P.PILLA

OUTIL DE TRAVAIL COLLABORATIF ET DE VERSIONNING

GIT

Vous consulterez avec intérêt les sites suivants :

http://gitref.org/ (explication des commandes)

https://www.miximum.fr/blog/enfin-comprendre-git/ (Explications sur le fonctionnement interne de GIT) - des schémas utilisés dans ce document proviennent de ce site.

https://git-scm.com **: aide en ligne GIT**

Démarrage :

1. Je crée un repository sur le serveur (repository = dépôt)
2. En local :
    a. Initialisation du repository local comme repository git : git init
    b. je renseigne mes coordonnées personnelles
        git config --global user.name "Prénom Nom"
        git config --global user.email moi@example.com
    c. Enregistrement de l'url du repository distant et je lui attribue un alias (origin) :

git remote add origin https://pillap69@bitbucket.org/pillap69/isi1-web-movies.git

Vous vérifiez que la bonne url a été attribuée à l'alias origin : git remote -v

Si vous vous êtes trompé, utiliser la commande suivante pour réécrire l'url de l'alias origin :

git remote set-url origin https://pillap69@bitbucket.org/pillap69/isi1-web-movies.git

    d. Je rajoute tous mes fichiers à la zone de staging (index) : git add *
    e. Je rajoute les fichiers qui sont dans l'index dans la base de données de git en local ce qui crée une branche appelée 'master':

git commit –m "commit initial"

    f. Je pousse tous les fichiers vers la branche 'master' du repository distant :

git push origin master

A l'issu de cela tous mes fichiers sont sur le serveur

Remarque : dans le cas où j'ai un repository distant existant avec des fichiers programme je peux demander un clone du repository avec la commande :

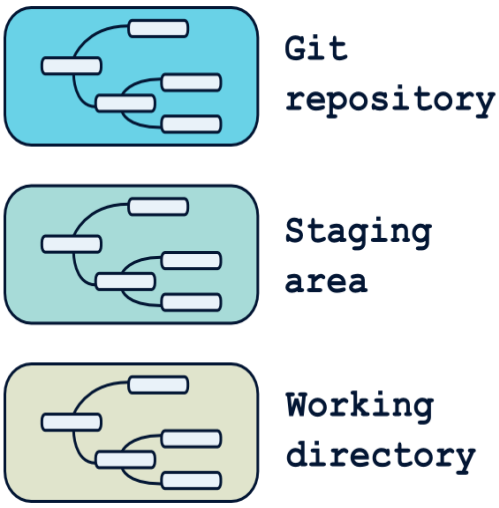
a. git clone https://pillap69@bitbucket.org/pillap69/isi1-web-movies.git
b. je renseigne mes coordonnées personnelles
    git config --global user.name "Prénom Nom"
    git config --global user.email moi@example.com

Version du : 04/01/2017

Cette commande me créera un repository local qui sera la copie exacte du repository git distant et bien entendu déjà initialisé comme repository git.

Remarques : les coordonnées saisies sont inscrites dans un fichier qui se trouve sous windows à la racine du profil utilisateur noté : ~/.gitconfig . Ce fichier contiendra le nom et l'email de l'utilisateur et permettra de savoir qui est à l'origine des modifications quand un développeur en fera la demande avec des commandes comme **"git log"** qui liste l'ensemble des commit avec leur commentaire et le nom/email de l'utilisateur , **"git blame nom_fichier.php"** qui liste les modifications sur un fichier en précisant le début de l'empreinte sha1 du commit, le nom/email de l'utilisateur, mais aussi **"git show extrait_empreinte_sha1"** qui permet d'avoir le détail d'une modification pour un commit donné. On voit que ces renseignements donnés à l'aide de "git config –global" sont importants et permettent ultérieurement une bonne gestion du code.

Architecture générale GIT



On voit à partir du schéma ci-dessus que GIT s'articule autour de trois zones, dans la suite de ce document je parlerai, par analogie, de 1er,2ème et 3ème étage. Le **répertoire de travail** qui est votre répertoire physique, *la* **zone dîtes de staging ou index** qui est la zone intermédiaire dans laquelle vous rajouterez le(s) fichier(s) avant qu'il(s) soi(en)t entreposé(s) dans **le repository git ou dépôt git en français** en réalisant un commit, dernière étape avant de mettre le(s) fichier(s) dans le repository distant.

Rajouter des fichiers dans la zone de staging puis dans le repository GIT local

Pour ajouter des fichiers dans la zone de de staging ou index (2ème étage) : git add
On distingue :

- git add -A ou git add *: Cela permet de rajouter tous les changements opérés dans le répertoire de travail (A ⇔ ALL) c'est-à-dire les nouveaux , les modifications et les suppressions.
- git add . : Tous sauf les suppressions
- git add -u : Tous sauf les nouveaux

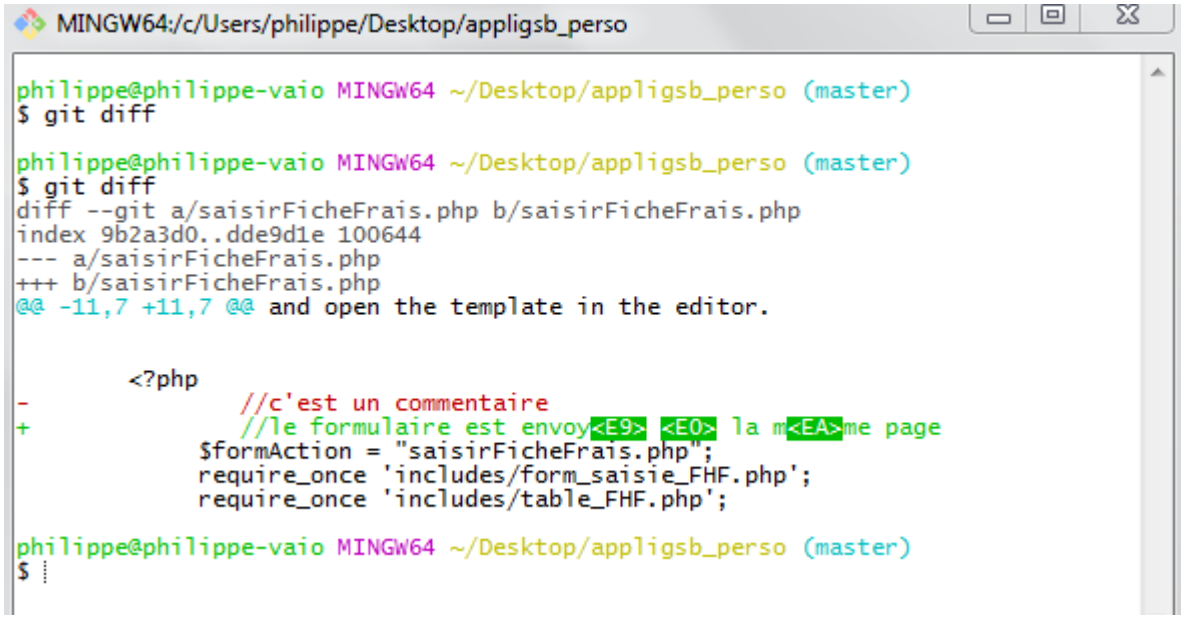
Pour ajouter des fichiers dans la répertoire GIT (3ème étage) : git commit -m "message du commit "

Noter que l'on peut combiner les deux commandes, ajouter à l'index **et** dans le repository git local avec la commande : git commit -am "message de commit". Toutefois cette commande ne rajoutera dans l'index que les informations concernant les fichiers modifiés et supprimés. Cette commande sera donc la contraction d'un "git add -u" suivi d'un "git commit -m 'message du commit'".

Attention : "commiter" au maximum les fichiers un par un de manière à avoir des messages de commit correspondant à une modification intervenue dans un fichier. Cela sera utile lorsque vous voudrez revenir à une version antérieure à l'aide des instructions que nous verrons plus loin dans ce document.

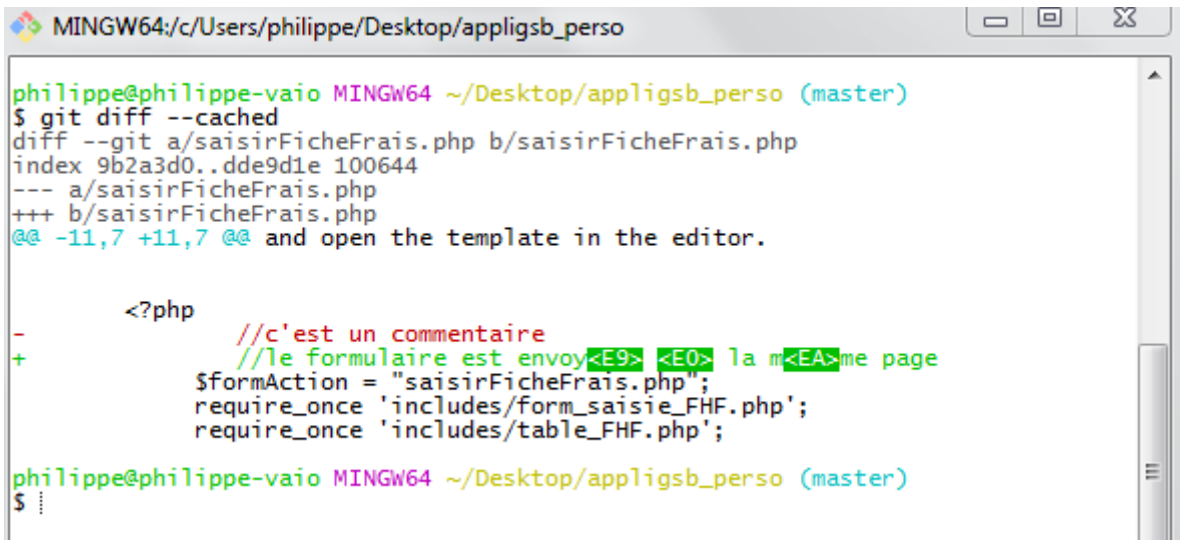
Savoir où nous en sommes : j'ai ajouté à l'index ? J'ai fait un commit ?

1. Différence de contenu entre votre répertoire de travail et la zone de staging : git diff



On remarque sur la figure ci-dessus que la commande 'git diff' fait apparaître une modification sur le fichier 'saisirFicheFrais.php'. C'est une ligne de commentaire qui a été modifiée. Il faudra donc faire une commande 'git add' afin de rajouter le fichier à l'index.

2. Différence de contenu entre votre zone de staging et votre repository git : git diff –cached



On remarque ici la même modification que ci-dessus mais cette fois-ci avec l'instruction 'git diff --cached', cela implique que le fichier saisirFicheFrais;php a été rajouté à l'index maintenant mais que ce fichier n'a pas fait l'objet d'un commit. Il faut donc faire 'git commit -m "modif commentaire dans saisieFicheFrais.php ligne 14" '

Pour avoir une vue globale de l'état relatif des différents étages entre eux : git status . Cette instruction donne un état concis de ce que git diff et git diff –cached affichent.

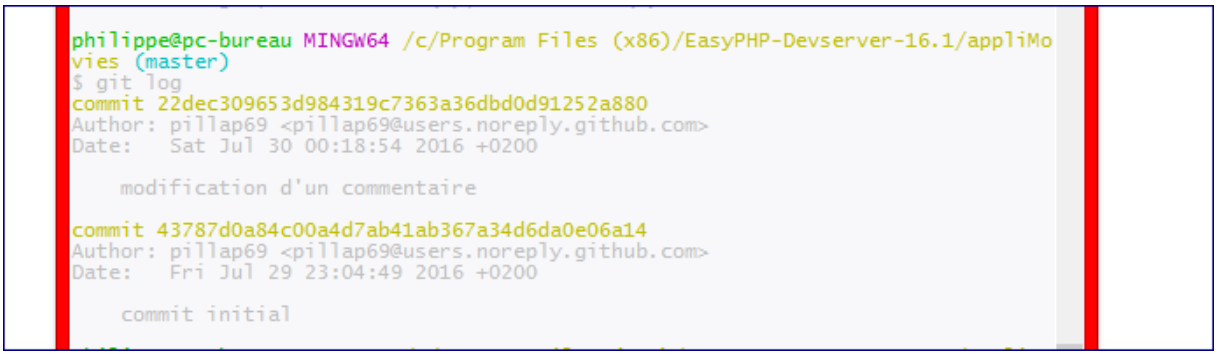
Travailler dans une autre branche que la branche 'master'.

Quand on commence à travailler on est dans la branche 'master'. Nous voulons faire des modifications, un correctif par exemple, mais sans impacter la branche 'master', nous décidons alors de créer une autre branche par exemple v.1.0.1 - A noter dès à présent que cette nouvelle branche ainsi créée **pourra être intégrée par fusion** dans la branche master comme nous le verrons un peu plus loin dans ce document.

Une définition intéressante d'une branche : **Une branche n'est qu'une étiquette qui pointe vers un commit**. Pas très parlante comme cela, cette définition va cependant nous permettre de comprendre en profondeur ce qu'est un commit et une branche et le rapport entre les deux.

Partons du commit pour bien comprendre : je fais un commit avec git commit. Si j'ai déjà fait un commit la commande 'git log' va me lister l'ensemble des commit réalisés et leur référence respective qui se trouve être une empreinte sha1 correspondant au commit.

Liste des commit successifs avec la commande 'git log'



On voit sur la figure ci-dessus deux commit avec leur référence respective de la branche « master ».

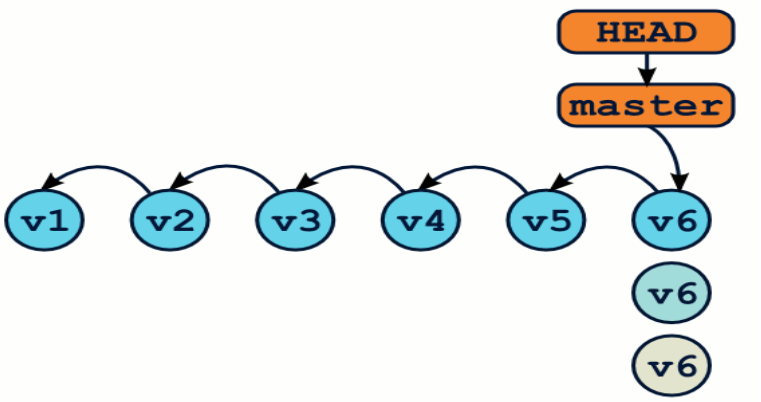
Si l'on sait qu'un commit comprend entre autres, de manière compressée l'ensemble des fichiers au moment du commit on peut comprendre qu'en se positionnant sur un commit précédent nous pouvons annuler les modifications des commit suivants et nous retrouver avec nos anciennes versions de fichier.

Ici en se positionnant sur le commit dont le message est "commit initial" j'annule en quelque sorte la modification du commit suivant dont le message est "modification d'un commentaire". La branche master pointe sur le dernier commit.

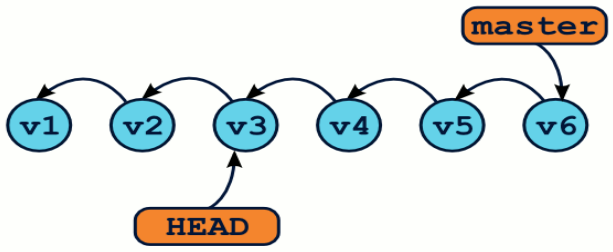
```
git checkout 43787d0a84c00a4d7ab41ab367a34d6da0e06a14
```

la branche master pointe toujours vers le dernier commit mais je n'ai à l'écran avec la commande 'git log' que le premier commit. Pourquoi ? C'est ce que l'on appelle l'état **'detached head'** du nom de la référence qui pointe sur master et qui se nomme HEAD. Faisons un schéma :

La référence Head pointe vers la branche Master – 'Etat initial'



La référence Head pointe vers un commit précédent – Head est détaché de Master



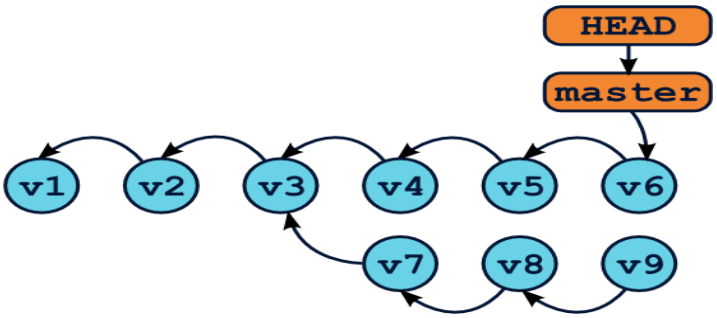
HEAD pointe vers commit N-1 . MASTER continue de pointer vers commit N. J'ai donc détaché HEAD de master (detached head). Une commande dont nous discuterons un peu plus loin (git reset) dans le document nous permettra de faire pointer MASTER sur 'commit N-1' définitivement si nous le désirons.

Créer une nouvelle branche pointant vers un commit antérieur.

Dans cet état 'detached head' je peux réaliser des commit qui n'auront aucun impact sur mes branches existantes. Je peux donc faire des essais, faire des commit et seul la référence HEAD pointera sur le dernier commit réalisé. Dès que je reviendrai sur master avec la commande 'git checkout' (head pointera alors sur master) les commit ainsi réalisés seront soit éliminés par le **GARBAGE COLLECTOR de Git** si je n'associe aucune référence de branche dessus soit si je le désire je peux créer une nouvelle branche qui pointera vers ce dernier commit et ainsi retrouver mon travail, c'est-à-dire l'ensemble des commit que j'ai créé en état 'detached head' .

les commit V7 V8 et V9 sont perdus – Aucune référence dessus

*:



Comment retrouver la référence vers V9 ? Il faut utiliser la commande 'git reflog', elle va nous permettre d'obtenir la référence (empreinte sha1) de ce commit, en fait elle nous indique la dernière référence quittée avant le retour vers la branche master. Une fois cette référence obtenue nous pouvons créer une nouvelle branche (étiquette de branche) qui pointera ici sur V9 en validant la commande suivante : git branch nom_nouvelle_branche sha1_V9 .

Remarque : Je peux ***lister l'ensemble des branches*** avec la commande : git branch

Revenons maintenant à l'état initial HEAD pointe MASTER qui lui-même pointe 'commit N': git checkout master.

Revenons sur la définition de la branche de la page précédente une branche est bien une référence pointant sur un commit. Cette référence 'branche' est-elle-même pointée par une autre référence qui se nomme HEAD. Quand on se trouve dans une branche, **on dit que la branche est extraite**, cette branche est alors forcément pointée par HEAD.

Que se passe-t-il si je crée une nouvelle branche pour les raisons indiquées un peu plus haut dans ce document ?

Créer une branche : git branch v.1.0.1
Se positionner sur une branche : git checkout v1.0.1
Nous pouvons aussi créer et nous positionner sur la branche en une seule commande :
git checkout –b v1.0.1
Si je valide une commande 'ls' sur la branche nouvellement créée on s'aperçoit qu'elle n'est pas vide et qu'elle contient les derniers fichiers commités dans la branche 'master'. Ainsi la référence 'HEAD' pointe maintenant sur v1.0.1 et v1.0.1 pointe maintenant sur le dernier commit ou pointe aussi 'master'.

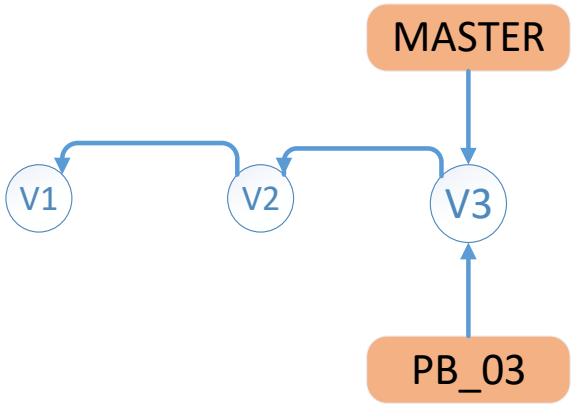
Poussée la branche nouvellement créer vers le serveur (branche v.1.0.1)
git push origin v.1.0.1
Remarque : tant que le branche n'est pas poussée, elle reste locale et n'est donc pas divulguée au reste de l'équipe – A noter que la commande 'git push' sans préciser de nom de branche va pousser vers le serveur l'ensemble des branches.
Détruire une branche distante (branche v.1.0.1)
git push origin :v.1.0.1 (à noter l'espace après origin et avant les deux points)
Remarques :

- Si après la commande nous réalisons un 'git branch -a' la branche est encore notée comme existante à distance. Il faut alors rafraichir ses informations avec la commande 'git fetch –prune'
- Cette commande détruit la branche distante, mais **la branche locale est conservée**. Si l'on désire aussi **détruire la branche locale** : 'git branch -d v.1.0.1' ou si elle n'a pas été encore fusionnée on force la destruction avec 'git branch -D v.1.0.1'

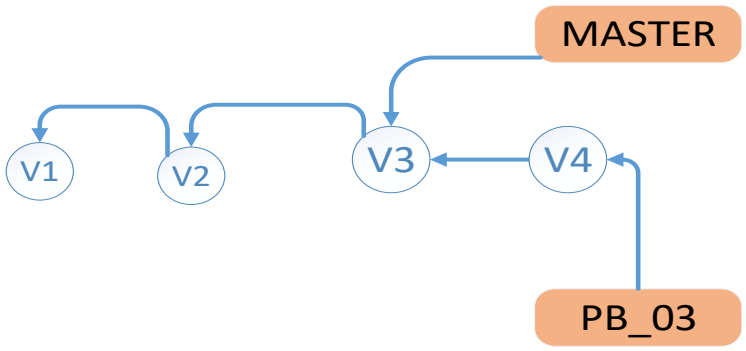
Fusionner une branche dans une autre

Imaginons une nouvelle situation dans laquelle nous décidons de créer une nouvelle branche "PB_03" à partir de master pour un problème sur l'application numéroté 03 : nous sommes sur master et faisons 'git branch PB_03'.

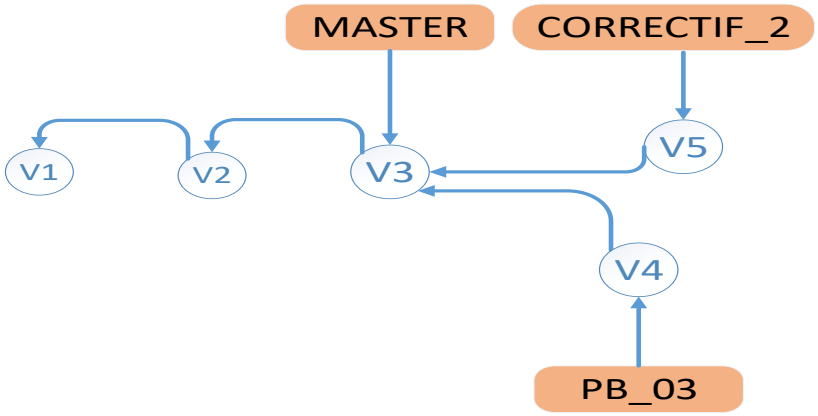
A ce moment MASTER et PB_03 pointe vers le même commit, ici V3.



Nous avançons sur le problème 03 et un nouveau commit est réalisé alors que nous nous trouvons sur la branche 'PB_03'. **Maintenant seul l'étiquette 'PB_03' pointe sur V4**, comme le montre la figure suivante



Entre-temps nous créons à partir de MASTER une nouvelle branche, correspondant au développement d'un correctif sur notre application. Cette nouvelle branche se nomme 'CORRECTIF_2'. Le travail sur CORRECTIF_2 est terminé et est commité ce qui donne le commit V5 sur lequel seul l'étiquette 'CORRECTIF_2' pointe désormais. Nous avons à ce niveau 3 branches en présence 'MASTER' 'PB_03' et 'CORRECTIF_2' comme le montre la figure suivante.

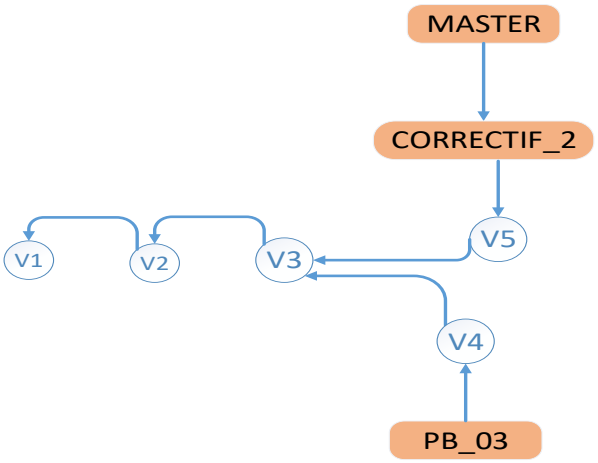


La branche 'CORRECTIF_2' peut être alors fusionnée à la branche MASTER. Pour cela nous nous positionnons dans un premier temps sur MASTER avec la commande 'git checkout master' puis nous réalisons la commande 'git merge correctif_2'. A ce moment-là 'CORRECTIF_2' et 'MASTER' pointent alors sur le commit V5 comme le montre la figure suivante. Nous sommes dans le cas le plus simple de la fusion. Pourquoi ? car le commit V5 a comme parent directement le commit ou se trouvait 'MASTER', il a suffi à GIT de simplement déplacer la référence 'MASTER' sur V5 sans véritablement fusionner c'est ce que l'on nomme une fusion 'fast forward' .

La fusion dans la console

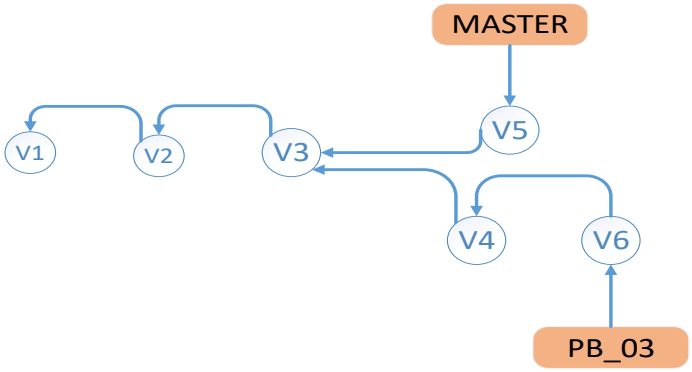
```
zebul@DESKTOP-21D5A1L MINGW64 ~/Desktop/test-GIT-ISIWEB1/test_cours (master)
$ git merge CORRECTIF_2
Updating 1213d7b..865444a
Fast-forward
 seConnecter.php | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Les étiquettes après la fusion de CORRECTIF_2 dans MASTER



A partir de cet instant la branche 'CORRECTIF_2' n'étant plus utile, elle peut être détruite par la commande 'git branch -d correctif_2'.

Par ailleurs le développement du problème 03 continue et la branche 'PB_03' pointe désormais sur le commit V6.



Nous voulons maintenant réunir le travail de la branche 'MASTER' et celui de la branche 'PB_03'. Nous voyons tout de suite que le problème va être tout autre et plus ardu pour GIT par rapport à la précédente fusion. Pourquoi ? car V5 n'est pas un parent immédiat de V6 !

Après la commande 'git merge PB_03' réalisé en étant sur 'MASTER' GIT va créer un commit spécial nommé **commit de fusion (V7)** qui aura deux ancêtres V6 et V5 et qui comprendra l'ensemble des modifications opérées dans V5 et V6 depuis leur ancêtre commun le commit V3.

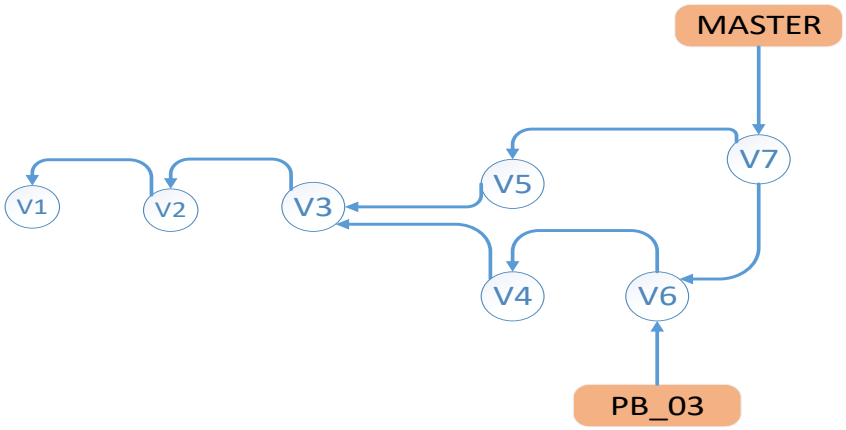
A ce moment-là deux cas de figure peuvent se présenter soit la fusion se fait automatiquement car les modifications dans V5 et V6 ne concernent pas les mêmes lignes, soit la fusion concerne des modifications opérées sur les mêmes lignes ce qui va occasionner **un conflit de fusion**. Le problème des résolutions de conflit est abordé d'une manière simple dans la dernière partie de ce document traitant du travail collaboratif.

Le commit de fusion apparait après une commande 'git log' (ici après résolution d'un conflit)

```
zebul@DESKTOP-21D5A1L MINGW64 ~/Desktop/test-GIT-ISIWEB1/test_cours (master)
$ git log
commit 2d47d941076e2d7cfaa8812361681c7248f5cc4b
Merge: 865444a 4982456
Author: Philippe Pilla <philippe.pilla@ac-lyon.fr>
Date:   Wed Jan 4 16:38:12 2017 +0100

    resolution du conflit avec PB_03
```

Commit de fusion V7



Faire pointer une référence de branche vers un autre commit : la commande 'git reset'.

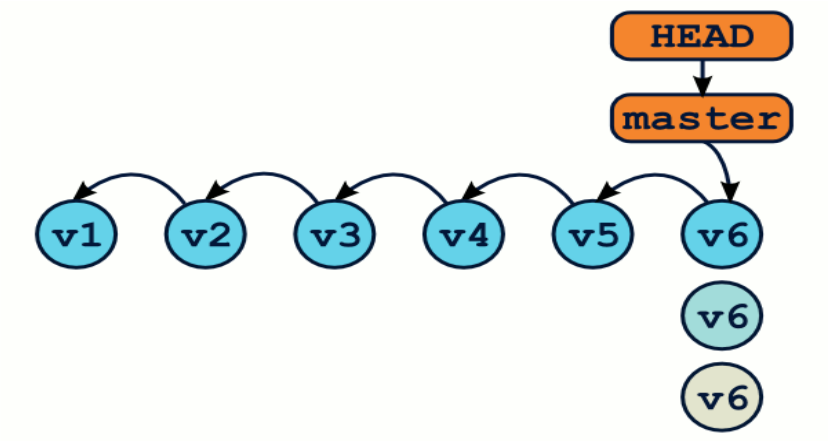
Cette commande peut être dangereuse et doit être utilisée avec précaution.

Je vais vous montrer son utilisation dans un cas précis avec l'ensemble des possibilités pour ce cas.

Je suis dans la branche master et je fais une modification que je mets dans l'index et que je commite puis je refais une modification que je mets aussi dans l'index et que je commite à nouveau.

Comment annuler mes modifications (Ce qui revient à revenir deux états en arrière de V6 à V4) ?

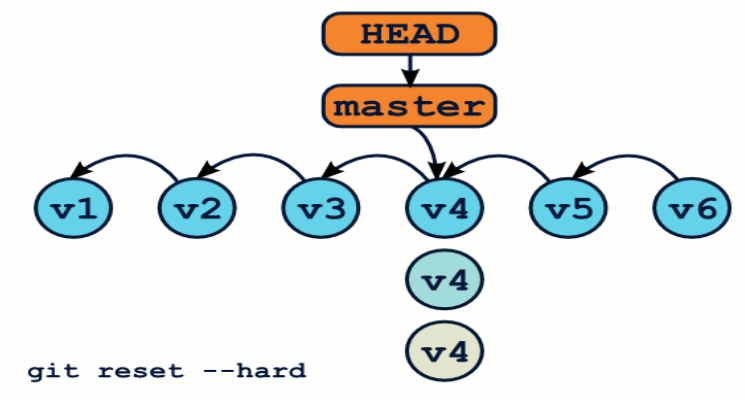
Situation de départ : HEAD pointe sur master et master pointe sur le commit V6. Chacune des trois zones est à V6 (espace de travail – index -répertoire GIT)



git log : je liste les commit – je récupère l'empreinte de l'antépénultième commit V4.

git reset empreinte_sha1_V4 –hard

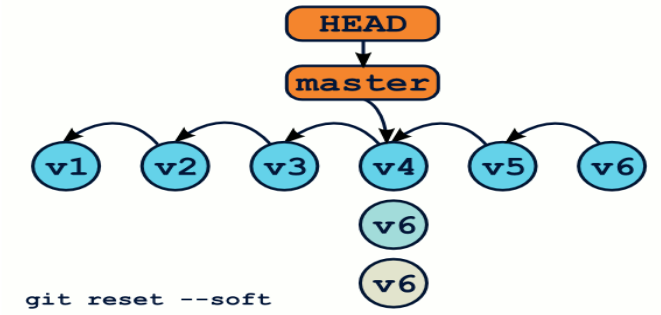
Situation d'arrivée avec l'option --hard : HEAD pointe sur master et master pointe vers le commit V4 d'adresse 'empreinte_sha1'

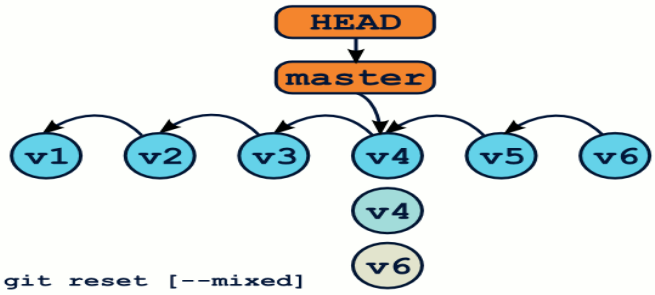


La commande reset avec l'option –hard va faire en sorte que les trois étages **(espace de travail – index -répertoire GIT)** vont revenir à l'état du commit spécifié c'est à dire V4 sur le schéma ci-dessus. Ainsi les modifications sont annulées dans les 3 étages et nous sommes revenus à l'état initial avant les deux modifications.

Il existe deux autres options (--soft et –mixed). Ces deux options permettent respectivement de revenir en arrière que pour le 3ème étage et de revenir en arrière pour le 2ème et 3ème étage.

Situation d'arrivée dans le cas de l'utilisation de l'option '--soft' ou '--mixed'

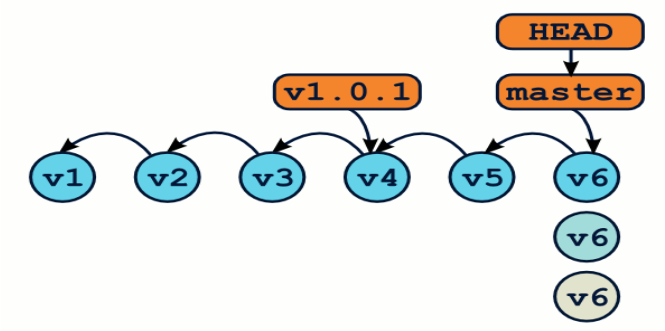




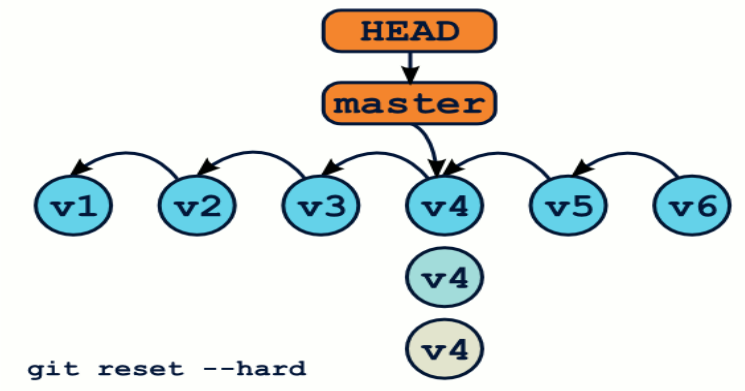
La commande **git reset peut aussi se faire avec le nom d'une branche** en lieu et place de l'empreinte sha1 d'un commit, dans ce cas les références head et master pointeront après la commande 'git reset' sur le dernier commit de cette branche.

Exemple toujours en étant sur master au départ, je veux que master pointe vers le dernier commit de la branche V1.0.1 : 'git reset V.1.0.1 –hard' fera en sorte que master et head pointe sur la même référence que V.1.0.1 et que les 3 étages de master deviennent les mêmes que ceux de V.1.0.1.

Situation de départ



Situation d'arrivée HEAD pointe sur master et master pointe sur V4 mais V1.0.1 continue aussi à pointer sur V4 (non montré sur la figure)



Travailler en collaboration avec un ou plusieurs autres développeurs

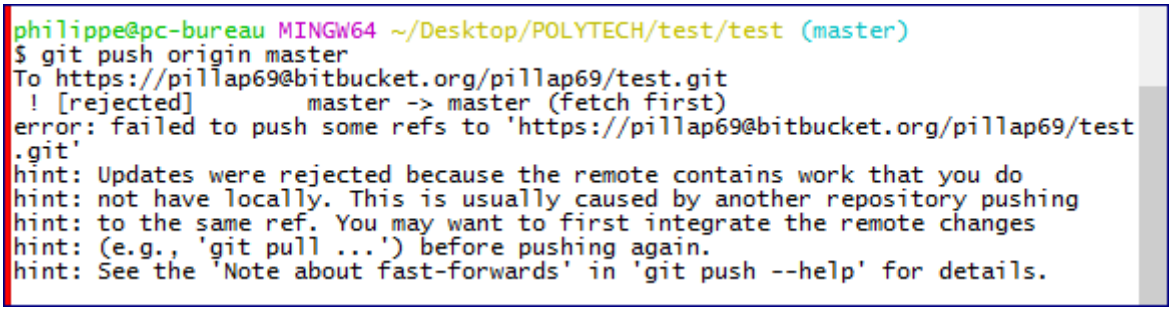
Les commandes passées en revue ci-dessus restent bien entendu valable dans le cadre d'un travail collaboratif. Mais une modification opérée en local ne peut être poussée vers le serveur (avec git push) si une modification réalisée par un autre développeur a déjà été 'pushée' sur le serveur . Dans ce cas le développeur voulant réaliser son push se voit offrir un refus lui expliquant que des modifications présentes sur le serveur ne sont pas dans son 'repository git' local. Comment résoudre ce problème ?

Le développeur réalise dans un premier temps une commande 'git fetch origin' qui lui indique les branches sur lesquelles des modifications ont été réalisées sans qu'il ne les possède en local. Le développeur repère la branche sur laquelle il travaille et réalise une commande 'git merge origin/La_branche' afin de fusionner (merge = fusionner) les données de la branche en question dans sa branche locale. Si la branche est master alors le développeur réalise une commande 'git merge origin/master'.

Dès lors les données manquantes en local ne le sont plus et le développeur peut maintenant pousser ses propres modifications opérées en local vers le serveur.

Si les deux développeurs ont travaillé sur les mêmes lignes de code la fusion ne pourra pas se faire et git vous avertira de la présence d'un conflit non géré automatiquement.

Le développeur n°1 a réalisé une modification, dans cet exemple il ajoute un commentaire. Ligne 9 il ajoute : "//c'est un commentaire" et a poussé sa modification vers le serveur. Le développeur n°2 modifie la même ligne en mettant son commentaire : "c'est un commentaire de ACER", au moment où il veut opérer la commande "git push" le serveur lui indique le conflit concernant le fichier "saisirFichefrais.php" :



Le développeur n°2 réalise alors deux commandes :

La commande "git fetch origin" : cette commande rapatrie localement les éventuels changements opérés sur la ou les branches distantes ainsi que les étiquettes (tags) sans les fusionner.

Puis la commande "git merge origin/master" qui fusionne les changements opérés sur la branche choisie ici master, la ligne en conflit apparait alors en mettant côte à côte les deux modifications séparées par "======" et à l'intérieur d'une zone délimitée par des marqueurs de conflit (conflict markers) "<<<<<<<" :

```
<<<<<<< HEAD
    <?php
    // C'est un commentaire de ACER
=======
    <?php
    //c'est un commentaire
>>>>>>> origin/master
```

Le développeur n°2 enlève les délimiteurs et garde la version qu'il pense opportune et fait un "git add" suivi d'un "git commit" **ce qui indique que le conflit est résolu** puis pousse les modifications vers le serveur.

Au prochain "git fetch/git merge" le développeur n°1 obtiendra les modifications du développeur n°2.

Remarque générale sur les conflits :

Ici le conflit concerne la branche master locale du développeur n°2 avec la branche master distante modifiée par le développeur n°1. Dans le cas d'un confit de fusion purement local entre deux branches le conflit est résolu de la même manière, sur chaque fichier impliqué il suffira d'enlever les marqueurs de conflit, de faire la modification ou de choisir un des codes existants puis d'ajouter dans l'index et enfin de commiter.