Essentiel

GIT

Table des matières

I - Présentation de GIT

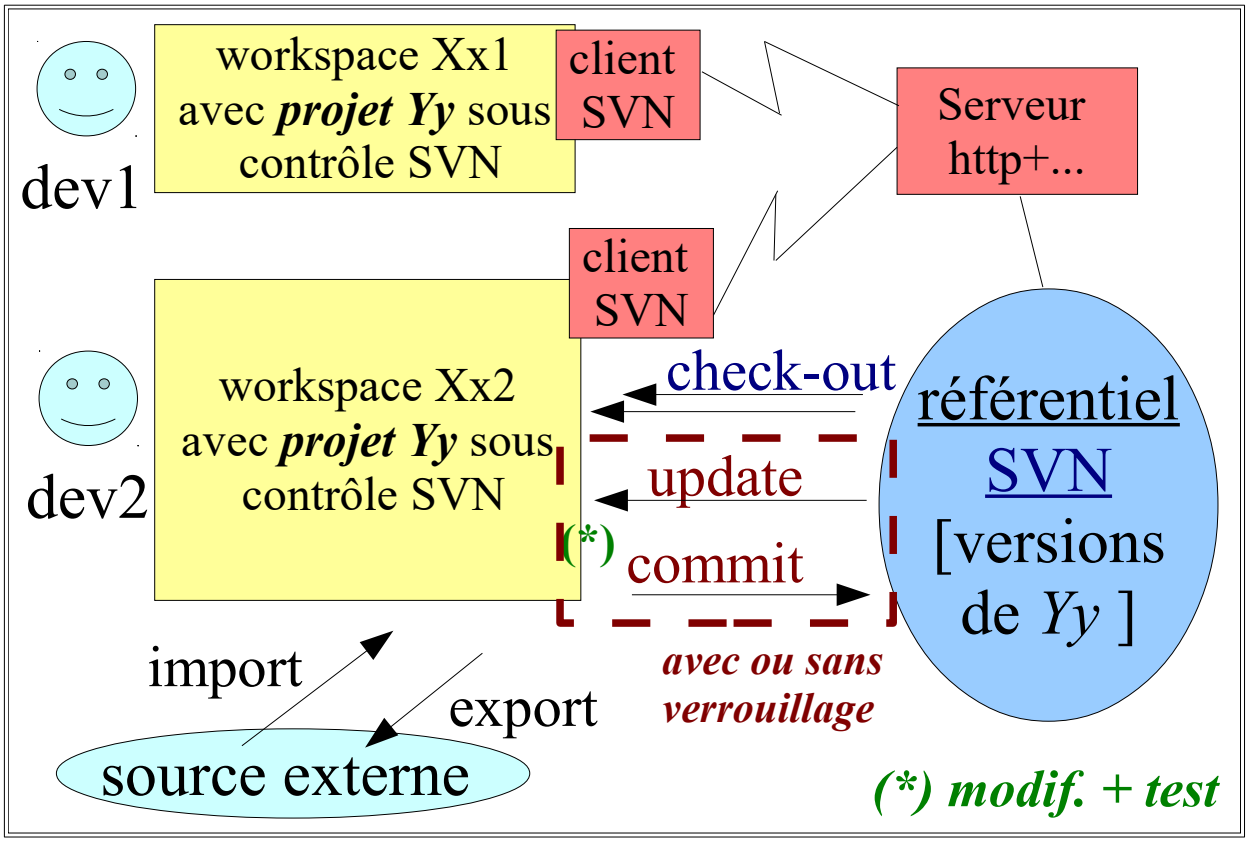
1. SVN/Subversion "centralisé" avant GIT

CVS = **C**oncurrent **V**ersion **S**ystem , **SVN** = **S**ub**V**ersio**N**

CVS est un produit "Open Source" qui permet de **gérer différentes versions d'un ensemble de fichiers sources** lié au développement d'un certain module logiciel.
Différents programmeurs peuvent travailler en équipe sur des fichiers partagés au niveau d'un référentiel commun.

SVN se veut avant tout être une "***version améliorée de CVS***" qui

- ne remet pas en cause les principes fondamentaux de CVS (référentiel commun et commit,update,...)
- a refondu l'implémentation du serveur et des référentiels (meilleur gestion des transactions, protocole d'accès plus simples, ...)



Beaucoup utilisé avant GIT , **SVN** est/était un système de gestion de code source **centralisé** .

2. Présentation de GIT

2.1. Fonctionnement distribué de GIT

Présentation de GIT

GIT est un **système de gestion du code source** (avec prise en charge des différentes **versions**) qui fonctionne en **mode distribué** .

GIT est moins centralisé que SVN . Il existe deux niveaux de référentiel GIT (local et distant).
Un référentiel GIT est plus compact qu'un référentiel SVN.

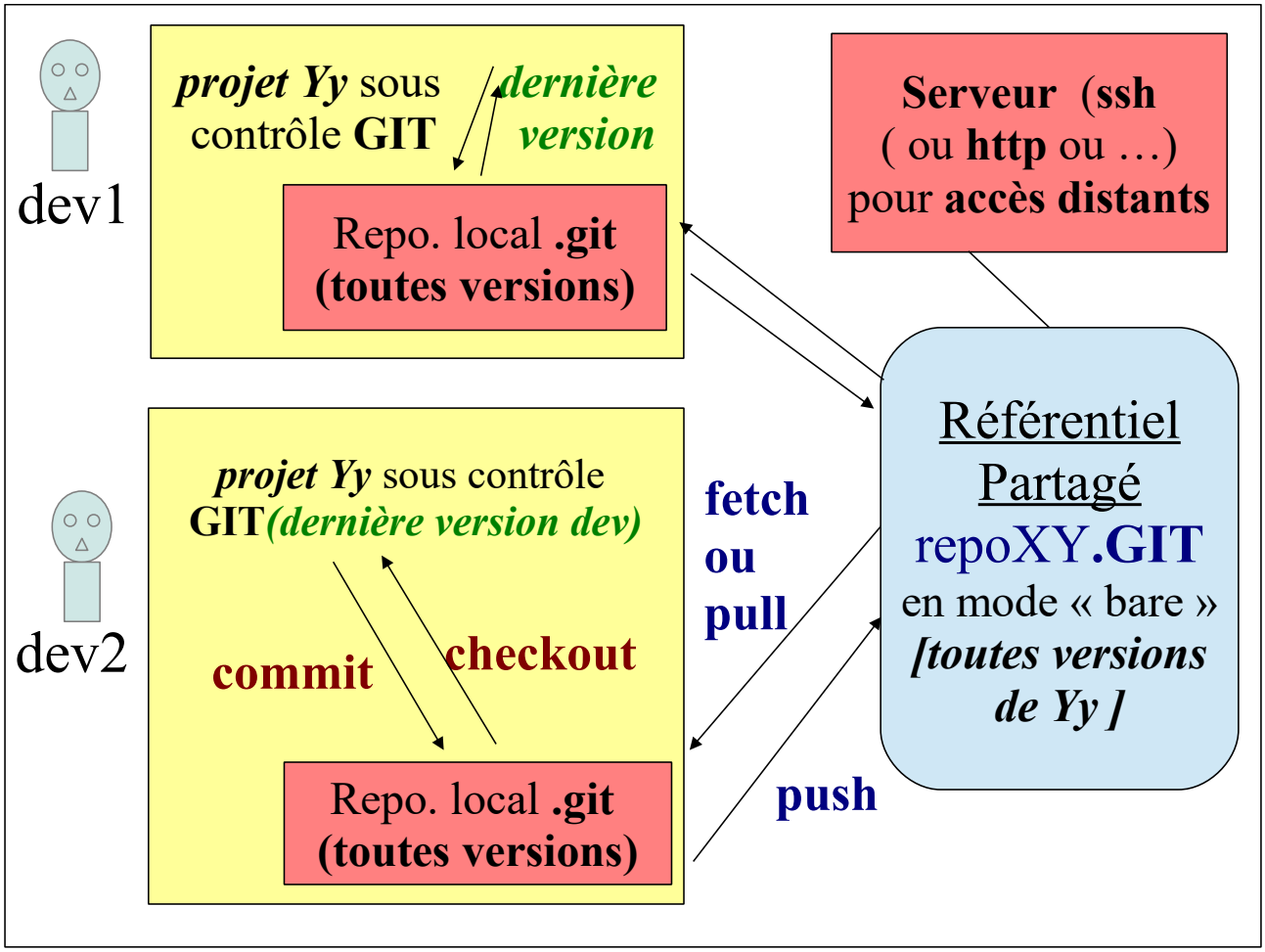
GIT a été conçu par **Linus Torvalds** (l'inventeur de **linux**) .

Un produit concurrent de GIT s'appelle « **Mercurial** » et offre à peu près les mêmes fonctionnalités.

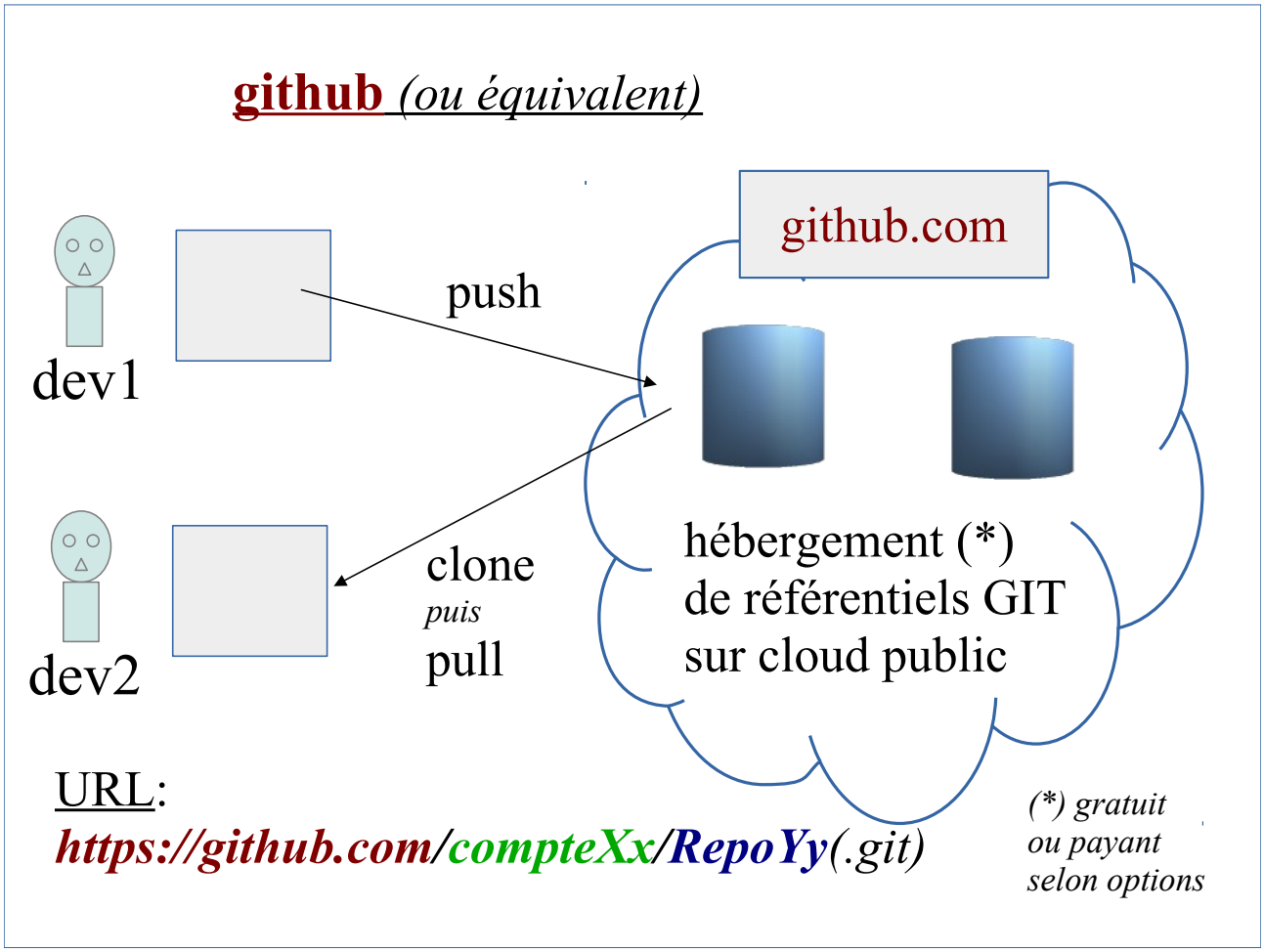
Mode distribué de GIT

Dans un système « scm » centralisé (tel que CVS ou SVN) , le référentiel central comporte toutes les versions des fichiers et chaque développeur n'a (en général) sur son poste que les dernières versions des fichiers.

Dans un **système « scm » distribué** (tel que **GIT** ou **Mercurial**) , **le référentiel central ne sert que pour échanger les modifications et chaque développeur a (potentiellement) sur son poste toutes les versions des fichiers.**


dev1
projet Yy sous contrôle GIT
dernière version
Repo. local .git (toutes versions)
Serveur (ssh (ou http ou …) pour accès distants
Référentiel Partagé repoXY.GIT en mode « bare » [toutes versions de Yy]
dev2
projet Yy sous contrôle GIT(dernière version dev)
commit
checkout
fetch ou pull
push
Repo. local .git (toutes versions)

2.2. Hébergement de référentiel GIT



Principaux sites d'hébergement de référentiels GIT :

- **github**
- **gitlab**
- **bitbucket**
- ....

2.3. Installation et configuration minimum :

1) **installer** "***Git for windows***" ou bien "***Git sur linux***" (via **yum** ou **apt-get** ou autre) .
2) **git --help**
3) **git config ....**

*En bref, les commandes «**commit** » et «**checkout**» de **GIT** permettent de gérer le référentiel **local** (propre à un certain développeur) et les commandes « **push** » et « **fetch / pull** » de **GIT** permettent d'effectuer des **synchronisations** avec le **référentiel partagé distant** .*

Configuration locale de GIT

Installation de GIT sous linux (debian/ubuntu) :

```
sudo apt-get install git-core
```

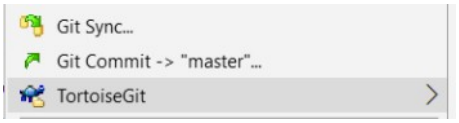
Configuration locale:

```
git config --global  user.name  "Nom Prénom"
git config --global  user.email  "poweruser@ici_ou_la.fr"
#...
```

```
# pour voir ce qui est configuré :
git config --list
```

2.4. GIT , Tortoise-GIT et intégration IDE

Utilisation directe ou indirecte de GIT



Ajoute des menus contextuels à l'explorateur de fichiers

Menu "team" et perspective "GIT"

tortoiseGit

...

plugin GIT pour **IDE** **eclipse** ou ...

GIT (en ligne de commande)

O.S. (linux ou windows ou ...)

II - Bases locales de GIT

1. Principales commandes de GIT (en mode local)

Commandes GIT (locales)	Utilités
git init	Initialise un référentiel local git (sous répertoire caché « .git ») au sein d'un projet neuf/originel.
git clone *url_referentiel_git*	Récupère une copie locale (sous le contrôle de GIT et avec toutes les versions des fichiers) d'un référentiel git existant (souvent distant)
git status **git diff** *fichier*	Affiche la liste des fichiers avec des changements (pas encore enregistrés par un commit) et git diff affiche les détails (lignes en + ou -) dans un certain fichier.
git add *liste_de_fichiers*	Ajoute un répertoire ou un fichier dans la liste des éléments qui seront pris en charge par git (lors du prochain commit).
git commit *-m message [-a]*	Enregistre les derniers fichiers modifiés ou ajoutés dans le référentiel git local (ceux préalablement précisés par *add* et affichés par *status*) . si option -a tous les fichiers modifiés (ou supprimés) qui étaient déjà pris en charge par git seront enregistrés
git checkout *idCommit (ou branche)*	Récupère les (dernières ou ....) versions depuis le référentiel local
git **--help** git *cmde* **--help**	Obtention d'une aide (liste des commandes ou bien aide précise sur une commande)
git **log --stat** *ou git log -p*	Affiche l'historique des mises à jour -p : avec détails , --stat : résumé
git branch , **git checkout** *nomBranche*, **git merge**	Travailler (localement et …) sur des branches
git grep texte_a_rechercher	Recherche la liste des fichiers contenant un texte
git tag *NomTag IdCommit*	Associer un **tag** parlant(ex: ***v1.3***) à un id de commit .
git tag -l	Visualiser la liste des tags existants
git checkout tags/*NomTag*	Récupère la version identifiée par un tag

Exemples :

#initialisation

```
cd p1; git init
```

#affichage des éléments non enregistrés

```
cd p1; git status
```

→ *affiche:*

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#         modified:   src/f1.txt
#         modified:   src/f3-renamed.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

commit all already tracked/added :

```
cd p1
#  -a  pour tous les fichiers listés dans git status
git commit -a  -m "my commit message"
```

#commit all (with all new and deleted) :

```
cd p1
git add pom.xml.txt   src/*
git status
# git commit gère tous les fichiers ajoutés (et supprimera de l'index ceux qui
# n'existent plus si option -a)
git commit -m "my commit message" -a
```

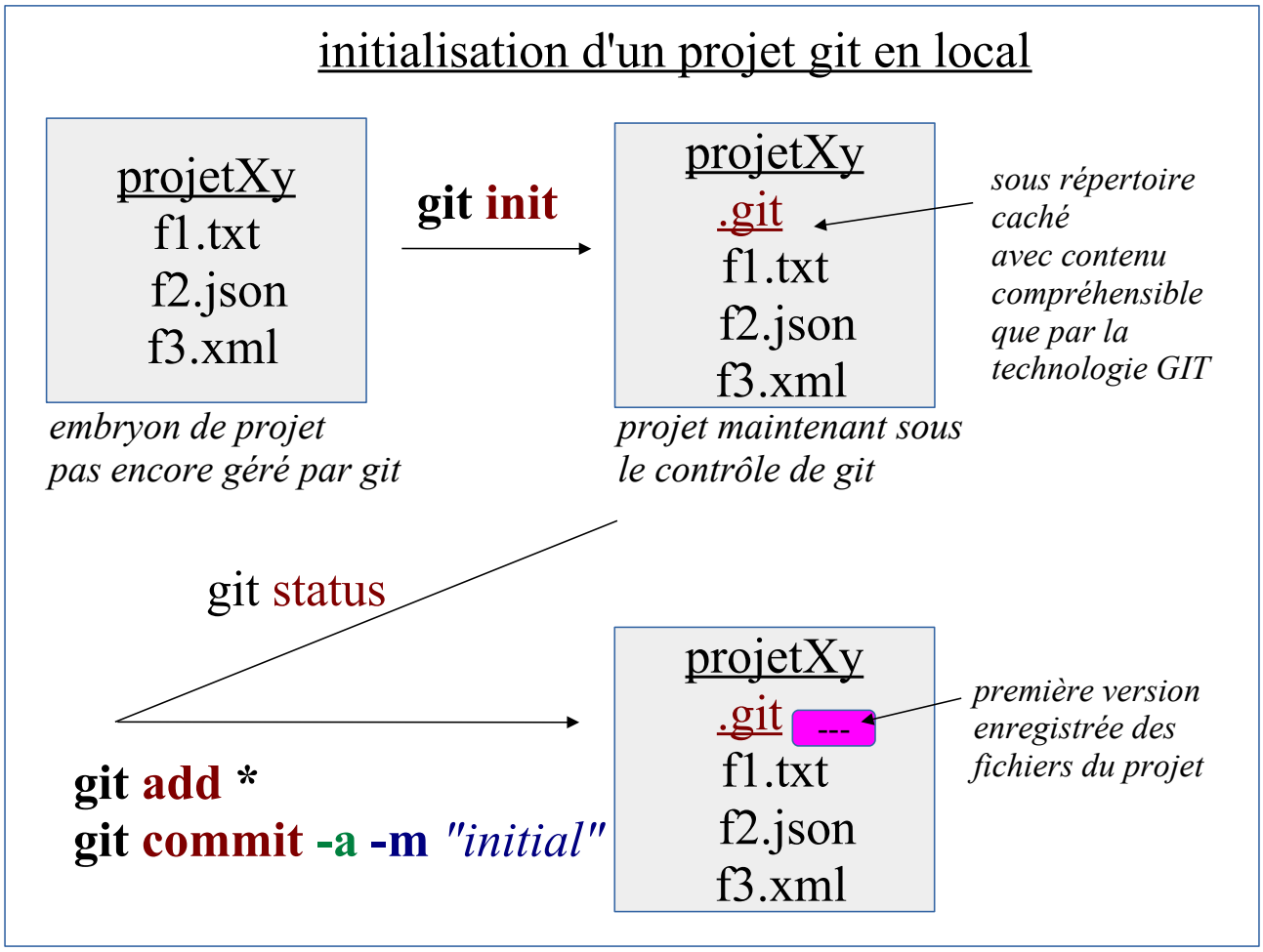
#historique des dernières mises à jour :

```
cd p1;  git log –stat
```

---> *affiche*:

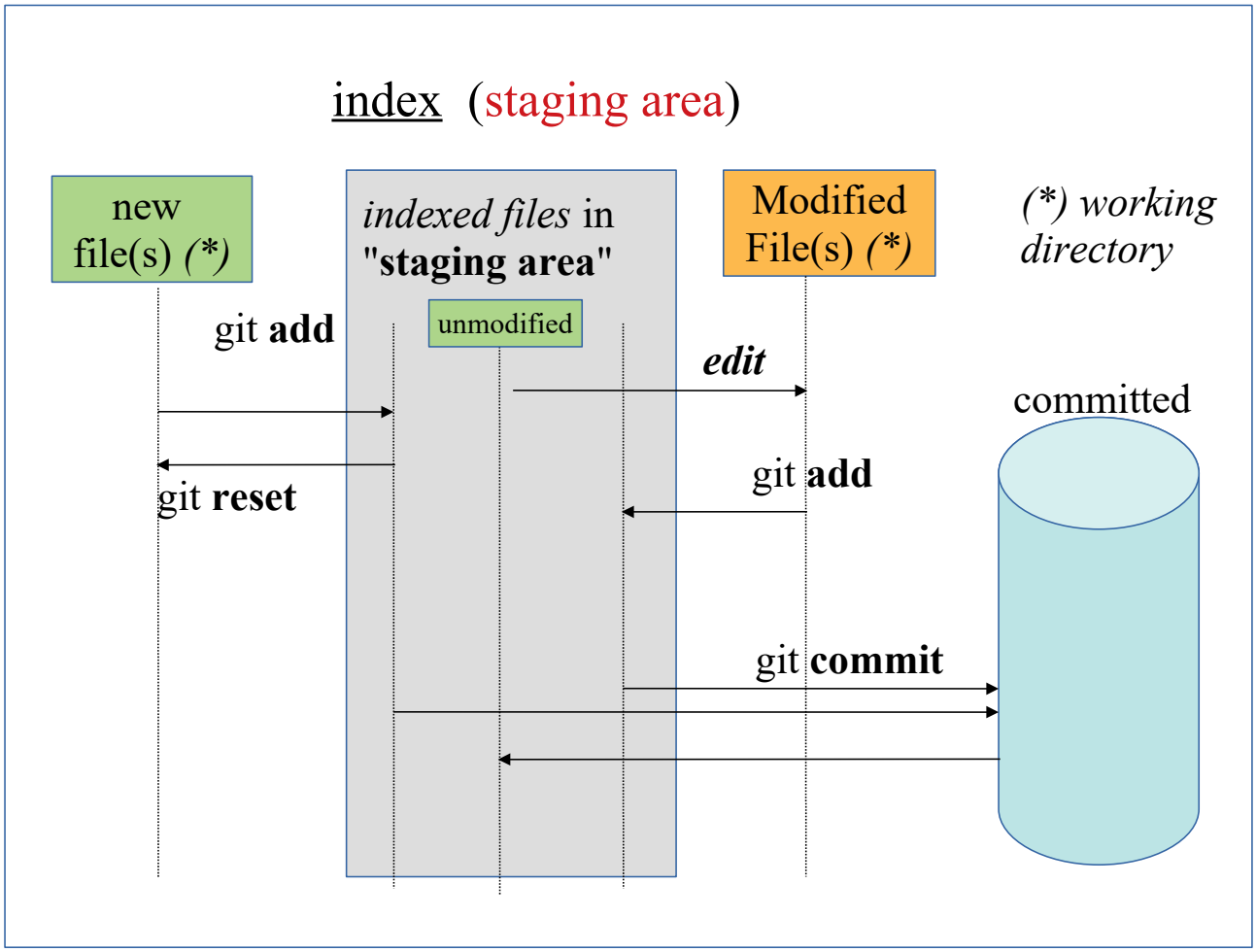
```
commit 93446a0f2194089d83c941a63768f212eb96e0f8
Author: developpeur fou <moi@ici_ou_la.everywhere>
Date:   Wed Dec 12 18:36:40 2012 +0100
    my commit message
 pom.xml.txt          |    2 ++
 src/f1.txt           |    1 +
 src/f3-renamed.txt   |    1 +
 src/f4-renamed.txt   |    1 +
 src/p/pf2-renamed.txt |    2 ++
 5 files changed, 7 insertions(+)
```

2. initialisation d'un projet GIT (en local)



3. Index (staging area)

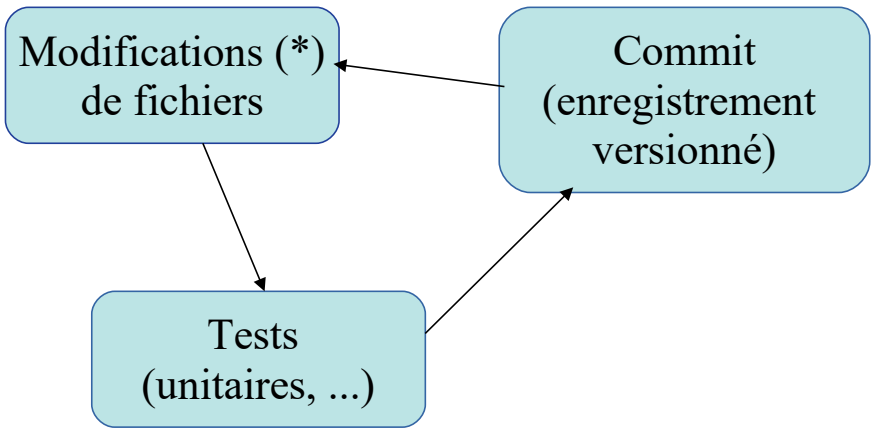
3.1. fichiers indexés par GIT



Si besoin , **git reset f1.txt** permet d'annuler **git add f1.txt**

3.2. cycle des mises à jour

Cycle de mises à jour



(*) Un fichier existant déjà lors du dernier commit et re-modifié depuis fait d'office partie de la "***staging area***" et sera par défaut re-commité . Les commandes git add … , git rm … , git rm --cached … permettent d'ajouter ou retirer des fichiers à commiter ultérieurement avec l'option -a .

git commit --amend -m *nouveauMessage* permet si besoin de remplacer le message de commit

4. indispensable .gitignore

Le fichier caché .gitignore (à placer à la racine d'un référentiel git) est indispensable pour préciser la liste des fichiers à ne pas stocker dans le référentiel git (ex : fichiers temporaires , spécifiques à un IDE , fichiers binaires générés , ...) .

Exemple de fichier ".gitignore" pour java / maven /eclipse :

.gitignore

```
target/
*.class
*/.settings/
.settings/*
.settings
*.jar
*.war
*.ear
```

Exemple de fichier ".gitignore" pour npm/javascript :

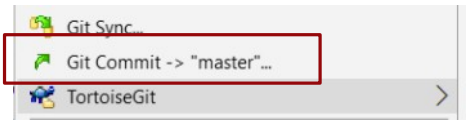
.gitignore

```
node_modules
node_modules/*
dist/*
dist
```

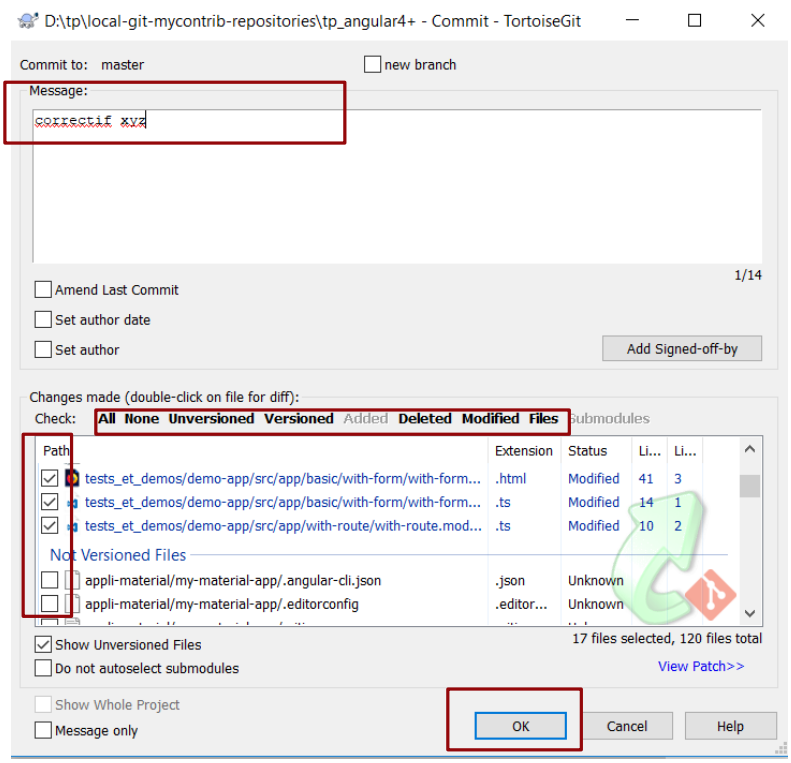
5. Commit et tags

commit via tortoiseGit

dans l'explorateur de fichiers à partir du répertoire du projet déclencher le menu contextuel **commit --> ...**



choix des fichiers à "commiter"



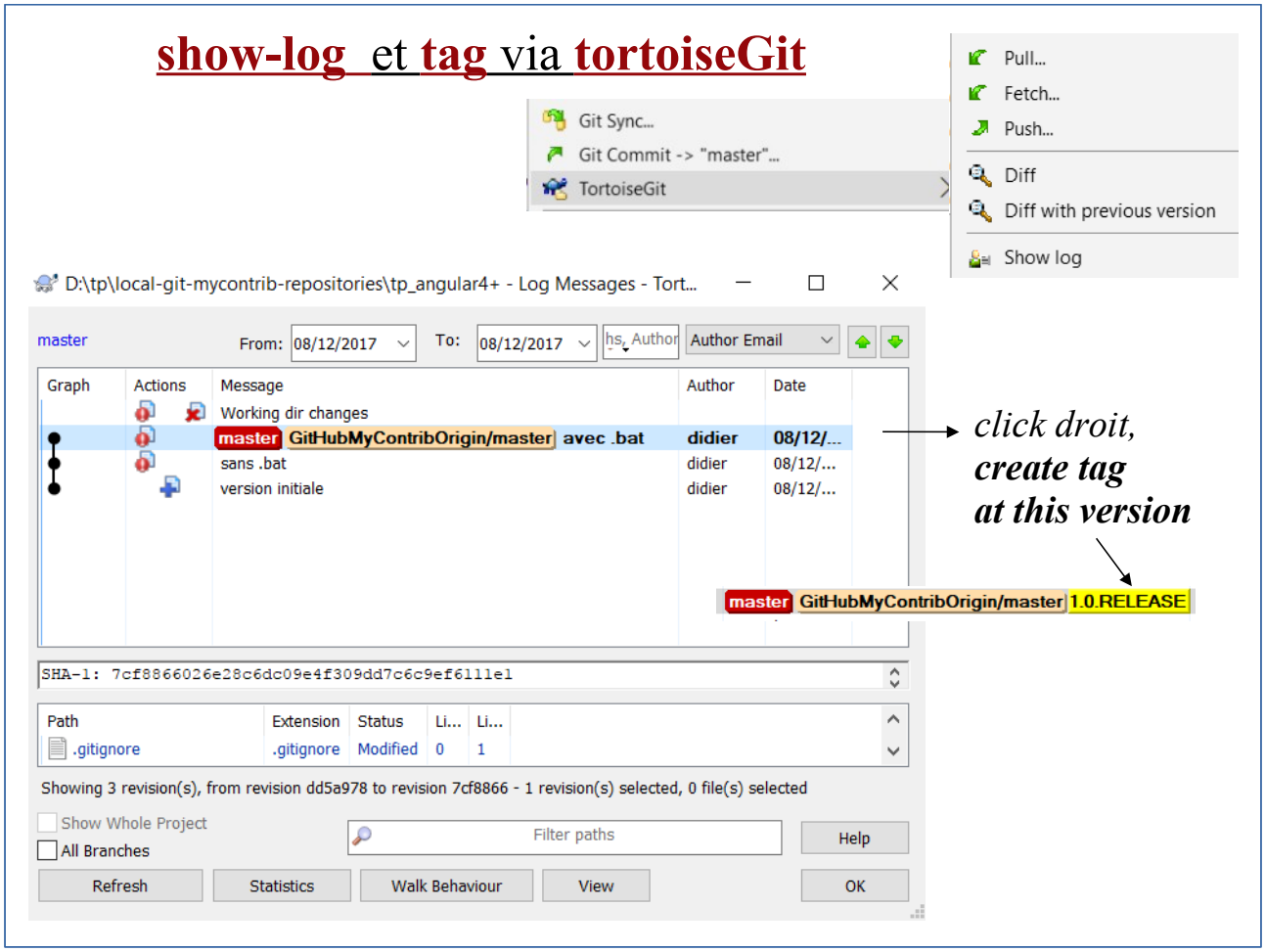
NB :

Il est éventuellement possible de visualiser l'état du projet dans une révision (numéro de commit bien précis) via la commande **checkout** *numero_de_commit_precis* .

Ceci dit , il ne vaut mieux pas éditer et enregistrer des fichiers juste après car on serait alors en mode "detached HEAD" (détaché d'une tête de branche) ce qui pose plein de problème par la suite .

---> pour effectuer des modifications à un niveau antérieur de l'historique , il faudra idéalement créer une nouvelle branche démarrant au niveau d'un commit_bien_précis .

show-log et tag via tortoiseGit

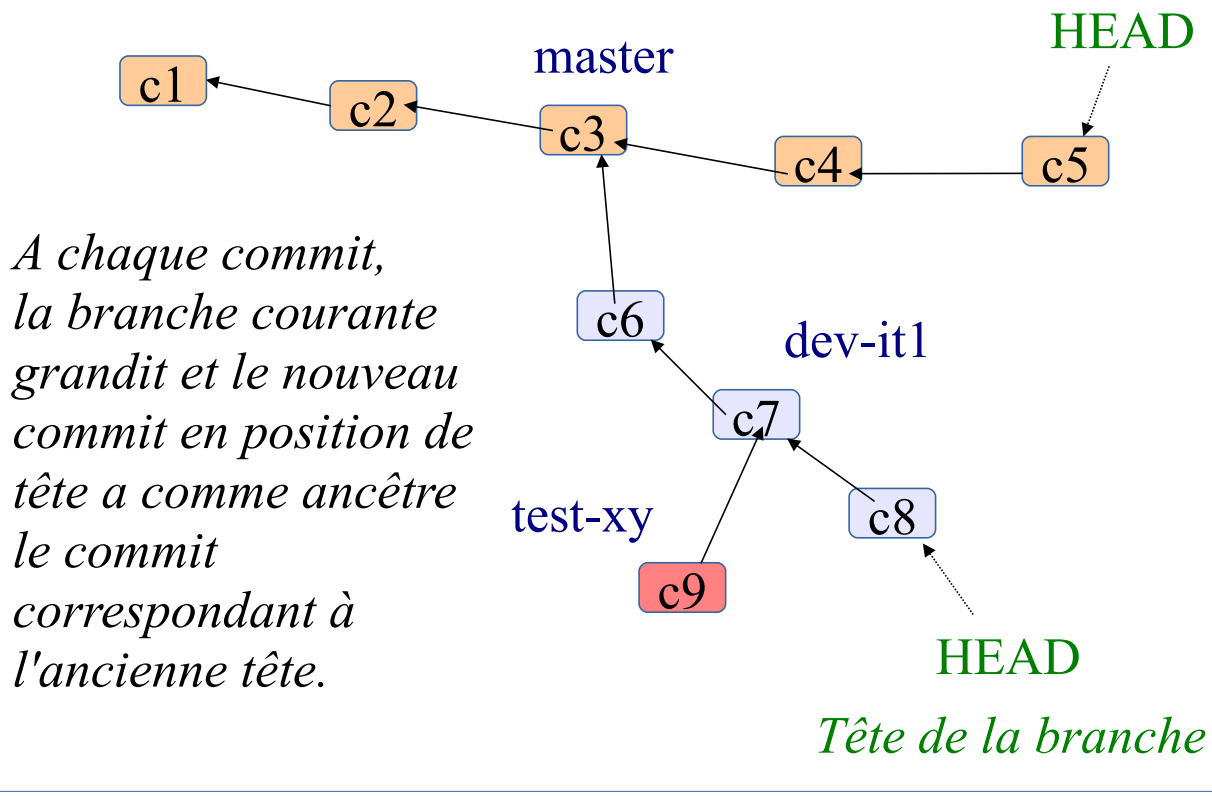


....

III - Gestion des branches / GIT

1. Gestion des branches avec GIT

Structures des branches de GIT



NB : au sein des anciennes versions de git, la baranche principale s'appelait souvent **master** .
au sein des versions récentes de git, la baranche principale s'appelle souvent **main** .

Le choix du nom de la branche principale est contrôlable via l'option suivante :

```
git config --global init.defaultbranch master
git config –list
git config --global init.defaultbranch main
git config --list
```

Annulation d'un commit erroné

En local, la commande **git reset --hard *idCommit*** permet de repositionner la tête de la branche courante sur un ancien commit et toute les modifications apportées par le(s) tout/s dernier(s) commit(s) seront effacées/perdues .
Si l'option --soft est utilisée à la place de --hard, les fichiers modifiés ne sont pas effacés du répertoire de travail et on peut les modifier avant d'effectuer un nouveau commit.

La syntaxe spéciale **HEAD~1** correspond à **l'avant dernier commit** et donc **git reset --hard HEAD~1** annule le dernier commit.

HEAD^ ou HEAD~1 : avant dernier commit
HEAD^^ ou HEAD~2 : avant avant dernier commit

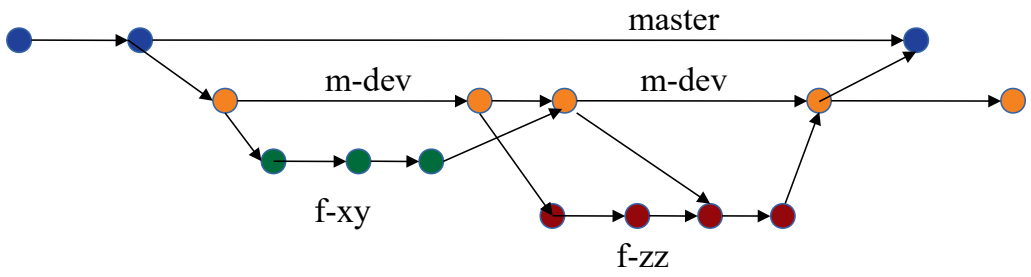
Tout projet commence avec une seule branche «**master**» .

Commandes GIT (branches)	**Utilités**
git branch	Affiche la liste des branches et précise la branche courante (*) .
git branch *nomNouvelleBranche*	Créer une nouvelle branche (qui n'est pas automatiquement la courante)
git checkout *nomBrancheExistante*	Changement de branche (avec mise à jour « checkout » des fichiers pour refléter le changement de branche) .
git checkout *master* **git merge** *autreBranche_a_fusionner*	Modifie la branche courante (ici «master») en fusionnant le contenu d'une autre branche
git branch -d *ancienneBrancheAsupprimer*	Supprime une ancienne branche (avec -d : vérification préalable fusion, avec -D : pas de verif , pour forcer la perte d'une branche morte)
...	

2. Bonnes pratiques dans la gestion des branches

Bonnes pratiques élémentaires (branches GIT)

- ne pas directement programmer sur la branche "master" (à considérer comme la branche stable à déployer en production) mais sur une branche de développement .
- Un certain développeur peut éventuellement créer une sous branche d'ajout de fonctionnalité purement locale (sans push) pour expérimenter et tester une extension, effectuer un merge local et ultérieurement effectuer un push sur la branche parente/commune de développement .
- Attention à bien organiser (en équipe) la gestion des branches distantes (pour éviter pagaille et gros problèmes)

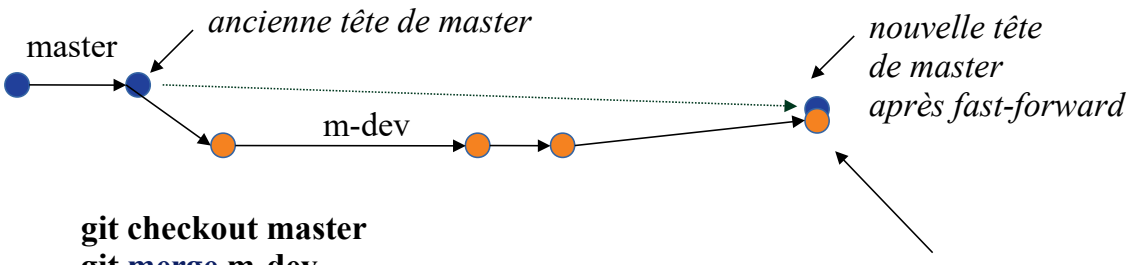


3. Merge

3.1. merge rapide (fast-forward)

Fast-forward (simple merge)

- Lorsque la branche à fusionner comporte quelques commits et que la branche actuelle (réceptrice de la fusion) ne comporte aucun autre commit depuis l'origine de la branche à fusionner , GIT peut effectuer un merge extrêmement rapide et simplifié appelé "fast-forward" .
- Ce "fast-forward" consiste à actualiser la référence de la tête de la branche réceptrice pour quelle coïncide avec la tête de la branche à fusionner .



git checkout master
git merge m-dev
Updating 07b692d..a465c43
Fast-forward
f1.txt | 4 ++--
f3.txt | 3 ++-
2 files changed, 4 insertions(+), 3 deletions(-)

les têtes des branches master et m-dev référencent le même dernier commit

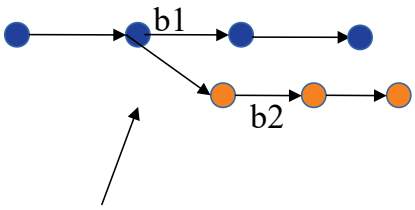
Mis à part ce cas très particulier , les autres sortes de "merge" conduiront à la création d'un nouveau commit de fusion (avec 2 ancêtres).

3.2. merge sans conflits

Merge automatique (sans conflit)

- Lorsque les 2 branches à fusionner comportent quelques commits associés à des modifications/ajouts/suppressions de fichiers différents , GIT peut alors effectuer un merge automatique .
- La branche courante avance et sa nouvelle tête référence un nouveau commit de fusion (avec 2 ancêtres).

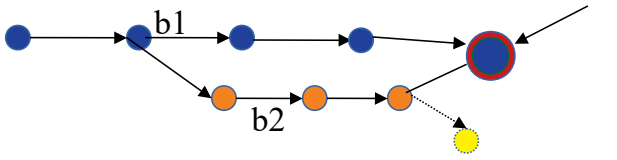
avant:



Les modifications "commitées" sur les branches b1 et b2 à fusionner portent sur des éléments (fichiers, ...) distincts .

après:

nouvelle tête de b1 positionné sur nouveau commit



git checkout b1
git merge b2
git log
....
Merge branch b2 into b1
....

futur avancement de b2

NB : Git est capable de gérer automatiquement un merge dans le cadre de modifications (de type "remplacements") apportées sur des lignes différentes d'un même fichier . Ceci n'étant qu'une heuristique , il convient de relancer certains tests unitaires (ou de relire certains fichiers) pour s'assurer que la fusion automatique de git a bien fonctionné .

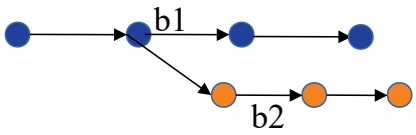
3.3. merge avec conflits

Merge avec conflit à résoudre

- Lorsque les 2 branches à fusionner comportent toutes les 2 quelques commits associés à des modifications/ajouts/suppressions qui portent sur des fichiers communs , GIT ne peut alors pas décider de ce qu'il faut garder et les conflits doivent être résolus de manière interactive .

- via "git status" on peut connaître la liste des fichiers en conflit lors du merge

- soit le merge doit être annulé via "***git merge --abort***"
  soit ***les conflits doivent être résolus*** (éditions, add , commit)
  pour que le "merge" puisse aboutir .



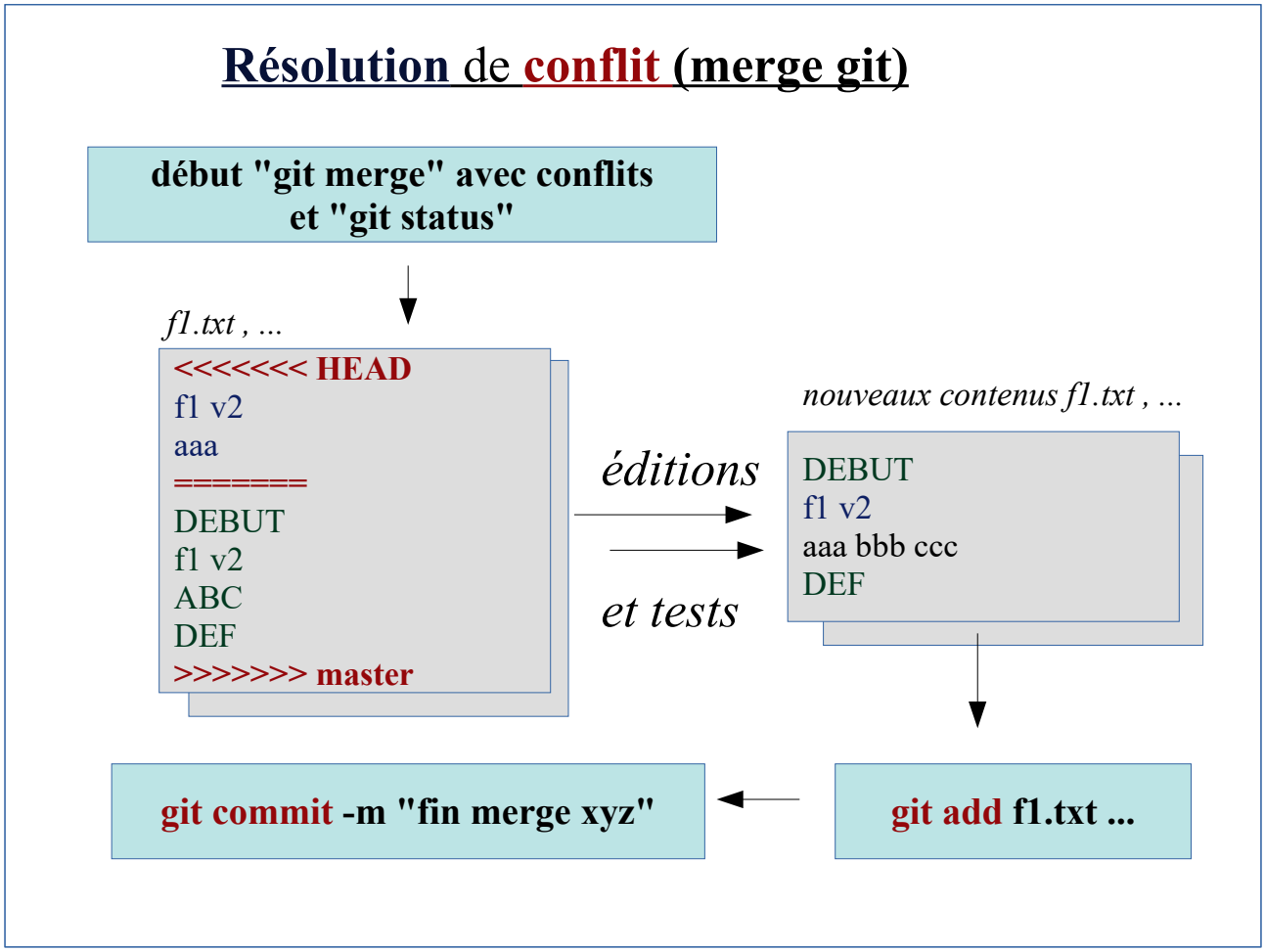
Les modifications "commitées" sur les branches b1 et b2 à fusionner portent sur quelques éléments (fichiers, ...) communs .

git status (après merge)
On branch m-dev
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified: f1.txt

no changes added to commit (use "git add" and/or "git commit -a")



git add f1.txt

>**git status**
On branch m-dev
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:

modified: f1.txt

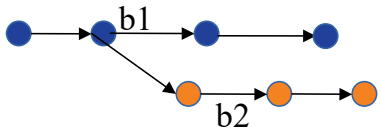
git commit -m "fin merge"

4. Git rebase (souvent sur branche privée)

git rebase

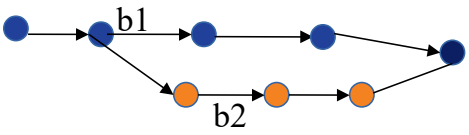
- La commande "git rebase"permet de reprendre toutes les modifications qui ont été validées sur une branche et de les rejouer sur une autre branche
- Ceci permet d'obtenir un historique linéaire (sans le nouveau commit de fusion d'un merge ordinaire)

avant merge:

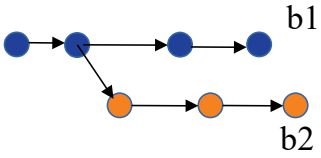


git checkout b1
git merge b2

après merge:

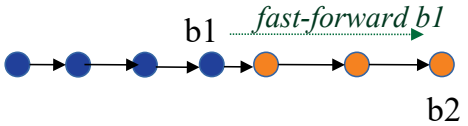


avant rebase:



git checkout b2
git rebase b1
git checkout b1
git merge b2
(fast-forward après rebase)

après rebase:



Si lors du "rebase" certains conflits sont détectés , l'opération de "rebase" est alors temporairement stoppée. On peut alors :
- soit annuler le "rebase" via "**git rebase –abort**"
- soit résoudre le conflit de la même manière que pour un merge (édition, add ...)
puis déclencher "**git rebase --continue**"

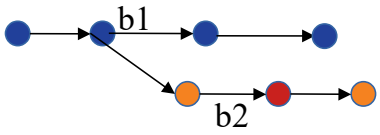
Un "rebase" est très déconseillé sur une branche publique car les réorganisations locales pourront une fois propagées mettre la pagaille chez les autres développeurs d'une équipe .

5. Git cherry-pick

git cherry-pick *(pour cas exceptionnel)*

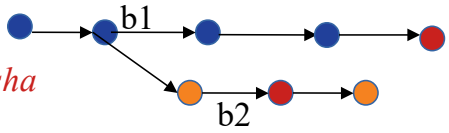
La commande "git **cherry-pick**"permet de récupérer des modifications d'un commit spécifique (d'une autre banche ou commit perdu/détaché) de manière à l'intégrer dans la branche courante .
Cette opération exceptionnelle peut s'avérer pratique pour réintégrer un "hotfix" (correction de bug en urgence) ou pour corriger certains problèmes.

avant cherry-pick:

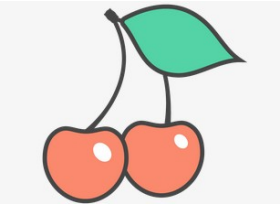


git checkout b1
git cherry-pick refCsha

après cherry-pick:



refCsha est ici une référence sha sur le commit rouge de la branche b2



IV - GIT en mode distant (--bare , github, ...)

1. Commandes de GIT pour le mode distant

Commandes GIT (mode distant)	Utilités
git init --bare	Initialisation d'un nouveau référentiel vide de type «nu» ou «serveur». (à alimenter par un push depuis un projet originel)
git clone --bare ***url_repo_existant***	Idem mais via un clonage d'un référentiel existant
git clone ***url_repo_sur_serveur.git***	Création d'une copie du projet sur un poste de développement (c'est à ce moment qu'est mémorisée l'url du référentiel « serveur » pour les futurs push et pull)
git pull	Rapatrie les dernières mises à jour du serveur distant (de référence) vers le référentiel local. (NB: *git pull* revient à déclencher les deux sous commandes *git fetch* et *git merge*)
git push	Envoie les dernières mises à jour vers le serveur distant (de référence) Attention: ***le push est irréversible*** et ***personne ne doit avoir effectuer un push depuis votre dernier pull !***
...	

Exemples:

#script de création d'un nouveau référentiel GIT (coté serveur) dans /var/scm/git ou ailleurs:

```
mkdir p0.git
cd p0.git
git init --bare
git update-server-info
mv hooks/post-update.sample hooks/post-update

#nb www-data est le groupe de apache2
cd ..
sudo chgrp -R www-data p0.git

# ce repository initial et vide pourra être alimenté par un push depuis un projet "original"
# depuis ce projet original , on pourra lancer git config remote.p0.url http://localhost/git/p0.git
#                          puis   git push p0 master
echo "fin ?"; read fin
```

ou bien

```
# construira p1.git
```

```
git clone --bare  file:///home/formation/Bureau/tp/tmp-test-git/original/p1
cd p1.git
git update-server-info
```

#récupération d'une copie du projet sur un poste de développement

```
git clone http://localhost/git/p1.git
```

#pull from serv:

```
cd p1
git pull
```

#push to serv:

```
cd p1
git push
```

2. Gérer plusieurs référentiels distants

Commandes GIT (mode distant)	Utilités
git remote -v	Affiche la liste des origines distantes (URL des référentiels distant
git remote add originXy url_repoXy	ajoute une origine distante (alias associé à URL)
git push -u originXy	Effectue un push vers l'origine (upstream) précisé (alias associé à l'URL du référentiel distant).
git push **--set-upstream originXy master**	push en précisant la branche remote à pister (track) ceci est particulièrement utile pour un push initial vers référentiel distant vide (pas encore initialisé)
...	

Exemples :

s_list_remote_git_url.bat

```
git remote -v
pause
```

s_set_git_remote_origin.bat

```
git remote set-url origin Z:\TP\tp_angular1.git
git remote -v
```

```
pause
```

s_push_to_remote_origin.bat

```
git push -u origin master
pause
```

s_push_to_gitHub.bat

```
git remote add GitHubMyContribOrigin https://github.com/didier-mycontrib/tp_angular.git
REM didier-mycontrib / gh14.....sm..x / didier@d-defrance.fr
git push -u GitHubMyContribOrigin master
pause
```

commit_and_push.bat

```
cd /d "%~dp0"
git add *
git commit -a -m "nouvelle version"
git push -u GitHubMyContribOrigin master
pause
```

Bien qu'il soit possible de placer username et password dans l'URL ceci est une mauvaise pratique d'un point de vue sécurité .

Il vaut mieux configurer GIT pour qu'il retienne le mot de passe saisi durant une certaine période (exemple : 3600 secondes = 1 heure) :

```
git config [ --global ]   credential.helper   'cache --timeout=3600'
```

Et si nécessaire (pour ancienne version de git) :

```
~~git credential-cache exit~~
```

pour que GIT oublie l'ancien mot de passe et que l'on puisse de ré-authentifier .

ou pour version récente de git :

```
git config --global --unset credential.helper
et/ou
git config --system --unset credential.helper
```

pour désactiver (temporairement ou pas) le "credential.helper" mémorisant les username/password .

Sur une machine windows d'entreprise, on peut également choisir "manager" comme type de "credential.helper" via la commande :

```
git config --global credential.helper manager ou wincred
```

puis en lançant la commande suivante pour vérifier :

```
git config --system --list
```

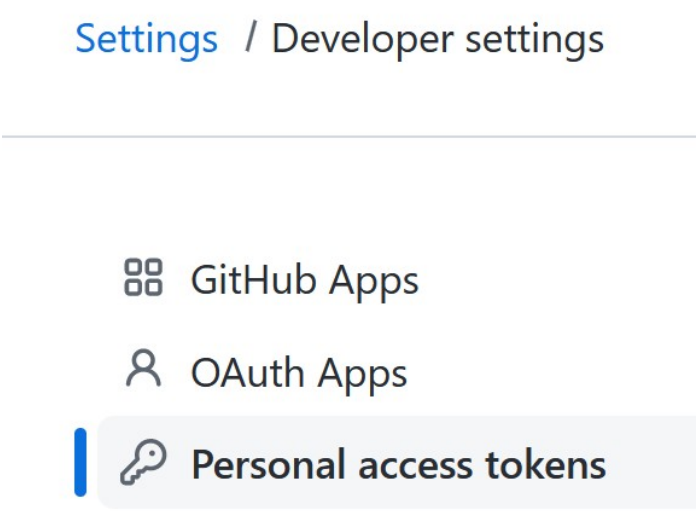
Ceci permet de configurer les informations d'authentification via la partie "*Comptes d'utilisateurs*

→ *Gestionnaire des informations d'identification* → *Gérer les informations d'identification Windows* " du panneau de configuration de windows d'entreprise (pas "windows famille") .

NB: l'exemple configuration ci dessus doit être adaptée au cas par cas (selon version de git , selon OS linux ou windows , selon github ou gitlab, selon l'année , ….)

Token pour opérations git distantes sans mot de passe

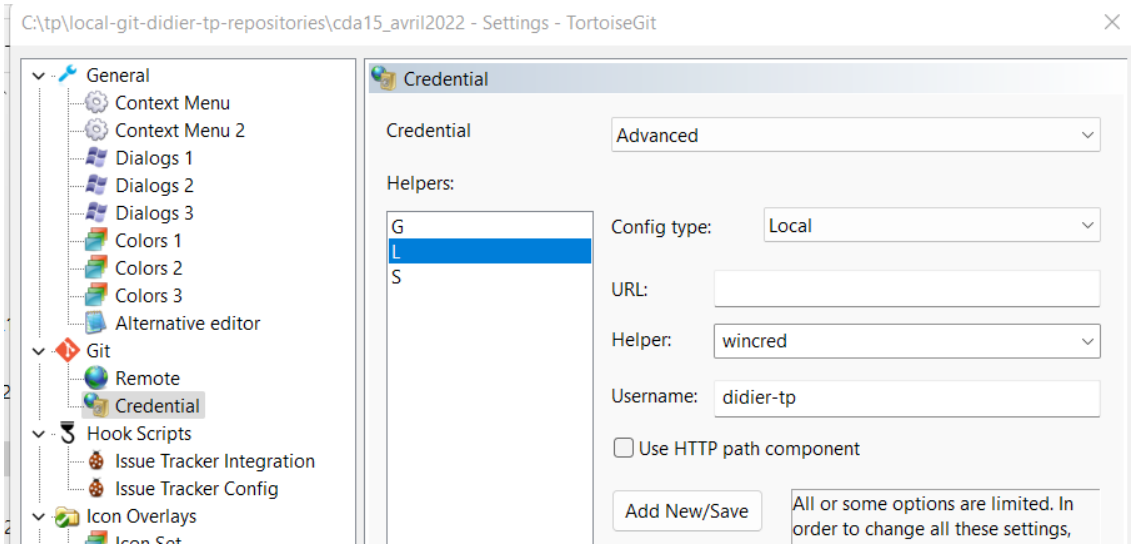
De manière à accéder en lecture/écriture à un référentiel distant géré de façon récente par GitHub ou bien GitLab on a besoin de générer et enregistrer un "**token développeur**" (via la page d'administration d'un compte GitHub ou GitLab) .



Une fois généré et mémorisé dans un fichier temporaire , ce "token développeur" (ressemblant à *gtNVm7wH8P9ohp_u5jIVxzs2AWx6zT2E5p4AYlsF* et valable pour une durée à choisir telle que 30 jours par exemple) devra être enregistré auprès d'un "***credential.helper***" du poste du développeur (ex : linux ou windows) et sera utilisé en tant qu'élément d'authentification .

Cette opération un peut technique peut éventuellement être effectuée graphiquement via une interface graphique telle que tortoise git ou autre .

Avec tortoise git (git settings , partie git/credentials ,
add **local** config with **credential-helper=wincred** and ***username***)



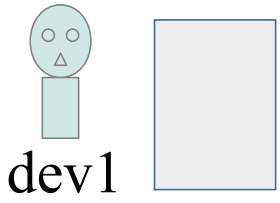
et spécifier le token via un copier/coller lorsque ce sera demandé (lors d'un git push par exemple).
Ce token (lié à votre username) sera ensuite automatiquement conservé lors des futures requêtes .

3. initialiser git en mode distant

initialiser git en mode "remote"

1 ***préparation d'un projet***
git en mode local *(java ou ...)*
git init , .gitignore , ... , commit

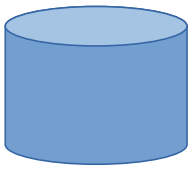
2 ***préparation du référentiel git distant vide***
(git init --bare)
ou équivalent via console github ou autre

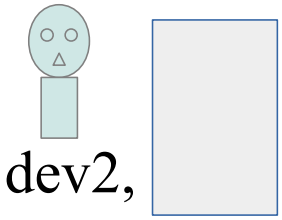


3 ***préciser url et premier push*** :

git remote add origin ***url_repo***

git push --set-upstream origin master





4 ***clonage(s)*** *(avec "origin" fixée automatiquement)* :

git clone ***url_repo***

Ensuite , ***git pull*** *et* ***git push*** *de chaque coté .*

4. pistage (track) entre branches locales et distantes

pistage (tracking) de branche

- Lorsqu'une branche locale est paramétrée pour pister (track) une branche distante , celles ci pourront ensuite être synchronisées via git push et git pull (et autres).

- Pour initialiser ce processus, il faut la première fois lancer la commande **git push** avec l'option **--set-upstream** origin

- Exemple : **git push --set-upstream origin m-dev**

[new branch] m-dev -> m-dev
Branch m-dev set up to track remote branch m-dev from origin.
git branch -vv
** m-dev 9f0d962 [origin/m-dev] ...*
master d41dc86 [origin/master] ...

pistage (tracking) de branche (sur clones)

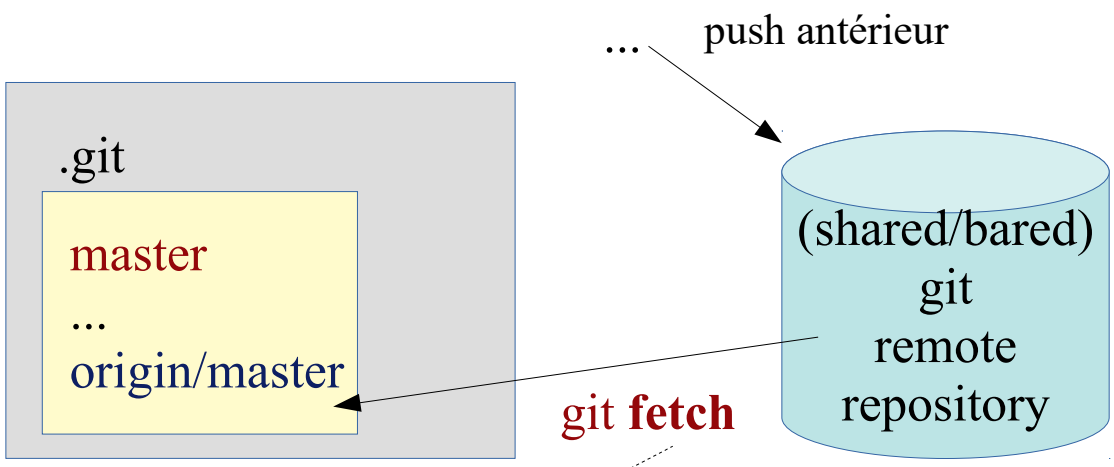
- **git branch -a** permet de visualiser toutes les branches dont les branches distantes existantes :
  ** master*
  remotes/origin/HEAD -> origin/master
  remotes/origin/m-dev
  remotes/origin/master

- Au niveau d'un clone secondaire , pour initialiser une branche locale pistant une branche distante existante on peut lancer la commande suivante :
  git checkout -b m-dev origin/m-dev
  Branch m-dev set up to track remote branch m-dev from origin.
  Switched to a new branch 'm-dev'

  git branch -vv
  ** m-dev 439df8e [origin/m-dev] ...*
  master d41dc86 [origin/master] ...

5. Git fetch et git pull

git fetch



2 on peut ensuite , en étant placé sur la branche locale master voir les différences via ***git branch -vv***
et via
git diff origin/master

1 rapatrie (réplique) les mises à jour du référentiel distant dans la branche ***origin/master*** du référentiel local.
Rien ne change sur la branche locale "master" ni dans le répertoire de travail (du projet).

git pull

- En étant placé sur la branche master pistant la branche distante origin/master,
  git **pull** est équivalent à git **fetch**
  suivit de git **merge** origin/master

- il faut quelquefois résoudre certains conflits

- Les mises à jours distantes sont alors vues / répercutées dans le répertoire de travail (du projet)

NB: Dans certains cas , on peut avoir envie de mettre de coté rapidement certaines opérations temproraires et expérimentales sur la branche courante pour y revenir après sachant qu'une opération plus importante est nécessaire (ex : rapatrier des modifications distantes via ***git pull)*** .
On peut éventuellement utiliser la commande **git stash** pour cela .
NB: après un "git stash" , les modifications temporaires (non commitées) sont stockées dans un espace local de type "brouillon temporaire réactivable" et ne seront jamais propoagées lors d'un git push .
Si l'on souhaite réactiver les modifs temporaires enregistrée via git stash , on peut lancer la commande **git stash pop**
L'option **-a** existe sur ***git stash*** de la même façon que sur git commit .

V - Git-flow et aspects divers

1. Pièges de GIT

1.1. Passwords oubliés (ou anciens passwords mémorisés)

```
git credential-cache exit
```

pour que GIT oublie l'ancien mot de passe et que l'on puisse de ré-authentifier .

1.2. Mauvaise pratique : password en clair dans URL

NB : Lorsqu'une authentification est nécessaire , celle ci peut éventuellement ***ET DANGEREUSEMENT*** *être placée dans L'URL du référentiel git distant :*

https://username:password@github.com/xxx/yyy/zzz.git

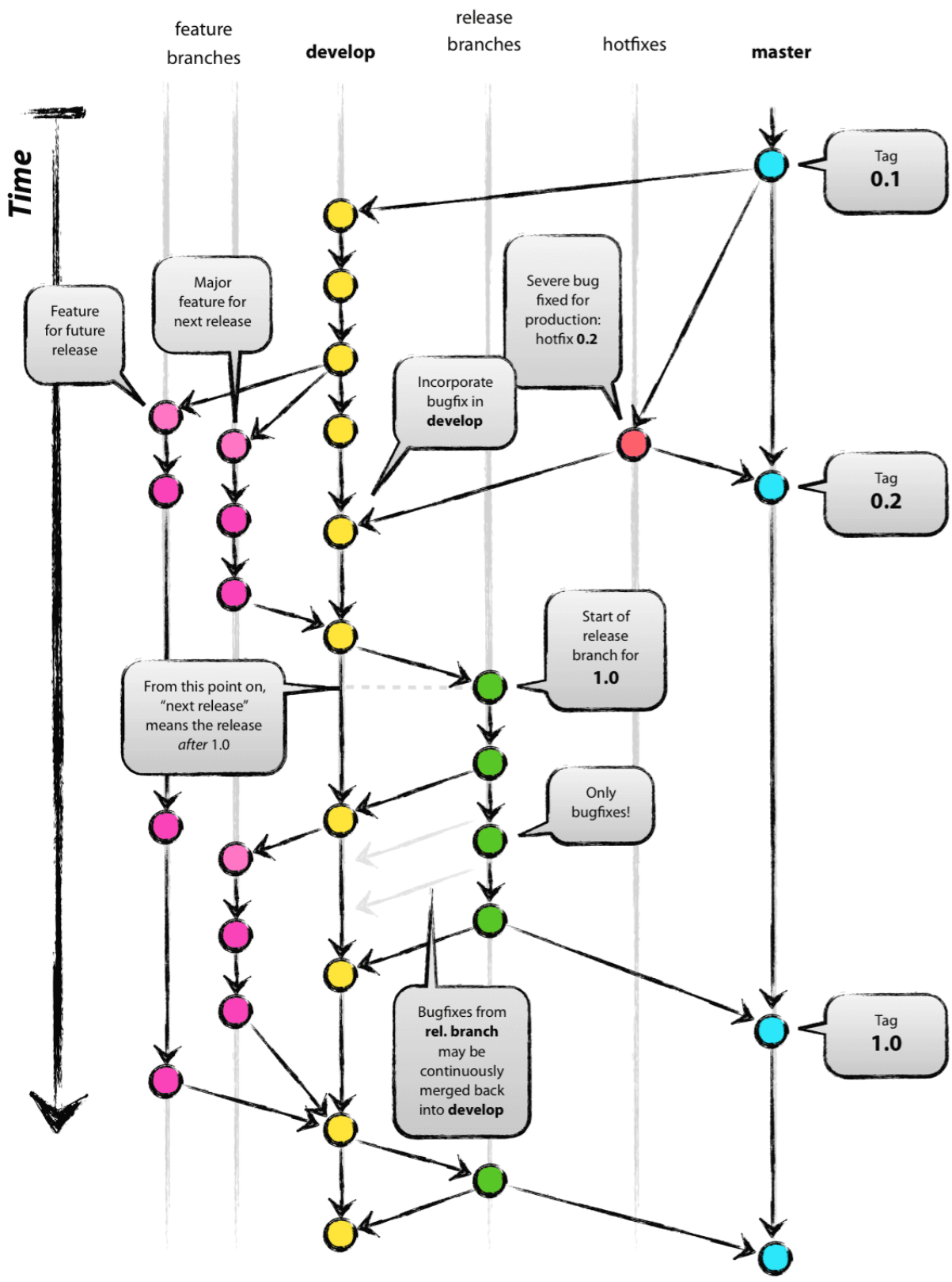
Exemple: init_push.bat

```
git init
cd /d "%~dp0"
git add *
git commit -a -m "version initiale"
REM password doit être remplacé par la valeur du password !
git remote add gitHubOriginLilleSql https://didier-tp:password@github.com/didier-tp/lille_sql.git
git push -u gitHubOriginLilleSql master
pause
```

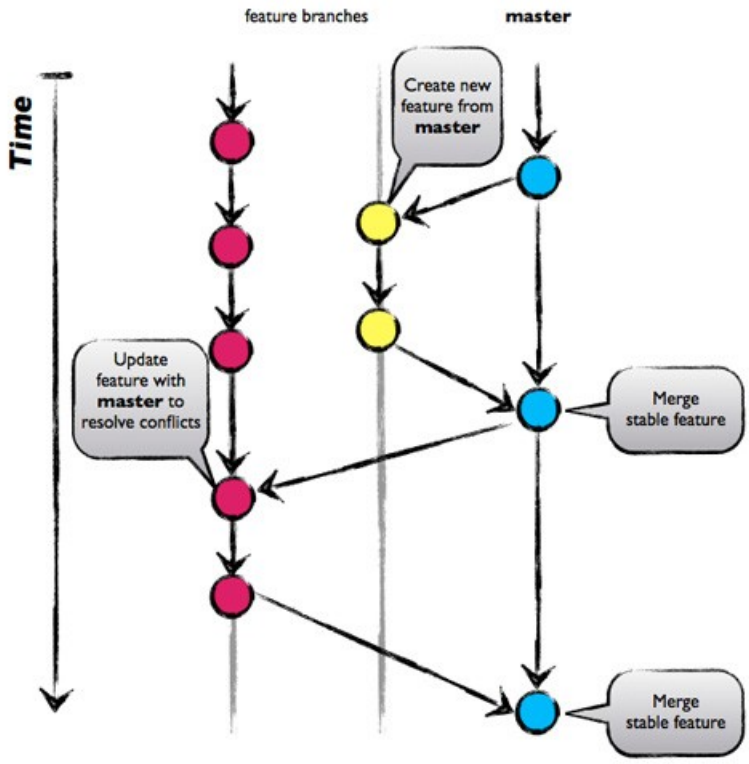
Attention à bien régler .gitignore pour ne pas placer le mot de passe dans le référentiel distant !!! (autre inconvénient : password visible dans URL via git remote -v) .

2. Quelques workflows pour GIT

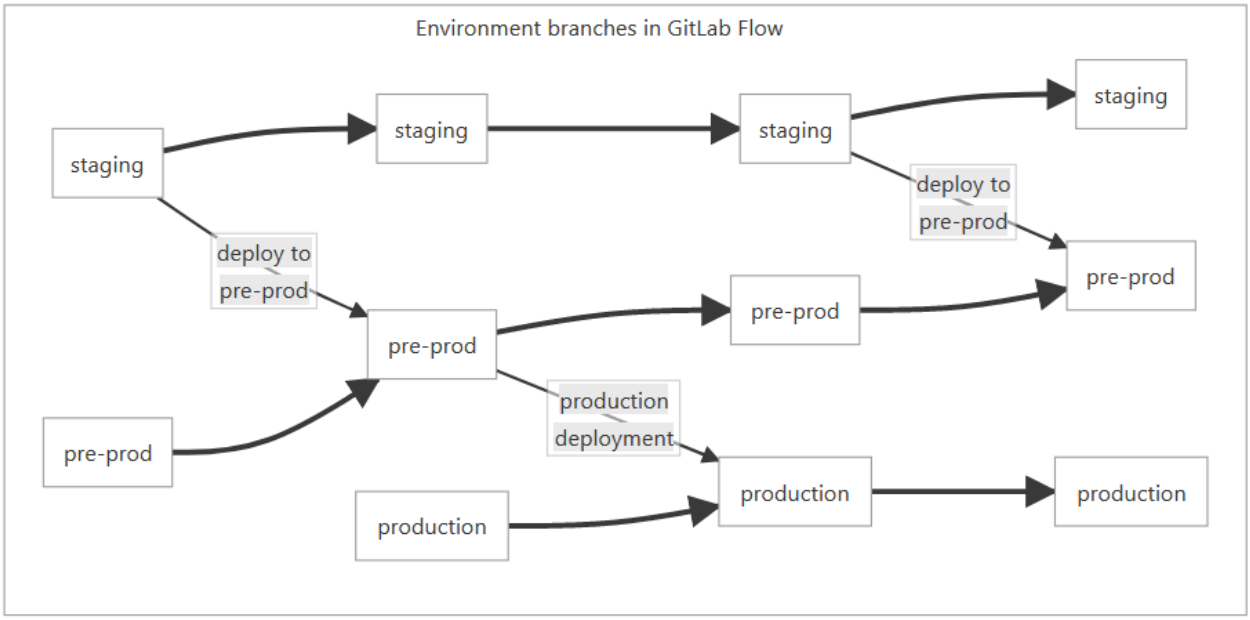
2.1. git-flow (évolué)

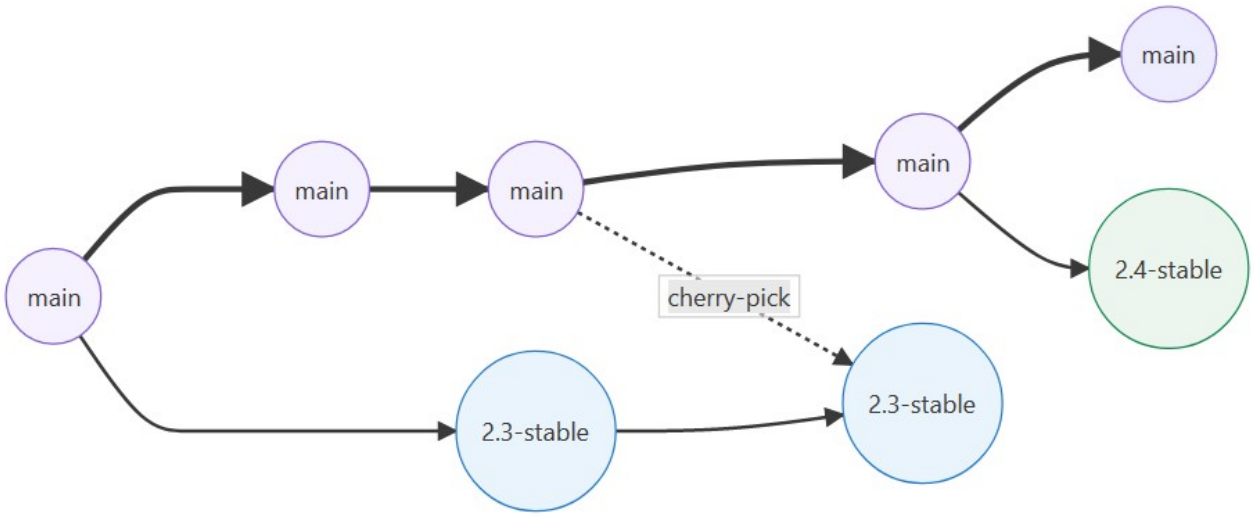


2.2. github-flow (simple)


feature branches
master
Time
Create new feature from master
Update feature with master to resolve conflicts
Merge stable feature
Merge stable feature

2.3. Gitlab flow (moins rapide , fiabilité privilégiée)





3. Git patch

GIT patch ou GIT diff est utilisé pour partager des propositions de changements sans les diffuser (via git push) sur une branche importante. De cette façon les autres personnes (ayant reçu un patch par exemple par email) pourront appliquer et tester ces changements temporaires pour les évaluer. Git patch peut éventuellement être vu comme un moyen particulier de diffuser certains correctifs.

ANNEXES

VI - Annexe – GIT avec eclipse

1. Plugin eclipse pour GIT (EGIT)

Le plugin eclipse pour GIT s'appelle **EGIT** .

1.1. Actions basiques (commit , checkout , pull , push)

→ Se laisser guider par la perspective "GIT" et via le menu "Team"

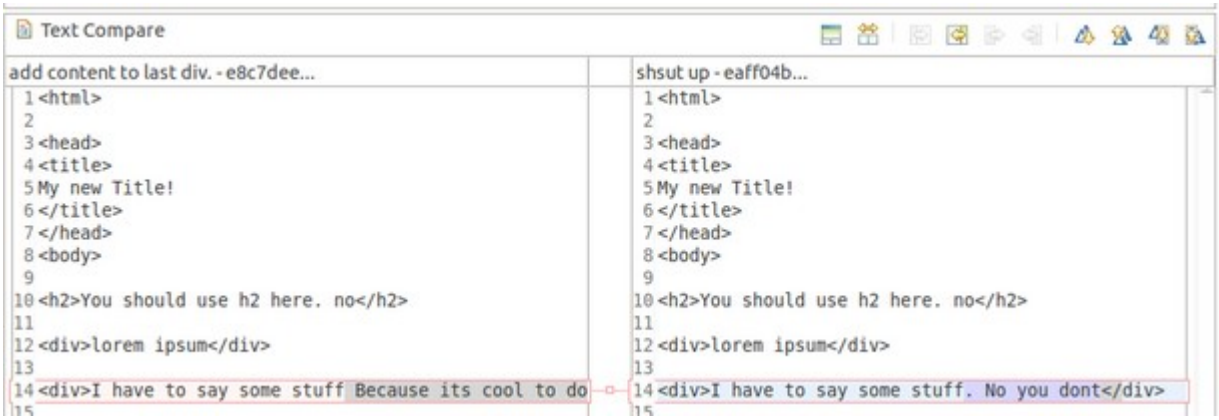
1.2. Résolution de conflits

1) déclencher "**Team / pull**" pour récupérer (en tâche de fond) la dernière version (partagée / de l'équipe). Le plugin EGIT va alors tenter un "***auto-merge***" ("git fetch FETCH_HEAD" suivi par "git merge").
2) En cas de conflit (non résoluble automatiquement) , les fichiers en conflit seront marqués d'un point rouge.

```
<<<<<<< HEAD
<div>I have to say some stuff Because its cool to do so.</div>
=======
<div>I have to say some stuff. No you dont</div>
>>>>>>> branch 'master' of /var/data/merge-issue.git
```

Sur chacun des fichiers en conflit , on pourra déclencher le *menu contextuel* "**Team / Merge tool**" . (laisser par défaut la configuration de "Merge Tool" : use HEAD).

3) ***Saisir , changer ou supprimer alors au moins un caractère dans la zone locale (à gauche) + save*** :



… au cas par cas ….

4) Déclencher le menu contextuel "**Team / add to index**" pour ajouter le fichier modifié dans la liste de ceux à gérer (staging).
5) Effectuer un "**Team / commit**" local .
6) Effectuer un "**Team / push to upstream** …" pour mettre à jour le référentiel distant/partagé .

VII - Annexe – Accès distant via http (conf, admin)

1. Configuration d'accès distant à un référentiel Git

Si Répertoire "réseau" partagé (via NFS ou autre) , URL possible en file:///
(ex: file:///var/git/project.git)

1.1. Accès distant (non sécurisé) via git

URL de type **git://nomMachineOuDomaine/xxxx/projetYy.git**
où *xxx* est le chemin menant au référentiel git sur la machine "serveur" (ex: **/var/git/**)
(Alias possible dans .gitconfig)

1.2. Accès distant sécurisé via git+ssh

URL de type **ssh://user@hostXx/var/git/projectYy.git**

git+ssh est un tunneling sécurisé ssh pour le protocole git

Les clefs (publiques et privées) ssh sont placés dans le répertoire **$HOME/.ssh**
Celles ci se génèrent via la commande **ssh-keygen** .
Lors de la génération des clefs , un mot de passe (*passphrase*) à retenir est demandé .
Ce mot de passe peut éventuellement être vide (sécurité alors que via la clef publique) .
La **clef publique** (à envoyer par email ou ….) correspond au fichier ***id_dsa.pub*** (ou bien ***id_rsa.pub***) .

Si la partie "serveur" de ssh n'est pas encore installée , on peut alors lancer "**sudo apt-get install openssh-server**" puis éventuellement "***sudo service ssh start***" .
NB : sur certaines versions de Linux Ubuntu , la commande apt-get install ne fonctionne pas bien avec openssh-server et l'on peut dans ce cas installer alternativement openssh-server de la façon suivante :

1) télécharger le fichier ***openssh-server_5.9p1-5ubuntu1_i386.deb*** (via une recherche google)
2) lancer **sudo dpkg --install ./openssh-server_5.9p1-5ubuntu1_i386.deb**
3) redémarrer linux (ou …) pour que le service "ssh" soit activé

1.3. Accès distant en lecture seule via http (sans webdav)

Configurer un accès de apache2 vers un répertoire correspondant à un référentiel git et activer le hook "post-update" en renommant post-update.sample en post-update .

```
cd  xy.git
mv  hooks/post-update.sample   hooks/post-update
```

1.4. Accès distant en lecture/browsing via gitweb

En configurant sur le poste serveur , l'extension "gitweb"(pour apache2 et git) , on peut alors parcourir toute l'arborescence d'un projet GIT via un simple navigateur.

http://localhost/**gitweb**/?p=p0.git

Activation et configuration du site web "gitweb":

```
cd /var/www;
sudo mkdir gitweb;
cd gitweb;
sudo cp /usr/share/gitweb/* . ;
sudo cp /usr/share/gitweb/static/* .
```

Il faut également fixer la variable **$projectroot = "/var/scm/git/"**
dans le fichier **/etc/gitweb.conf**

/etc/gitweb.conf

```
$projectroot="/home/formation/scm/git";
# directory to use for temp files
$git_temp="/tmp";
# html text to include at home page
$home_text="indextext.html";
# file with project list; by default, simply scan the projectroot dir.
$projects_list=$projectroot;
# stylesheet to use
# I took off the prefix / of the following path to put these files inside gitweb directory directly
$stylesheet="gitweb.css";
# logo to use
$logo="git-logo.png";
# the 'favicon'
$favicon="git-favicon.png";
```

D'autre part, le module " RewriteEngine" d'apache2 doit être activé.
Si ce n'est pas encore le cas, on l'active via la commande "**sudo a2enmod rewrite**"

Créer et configurer un nouveau fichier pour configurer gitweb sous apache2 **:**

/etc/apche2/conf.d/git.conf

```
...
<Directory /var/www/gitweb >
SetEnv  GITWEB_CONFIG  /etc/gitweb.conf
DirectoryIndex gitweb.cgi
Allow from all
AllowOverride all
Order allow,deny
Options +ExecCGI
AddHandler cgi-script .cgi
<Files gitweb.cgi>
  SetHandler cgi-script
</Files>
RewriteEngine on
RewriteRule ^[a-zA-Z0-9_-]+.git/?(\?.)?$ /gitweb.cgi%{REQUESTURI} [L,PT]
</Directory>
….
```

Redémarrage du service apache2:
service apache2 restart

1.5. Accès distant "rw" via http/https (webdav)

Activer les ***modules apache2*** "**dav**" , "**dav_fs**" et "**dav_lock**"

sudo a2enmod dav
sudo a2enmod dav_fs
sudo a2enmod dav_lock

Créer et configurer un nouveau fichier pour configurer git sous apache2 **:**

/etc/apche2/conf.d/git.conf

```
...
Alias /git /var/scm/git/

<Location /git>
      DAV on
      #AuthType Basic
      #AuthName "Git"
      #AuthUserFile /etc/apache2/dav_git.passwd
      #Require valid-user
</Location>
...
```

Redémarrage du service apache2:
service apache2 restart

+ si besoin paramétrage d'autres détails (sécurité , …) :

```
<Directory "/var/scm/git/">
Options Indexes FollowSymLinks MultiViews ExecCGI
#DirectoryIndex index
AllowOverride None
Order allow,deny
allow from all
</Directory>
```

Eventuel paramétrage (facultatif et très délicat) pour optimiser les transferts (par paquets) via HTTP:

```
# Git-Http-Backend (for smart http push , useful with egit )
SetEnv GIT_PROJECT_ROOT /var/scm/git/
SetEnv GIT_HTTP_EXPORT_ALL
#ScriptAlias /git/     /usr/lib/git-core/git-http-backend/
ScriptAliasMatch \
        "(?x)^/git/(.*/(HEAD | \
                info/refs | \
                objects/(info/[^/]+ | \
                        [0-9a-f]{2}/[0-9a-f]{38} | \
                        pack/pack-[0-9a-f]{40}\.(pack|idx)) | \
                git-(upload|receive)-pack))$" \
                "/usr/lib/git-core/git-http-backend/$1"

<LocationMatch "^/git/.*/git-receive-pack$">
      #AuthType Basic
      #AuthName "Git Access"
      #Require group committers
</LocationMatch>
```

VIII - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

https://git-scm.com/	site officiel de la technologie git
https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git	tutoriel sur GIT

2. TP

2.1. Installation de git en mode texte

Installer git en fonction du type d'ordinateur

Exemple pour linux debian ou ubuntu :

```
$ sudo apt update
$ sudo apt upgrade -y
$ sudo apt install git
$ git --version
```

Exemple pour windows :

https://git-scm.com/download *site officiel pour télécharger l'installeur de git*

https://github.com/git-for-windows/git/releases/download/v2.41.0.windows.1/Git-2.41.0-64-bit.exe

Lancer l'installeur avec choix par défauts

Vérification:

git --version
git version 2.41.0.windows.1

2.2. Configuration locale fondamentale de git

git config --global user.name "*PrenomNom*"
git config --global user.email "*prenom.nom@ici_ou_la.fr*"
git config –list

2.3. Eventuelle installation de compléments graphiques :

Liste des clients "git" graphiques: https://git-scm.com/downloads/guis

TortoiseGit *(gratuit , sous windows)*

https://tortoisegit.org/download/

https://download.tortoisegit.org/tgit/2.14.0.0/TortoiseGit-2.14.0.1-64bit.msi

GitKraken

https://www.gitkraken.com/download/windows64 *(produit payant , version gratuite limitée à 7jours à priori)*

+ ***plugins "git" des principaux IDE*** *(eclipse, intelliJ , Visual studio code , ...)*

2.4. Git élémentaire en mode local

Se créer un répertoire de tp (ex: ***c:\tp\tp-git*** *)*

Créer les sous répertoires c:\tp\tp-git***local-git-repositories-devA**

et c:\tp\tp-git***local-git-repositories-devA\p1**

Au sein du répertoire p1 , créer (par exemple via notepad++) les fichiers simples suivants :

p1\f1.txt

```
ligne1
ligne2
```

p1\f2.txt

```
abc
def
```

p1\inutile.bad

```
fichier pas toujours utile et potentiellement de grande taille
```

Création d'un référentiel local git au sein du projet p1 :

se placer dans ***local-git-repositories-devA\p1***

```
git init
```

→ Visualiser via l'explorateur de fichiers , le **nouveau sous répertoire caché** "**.git**" .

Générer (dans p1) le souvent indispensable fichier **.gitignore**

```
echo *.bad > .gitignore
```

Visualiser le contenu de ***.gitignore*** via notepad++ et ajouter y les lignes suivantes :

.gitignore

```
*.bad
*.jar
/node_modules
/bin
/target
```

Lancer les commandes suivantes et visualiser bien les effets de chacune d'elles

```
git status
git add *.txt
git add .gitignore
git status
git commit -m "initial"
git status
git log
```

En exercice :

- **ajouter** le fichier ***f3.txt*** contenant "*f3 c'est 3 fois rien*"
- **modifier** le contenu de ***f1.txt*** avec quelques majuscules
- lancer git status , git add … , git commit -a -m "…" de façon à enregistrer tous les changements
- réafficher l'historique via **git log --stat**
- **ajouter** le fichier ***f4.txt*** contenant "*f4 est un nouveau fichier*"
- **modifier** le contenu de ***f1.txt*** avec une ligne en plus
- lancer git status , git add … , git commit -a -m "…" de façon à enregistrer tous les changements
- **réafficher l'historique via git log**
  et copier dans le presse papier l'**id de l'avant dernier commit** associé à "*f3 et modif f1*"
  (exemple : ***775c58b7a5f39c2d2d61d1e0bcc7bcbe3f412905***)

Retour en arrière (vers ancienne version sauvegardée) :

```
git checkout 775c58b7a5f39c2d2d61d1e0bcc7bcbe3f412905
```

→ visualiser le nouvel état de p1 (f4.txt a disparu et f1.txt ne comporte plus le 3ème ligne ajoutée)

Revenir en tête de branche principale :

```
git checkout main
git log
```

→ visualiser le nouvel état de p1 (f4.txt est revenu et f1.txt comporte la 3ème ligne ajoutée)

Associer le tag *v1* à l'avant dernier commit :

```
git tag v1 775c58b7a5f39c2d2d61d1e0bcc7bcbe3f412905
git tag -l
git log
```

git checkout tags/v1

→ *visualiser l'absence de f4.txt*

git checkout main

→ *visualiser le retour de f4.txt*

2.5. Gestion élémentaire des branches de git

git branch

→*par défaut une seule branche master ou* **main**

création d'une nouvelle branche b1 :

```
git branch b1
```

Se positionner sur branche b1

git checkout b1

git branch

main

** b1 (avec * devant branche courante/sélectionnée).*

En exercice :

- ajout de f5.txt et nouveau commit sur branche b1
- ajout de f6.txt et encore nouveau commit sur branche b1
- git log

- se positionner sur branche main (via git checkout)
  → visualiser absence de f5.txt et f6.txt
- se positionner sur branche b1 (via git checkout)
  → visualiser retour de f5.txt et f6.txt

Fusion élémentaire de branches (fast forward) :

Modifications effectuées que sur b1 , aucune modif effectuées sur main → fast forward

git checkout main
git merge b1
git log
git branch -d b1 ← *suppression ancienne branche b1*
git log

2.6. Merge de fusion sans conflit

En exercice :

- partir du sommet de la branche main
- créer un nouvelle branche temporaire b2
- se placer sur la branche b2
- ajouter le fichier fa.txt
- ajouter *** en fin de la première ligne de f1.txt
- commiter tous ces changements
- se placer sur la branche main
- ajouter fb.txt
- ajouter dernièreNouvelleLigne### en fin de f1.txt (sans changer les premières lignes)
- commiter tous ces changements
- rester sur la branche main
- déclencher une fusion avec la branche b2
- visualiser tous les effets (git log et nouveau commit de fusion, état des fichiers fa.txt , fb.txt , f1.txt)
- supprimer l'ancienne branche temporaire b2

2.7. Merge git avec résolution de conflit

- partir du sommet de la branche main
- créer un nouvelle branche temporaire b3

- se placer sur la branche b3
- modifier la première ligne de f2.txt (abc transformé en ABC)
- commiter tous ces changements
- se placer sur la branche main
- modifier différement la première ligne de f2.txt (abc transformé en a_b_c)
- commiter tous ces changements
- rester sur la branche main
- déclencher une fusion avec la branche b3
- visualiser tous le conflit (message de la commande, état du fichier f2.txt)

  Auto-merging f2.txt

  CONFLICT (content): Merge conflict in f2.txt

  Automatic merge failed; fix conflicts and then commit the result.

- **git status**
- On pourrait éventuellement annuler la fusion via *git merge --abort* (si trop de conflits à résoudre). On va préférer résoudre le conflit pour finaliser la fusion.
- visualiser l'état temporaire de ***f2.txt***

```
<<<<<<< HEAD
a_b_c
=======
ABC
>>>>>>> b3
def
```

- ne garder dans ***f2.txt*** qu'une des versions proposées a_b_c ou ABC ou un mixte des 2.

```
A_B_C
def
```

- indiquer la fin de la résolution du conflit sur f2.txt via la commande
  git add f2.txt
- finaler la fusion via la commande

  git commit
- git log
- supprimer l'ancienne branche temporaire b3

NB : un futur TP permettra d'expérimenter un merge avec un IDE tel que eclipse/java .

2.8. Git élémentaire en mode remote (clone, push, pull)

*Se placer dans répertoire de tp (ex: **c:\tp\tp-git**)*

Créer les sous répertoires c:\tp\tp-git***local-git-repositories-devB**

et c:\tp\tp-git***git-shared-repositories***

NB :

- ***git-shared-repositories*** *regoupera des* ***référentiels git partagés*** *(éventuellement accessibles à distance selon accès réseaux ...)*
- ***local-git-repositories-devA*** *regoupera des référentiels locaux pour le développeur devA*
- ***local-git-repositories-devB*** *regoupera des référentiels locaux pour le développeur devB*

Se placer dans le répertoire c:\tp\tp-git***git-shared-repositories***

et créer le sous répertoire c:\tp\tp-git***git-shared-repositories\p1.git**

Se placer dans le répertoire c:\tp\tp-git***git-shared-repositories\p1.git**

et ***initialiser le référentiel distant/partagé*** *en lançant la commande*

```
git init --bare
```

Visualiser le contenu de p1.git

Référencement de ***p1.git*** *en tant que* ***version distante/partagée*** *de* ***local-git-repositories-devA/****p1*

Se placer dans **local-git-repositories-devA/**p1 et lancer les commandes suivantes :

```
git remote -v
(git remote remove originTpP1 pour supprimer ancienne config en cas d'erreur d'URL)
git remote add originTpP1  file:///C:\tp\tp-git\git-shared-repositories\p1.git
git remote -v
```

NB: ***originTpP1 est à voir comme un alias pour l'url*** ***file:///C:\tp\tp-git\git-shared-repositories\p1.git***

Première publication partagée depuis ***local-git-repositories-devA/****p1:*

```
git push --set-upstream  originTpP1 main
```

Clonage du référentiel partagé depuis ***local-git-repositories-devB*** *(développeur B) :*

Se placer dans c:\tp\tp-git***local-git-repositories-devB**

```
git clone file:///C:\tp\tp-git\git-shared-repositories\p1.git
```

→ *Visualiser le sous répertoire p1 construit (par clonage)*

cd p1 *(depuis ...-devB)*

git remote -v

origin file:///C:\tp\tp-git\git-shared-repositories\p1.git (fetch)

origin file:///C:\tp\tp-git\git-shared-repositories\p1.git (push)

git checkout main

→ *visualiser les copies locales des fichiers f1.txt , ...*

Modif et publication depuis un développeur de l'équipe dev-B ou dev-A

- *modifier un des fichiers (ex : nouvelle ligne dans f1.txt)*
- *intégration d'éventuelles modifications distantes via git pull*
- *commit local*
- *publication via* ***git push***

Rapatriation des modifications enregistrées/partagées depuis autre développeur dev-B ou dev-A

- ***git pull***
- *visualiser les changements*

Variantes d'URL pour ce TP *:*

file:///C:\tp\tp-git\git-shared-repositories\p1.git *(sur un seul ordinateur)*

file:///Y:\git-shared-repositories\p1.git *où Y: est un lecteur réseau associé à un répertoire partagé par un autre ordinateur via le réseau*

https://github.com/xyz/p1.git *où p1.git est un référentiel git partagé géré par l'utilisateur xyz ayant un compte sur github .*

2.9. Merge git avec résolution de conflit et branches distantes

Expérimenter des commits/push/pull avec conflits à résoudre entre deux utilisateurs travaillant sur un même fichier d'un même projet (ex : p1).

On pourra effectuer se Tp en simulant deux utilisateurs sur le même ordinateur (configuration du TP précédent) ou bien en configurant plusieurs comptes ***gihub*** *ou* ***gitlab*** *ou* ***gitbucket*** *.*

Bonnes pratiques (en général) :

- commit d'abord sur branche locale

- pull avant merge ou push

2.10. Git avec résolution de conflit depuis un IDE

1. Créer un référentiel git comportant un petit projet java/maven très simple
2. Charger ce projet depuis un IDE (ex : eclipse , intelliJ , visual studio code, …)
3. Expérimenter les opérations git basiques (commit, push, pull)
4. Expérimenter les résolutions de conflits depuis l'IDE

2.11. Git en mode http (via github ou autre)

1. Se créer un compte gratuit sur github ou gitlab (une adresse email valide sera vérifiée) et mémoriser les infos d'authentification (username/password ou token ou ...)
2. Créer un nouveau référentiel distant (vide au départ)
3. Publier un projet local vers le référentiel distant (git remote … , git push …)
4. Cloner le référentiel distant ailleurs (git clone , ….)
5. effectuer une série de changements (commit , push , pull, …) sans conflit puis avec conflit soit en ligne de commande , soit via tortoiseGit , soit via un IDE (ex : eclipse).
6. On pourra éventuellement autoriser d'autres développeurs amis à effectuer des publications sur notre référentiel partagé github ou gitlab .

2.12. Expérimentation de certains aspects avancés de GIT

- Rebase , cherry-pick , ...