pistes de développement
- D'isoler les correctifs pour les appliquer ensuite aux multiples versions maintenues
- Favoriser le travail massivement distribué

Rappel : définir un « branching model » est indispensable

Intégration des contributions externes

Avec des droits restreints (lecture seule), il est possible de :

1. Cloner le dépôt
2. Travailler de son côté
3. Tester l'ensemble
4. Proposer la réintégration de ses modifications

Il n'y a donc pas de risque à s'ouvrir, pour stimuler ces contributions :

- Des règles de gouvernance pourront être édictées
- Les contributeurs méritants pourront se voir attribuer des droits avancés sur le dépôt

Le social coding

Si la compétence reste bien sûr nécessaire, participer à un même développement devient aussi simple que de poster un commentaire, publier une photo...

Le monde entier peut dorénavant collaborer !

Fin de la formation

Merci pour votre participation

Liens et ressources

La doc !

<https://git-scm.com/documentation>

Quelques guides chez GitHub

<https://guides.github.com/>

Tutorials

<http://gitimmersion.com/>

<http://rogerdudler.github.io/git-guide/>

Liens et ressources

Références

<http://gitref.org/>

La bible de Git

<https://git-scm.com/book>

Stackoverflow (You shall not copy paste !)

<http://stackoverflow.com/questions/tagged/git>

Git Flow

<http://danielkummer.github.io/git-flow-cheatsheet/>

Les points essentiels

Utiliser des **conventions**

Respecter les **processus** en place

Faites des branches, 5 fois par jour !

Commit fréquent, push moins souvent

Une erreur commise ?

Pas de panique, il y a **toujours une solution** avec Git (ou presque !)

