Afpa
se former, avancer

Versionnez-vous la vie !!

GESTIONNAIRE DE VERSIONS

Un gestionnaire de versions est un programme qui permet aux développeurs de conserver un historique des modifications et des versions de tous leurs fichiers.

L'action de contrôler les versions est aussi appelée "versioning" en anglais, vous pourrez entendre les deux termes.

- Si vous travaillez seul, vous pourrez garder l'historique de vos modifications ou revenir à une version précédente facilement.
- Si vous travaillez en équipe, plus besoin de mener votre enquête ! Le gestionnaire de versions fusionne les modifications des personnes qui travaillent simultanément sur un même fichier. Grâce à ça, vous ne risquez plus de voir votre travail supprimé par erreur !

Cet outil a donc trois grandes fonctionnalités :

1. Revenir à une version précédente de votre code en cas de problème.
2. Suivre l'évolution de votre code étape par étape.
3. Travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs.

GIT vs GITHUB ou GITLAB ou BITBUCKET

GIT

Git est un gestionnaire de versions.

Vous l'utiliserez pour créer un dépôt local et gérer les versions de vos fichiers.

GitHub ou GitLab ou Bitbucket

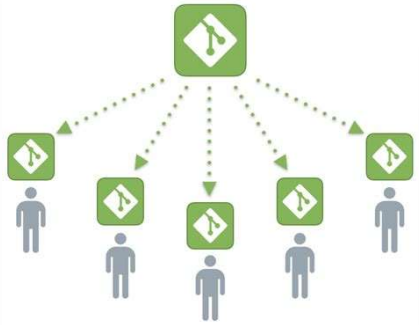
GitHub (ou GitLab/Bitbucket) est un service en ligne qui va héberger votre dépôt.

Dans ce cas, on parle de dépôt distant puisqu'il n'est pas stocké sur votre machine.

Ce sont des services payants mais ils offrent une possibilité d'utilisation en version gratuite.

GIT

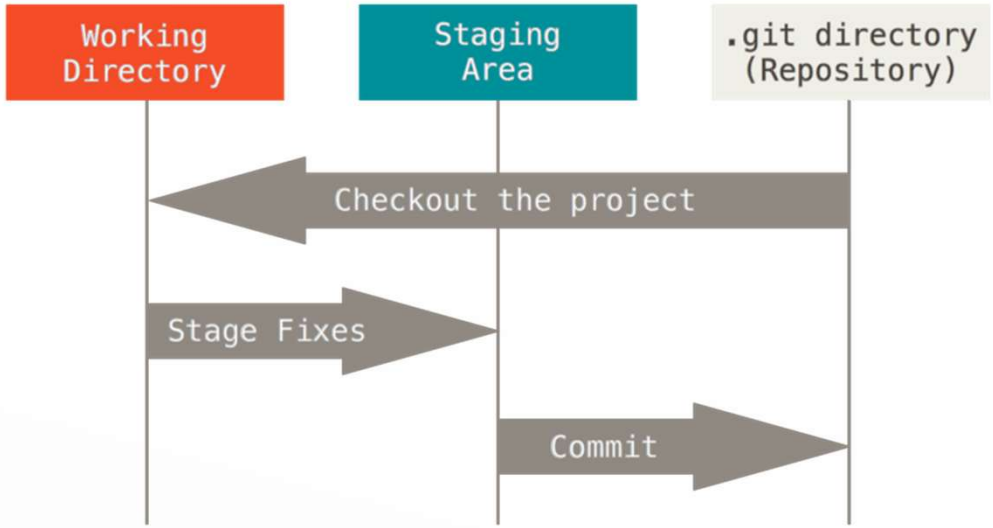
- Open source, projet débuté en 2005 par Linus Torvalds pour palier à l'alternative BitKeeper.
- Outil très largement utilisé.
- Outil de gestion de versions décentralisé.
  - L'historique complet du code n'est pas conservé dans un unique emplacement. Chaque copie correspond à un dépôt où est conservé l'historique des modifications.
- https://git-scm.com/



GIT : LES TROIS ÉTATS

Git gère trois états dans lesquels les fichiers peuvent résider :

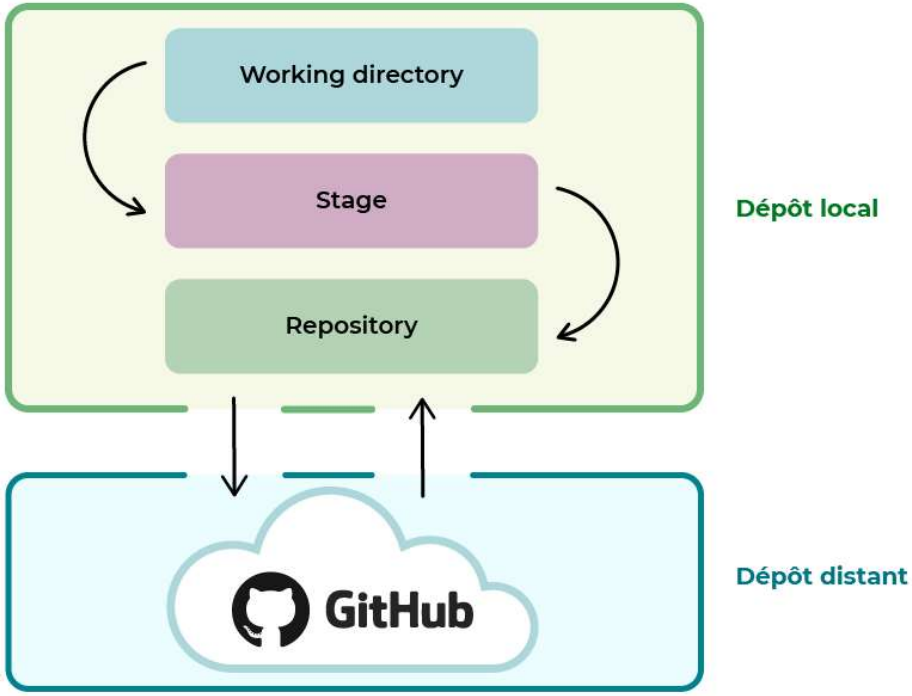
- Modifié : fichier modifié mais pas encore validé en base.
- Indexé : fichier marqué dans sa version actuelle pour faire partie du prochain instantané du projet.
- validé : données stockées en sécurité dans votre base de données locale.



GIT : PRINCIPE

L'utilisation standard de Git se passe comme suit :

1. Vous modifiez des fichiers dans votre répertoire de travail (Working directory).
2. Vous indexez (stagez) les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index (Stage)
3. Vous validez, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans la base de données du répertoire Git (Repository).



GIT : PRINCIPE

Le répertoire .git est l'endroit où Git stocke les métadonnées et la base de données des objets de votre projet. C'est la partie la plus importante de Git, et c'est ce qui est copié lorsque vous clonez un dépôt depuis un autre ordinateur.

Le répertoire de travail (Working directory) est une extraction unique d'une version du projet. Ces fichiers sont extraits depuis la base de données compressée dans le répertoire Git et placés sur le disque pour pouvoir être utilisés ou modifiés.

La zone d'index (stage) est un simple fichier, généralement situé dans le répertoire .git, qui stocke les informations concernant ce qui fera partie du prochain instantané. On l'appelle aussi des fois la zone de préparation.

GIT : LIGNE DE COMMANDE

Il existe de nombreuses manières différentes d'utiliser Git. Il y a les outils originaux en ligne de commande et il y a de nombreuses interfaces graphiques avec des capacités variables.

Nous utiliserons Git en ligne de commande. Tout d'abord, la ligne de commande est la seule interface qui permet de lancer **toutes** les commandes Git - la plupart des interfaces graphiques simplifient l'utilisation en ne couvrant qu'un sous-ensemble des fonctionnalités de Git.

Si vous savez comment utiliser la version en ligne de commande, vous serez à même de comprendre comment fonctionne la version graphique, tandis que l'inverse n'est pas nécessairement vrai.

GIT : INSTALLATION ET PARAMÉTRAGE

Installation des logiciels

- PowerShell
- https://git-scm.com/download/win
  - Prendre la version actuelle pour windows.

Paramétrage à la 1ère utilisation de GIT

- vérifier le paramétrage de Git

git config --list

- Votre identité :

git config --global user.name "John Doe"

git config --global user.email "votre@mail"

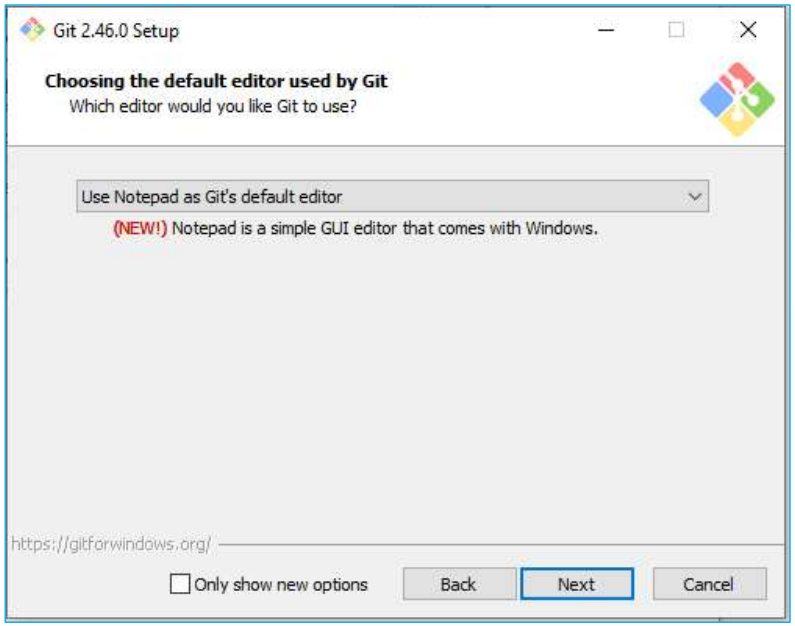
- Votre éditeur de texte :

git config --global core.editor notepad

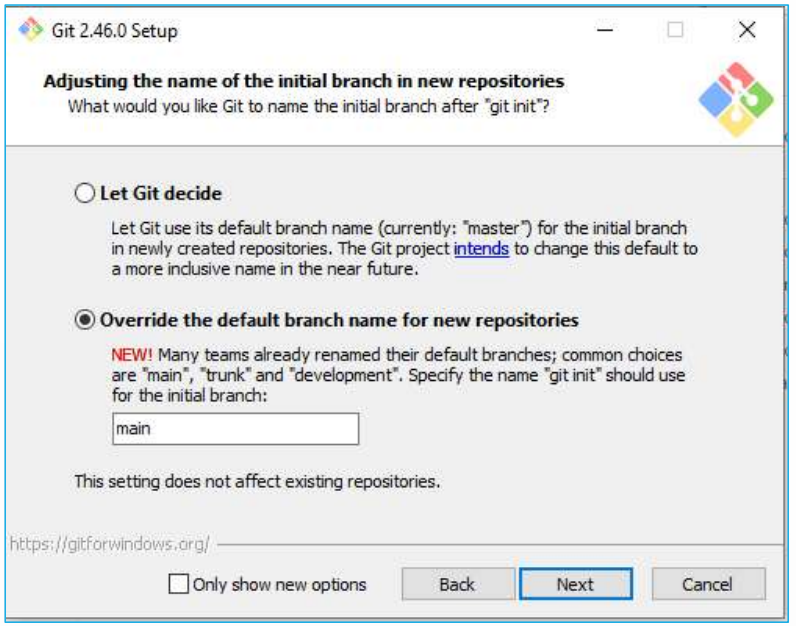
Options core.editor

DEUX OPTIONS IMPORTANTES À L'INSTALLATION

Editeur pour les commit



Branche principale



GIT : PARAMÉTRAGE ET AIDE

Autres paramétrages

- Votre nom de branche par défaut :

Par défaut Git crée une branche nommé main
Depuis la version 2.28, on peut définir le nom de branche par défaut.

git config --global init.defaultBranch main

- Vérifier vos paramètres :

git config --list

Obtenir de l'aide

git help <commande>

git <commande> --help

GIT : LES BASES

- Démarrer un dépôt

Vous pouvez principalement démarrer un dépôt Git de deux manières.

1. Vous pouvez prendre un répertoire existant et le transformer en dépôt Git.
2. Vous pouvez *cloner* un dépôt Git existant sur un autre serveur.

Dans les deux cas, vous vous retrouvez avec un dépôt Git sur votre machine locale, prêt pour y travailler.

- Initialisation d'un dépôt dans un répertoire existant

git init

- *Cela crée un sous répertoire .git qui contient tous les fichiers nécessaires au dépôt.*

- Cloner un dépôt existant

git clone [url]

GIT : ENREGISTRER DES MODIFICATIONS

- Vérifier l'état des fichiers

L'outil principal pour déterminer quels fichiers sont dans quel états est la commande git status.

Les fichiers non indexés sont alors indiqués en rouge par Git et vous propose de les inclure

- Placer de nouveaux fichiers sous suivi de version

Pour commencer à suivre un fichier, la commande à utiliser est git add nomFichier

1. On a la possibilité d'indiquer l'ensemble du répertoire avec git add * ou git add -A
2. Ensuite, on valide la présence d'un fichier avec la commande git status
3. A partir de l'ajout du fichier, celui-ci est indexé (passage en vert) et prêt à être versionné.

GIT : .GITIGNORE

- Ignorer des fichiers / des répertoires

Souvent, des fichiers n'ont pas besoin d'être ajoutés au dépôt (fichier de configuration, logs, sauvegardes etc..). Pour cela, on peut fournir à Git, les patrons de noms de fichiers à ignorer dans un fichier .gitignore placé à la racine de votre répertoire de travail.

Les règles de construction des patrons à placer dans le fichier .gitignore sont les suivantes :

- les lignes vides ou commençant par # sont ignorées ;
- les patrons standards de fichiers sont utilisables et seront appliqués récursivement dans tout l'arbre de travail ;
- si le patron commence par une barre oblique (/), le patron n'est pas récursif ;
- si le patron se termine par une barre oblique (/), il indique un répertoire ;
- un patron commençant par un point d'exclamation (!) indique des fichiers à inclure malgré les autres règles.

EXEMPLE DE .GITIGNORE POUR JAVA

Voici un exemple de .gitignore généré par GitHub à la création d'un dépôt.

```
### IntelliJ IDEA ###
out/
!**/src/main/**/out/
!**/src/test/**/out/

### Eclipse ###
.apt_generated
.classpath
.factorypath
.project
.settings
.springBeans
.sts4-cache
bin/
!**/src/main/**/bin/
!**/src/test/**/bin/
```

GIT : ENREGISTRER DES MODIFICATIONS

- Dé-indexer un fichier

git rm --cached <nomFichier>

- Inspecter les modifications indexées et non indexées

Si le résultat de la commande git status est encore trop vague — lorsqu'on désire savoir non seulement quels fichiers ont changé mais aussi ce qui a changé dans ces fichiers — on peut utiliser la commande git diff

GIT : COMMIT

- Valider vos modifications

Lorsqu'on est prêt à valider nos fichiers, il faut donner l'ordre à git de "commit" par la commande git commit

Dans le paramétrage de votre application, on a indiqué l'éditeur à utiliser pour permettre d'indiquer un message de validation de notre commit. Cependant, vous pouvez directement indiquer ce message par la commande :

git commit -m "message"

- Visualiser l'historique des validations

Après avoir réalisé des commits, ou cloner un dépôt ayant un historique, on peut visualiser l'historique des commits avec git log

GIT : DIVERSES ACTIONS

- Oublie d'un fichier après un commit : utilisation de --amend

```
git commit -m 'validation initiale'
git add fichier_oublie
git commit --amend        // --amend fait en sorte de valider le commit avec le même message
```

- Réinitialiser un fichier modifié

```
git restore <nomFichier>
```

TRAVAILLER AVEC DES DÉPÔTS DISTANTS

Pour pouvoir collaborer sur un projet Git, il est nécessaire de savoir comment gérer les dépôts distants (version de votre projet, hébergées sur internet ou sur un réseau d'entreprise).

On peut en avoir plusieurs avec des droits de lectures, d'écritures, ou de lectures/écritures.

Collaborer sur un projet Git consiste en poussant ou en tirant des données depuis et vers ce dépôt distant.

Gérer un dépôt inclut de savoir comment ajouter, effacer, gérer des branches etc…

TRAVAILLER AVEC DES DÉPÔTS DISTANTS

Afficher les dépôts distants

git remote -v

Liste les noms des différentes références distantes qu'on a spécifiées.

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Ajouter un nouveau dépôt

git remote add [nomcourt] [url]

TRAVAILLER AVEC DES DÉPÔTS DISTANTS

Récupérer et tirer depuis le dépôt

git fetch [remote-name]

- Il faut noter que la commande fetch tire les données dans votre dépôt local mais sous sa propre branche — elle ne les fusionne pas automatiquement avec aucun de vos travaux ni ne modifie votre copie de travail. Vous devez volontairement fusionner ses modifications distantes dans votre travail lorsque vous le souhaitez.

git pull [remote-name]

- Il faut noter que la commande pull récupère et fusionne automatiquement une branche distante dans votre branche locale.

Pousser son travail sur un dépôt distant

git push [nom-distant] [nom-branche]

- git push origin main

TRAVAILLER AVEC DES DÉPÔTS DISTANTS

Inspecter un dépôt distant

git remote show origin

- Donne la liste des URL pour le dépôt distant ainsi que la liste des branches distantes suivies, des informations sur le pull et le push etc…

```
$ git remote show origin
* distante origin
  URL de rapatriement : https://github.com/schacon/ticgit
  URL push : https://github.com/schacon/ticgit
  Branche HEAD : master
  Branches distantes :
    master suivi
    ticgit suivi
  Branche locale configurée pour 'git pull' :
    master fusionne avec la distante master
  Référence locale configurée pour 'git push' :
    master pousse vers master (à jour)
```

Retirer et renommer des dépôts distants

git remote rename [oldname] [newName]

- Modifie également le nom des branches du dépôt

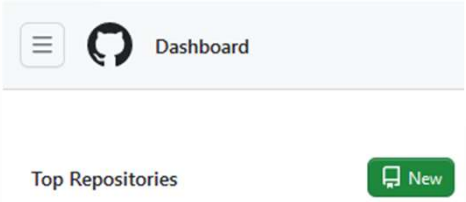
git remote rm [name]

RÉSUMÉ

Utilisation de Git dans un contexte de projet.

Après avoir créé son compte GitHub (ou GitLab) :

1. Création d'un repository qui sera le dépôt distant.



2. Depuis ce nouveau dépôt, copier le lien https correspondant à notre dépôt.

1. Se placer dans le dossier local du projet que l'on souhaite "git"
2. Initialiser le dossier local : git init
3. Se positionner dans la branch main : git branch -M main
4. Lier le dossier local avec le dossier distant : git remote add origin https:…
5. Créer le fichier .gitignore à la racine de votre projet pour indiquer à Git ce qu'il doit ignorer de vos dossiers.
6. Ajouter les fichiers / répertoires : git add *
    1. git status pour voir si cela correspond à vos souhaits
7. Préparer le commit : git commit -m "message du commit"
8. Pousser vers le dépôt distant : git push -uf origin main
    1. A l'inverse, si on souhaite récupérer de notre dépôt distant : **git pull** à faire dans le dossier projet

PARLONS DU HEAD

PARLONS DE HEAD

Pour mieux comprendre la suite, il nous faut préciser un peu mieux le rapport entretenu par Git entre l'historique et le répertoire de travail.

De manière conceptuelle, sans trop entrer dans les détails donc, il faut considérer que :

1. en utilisant la commande git add, on demande à Git de suivre certains fichiers, c'est-à-dire de prendre en charge la gestion de l'évolution de ces fichiers,
2. par la commande git commit, nous demandons à Git d'enregistrer un snapshot, c'est-à-dire une photographie instantanée de l'état des fichiers suivis (plus exactement, des modifications qui ont été placées dans l'index, mais il s'agit là en fait d'une facilité),

• une suite des commits constitue un historique.

RAPPORT ENTRE RÉPERTOIRE DE TRAVAIL ET COMMITS

Parlons de HEAD

Git va alors voir votre répertoire de travail comme étant en fait composé de

- l'ensemble des fichiers suivis, dans leur version correspondant à UN SNAPSHOT COURANT (un commit)
- PLUS les modifications existant sur ces fichiers par rapport à cette version spécifique
- PLUS des fichiers non suivis (pour lesquels on n'a pas effectué de git add)

!!! Relisez bien cela, c'est un des secrets pour utiliser Git avec facilité !!!

Pour désigner le SNAPSHOT COURANT, Git utilise une référence (c'est à dire, un pointeur vers un commit) qui s'appelle HEAD.

- Il existe une autre référence, que l'on va bientôt manipuler, et qui pointe vers la tête de l'historique.

Sachez juste pour l'instant qu'elle s'appelle main, on verra plus tard pourquoi elle porte ce nom.

MANIPULATION DE LA RÉFÉRENCE HEAD

Parlons de HEAD

Nous avons déjà vu une commande qui modifie ces deux références. En effet la commande git commit effectue deux choses :

1. Elle enregistre un snapshot ayant un lien de parenté avec le commit référencé par HEAD
2. Elle déplace la référence HEAD et la référence main sur ce nouveau commit.

Une autre commande modifiant la référence HEAD, que vous manipulerez bientôt, est git checkout.

- git checkout 56cd8a9 : la référence HEAD se déplace sur le commit 56cd8a9
- git checkout HEAD^ : la référence HEAD se déplace d'un cran vers le gauche (HEAD^ désigne le commit précédant HEAD)
- git checkout main : la référence HEAD se déplace vers la référence main, c'est-à-dire la tête de l'historique

Celle-ci va spécifiquement déplacer le HEAD vers un commit que vous lui précisez. En règle générale, elle ne peut être appliquée que si il n'y a pas de modifications en cours dans vos fichiers.

Pour savoir sur quel commit pointent les références, vous pouvez, entre autres, utiliser l'option --decorate de git log

INFLUENCE SUR LE RÉPERTOIRE DE TRAVAIL

Parlons de HEAD

- Si vous avez bien compris ce qui a été dit plus haut, que se passe-t-il au niveau de votre répertoire de travail lorsque HEAD est déplacé ?
  - Rappelons : votre répertoire de travail est constitué des fichiers suivis dans leur version correspondant au commit référencé par 'HEAD'

- Par conséquent, git checkout <idCommit>, par exemple, va déplacer HEAD sur le commit désigné, puis va faire en sorte que votre répertoire de travail soit constitué des fichiers dans leur version correspondant au nouveau HEAD.
  - Les fichiers sont donc modifiés, ajoutés, ou supprimés, pour que votre répertoire de travail soit dans l'état qui était le sien au moment du commit désigné...

Note : vous comprenez pourquoi il n'est pas possible (sauf dans un cas bien précis) d'utiliser git checkout s'il y a des modifications en cours : elles seraient perdues. C'est ce que Git vous répondra, en refusant d'effectuer le git checkout.

Afpa

LE SYSTÈME DE BRANCHES
LA MAGIE DE GIT

LES BRANCHES

Le principal atout de Git est son système de branches.

Les différentes branches correspondent à des copies de votre code principal à un instant T, où vous pourrez tester toutes vos idées les plus folles sans que cela impacte votre code principal.

Sous Git, la branche principale est appelée la ***branche main***, (ou ***master*** pour les dépôts créés avant octobre 2020).

La branche principale (main ou master) portera l'intégralité des modifications effectuées.

- Le but n'est donc pas de réaliser les modifications directement sur cette branche, mais de les réaliser sur d'autres branches, et après divers tests, de les intégrer sur la branche principale.
- Avec Git et ses fameuses branches, pas de soucis. Vous pouvez créer une branche correspondant à une évolution et cela sans toucher à votre application en cours de production.

Git est l'outil idéal dans ce cas. Il va créer une branche virtuelle, mémoriser tous vos changements, et seulement quand vous le souhaitez, les ajouter à votre application principale.

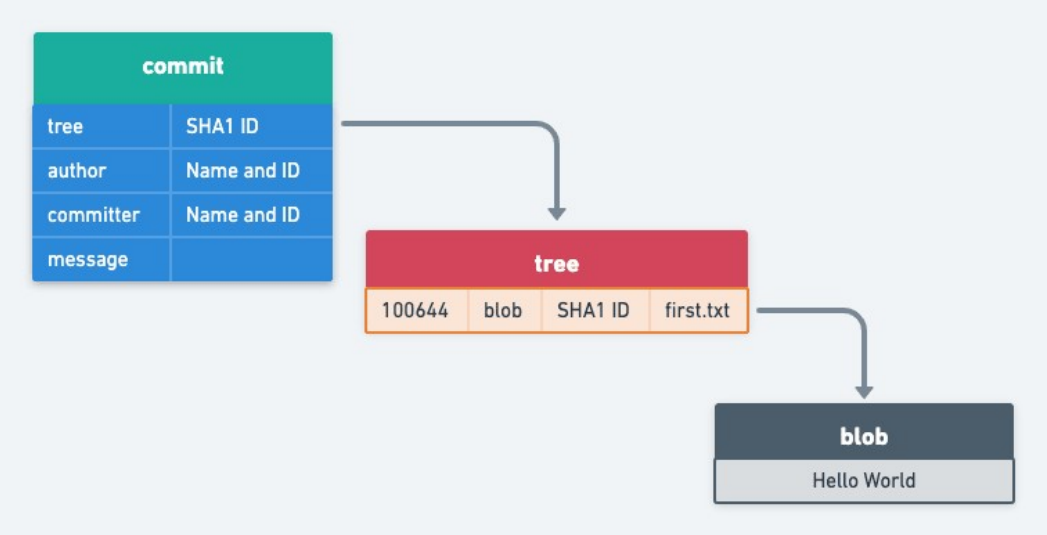
Il va vérifier s'il n'y a pas de conflits avec d'autres fusions, et hop, le tour est joué !

COMPOSITION D'UN DÉPÔT GIT

Un dépôt Git est composé de la manière suivante :

1. Un blob pour le contenu de chacun des fichiers
2. Un arbre qui liste le contenu du répertoire et spécifie quels noms de fichiers sont attachés à quels blobs
3. Un objet commit portant le pointeur vers l'arbre de la racine, ainsi que toutes les metadonnées attachées au commit.

Git ne stocke pas ses données comme une série de modifications ou de différences successives mais plutôt comme une série d'instantanés (appelés **snapshots**).



LES BRANCHES

Créer une nouvelle branche

Vous utilisez pour cela la commande

- git branch nom_de_la_branche

Quand on crée une branche, cela crée un nouveau pointeur.

Sur un dépôt, si je crée une branche test, cela nous donnera un pointeur sur le commit courant.

Comment Git connaît-il alors la branche sur laquelle on se trouve ?

- Il conserve un pointeur spécial appelé HEAD qui est le pointeur sur la branche local où on se trouve.

Voir les branches du dépôt local

Pour connaître, les branches de notre dépôt

- git branch

Pour connaître, sur quelle branche on se trouve

- git log --online --decorate

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA/Sparadrap (main)
$ git branch
* main
  test

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA/Sparadrap (main)
$ git log --oneline --decorate
0a257c9 (HEAD -> main, test) initial commit
```

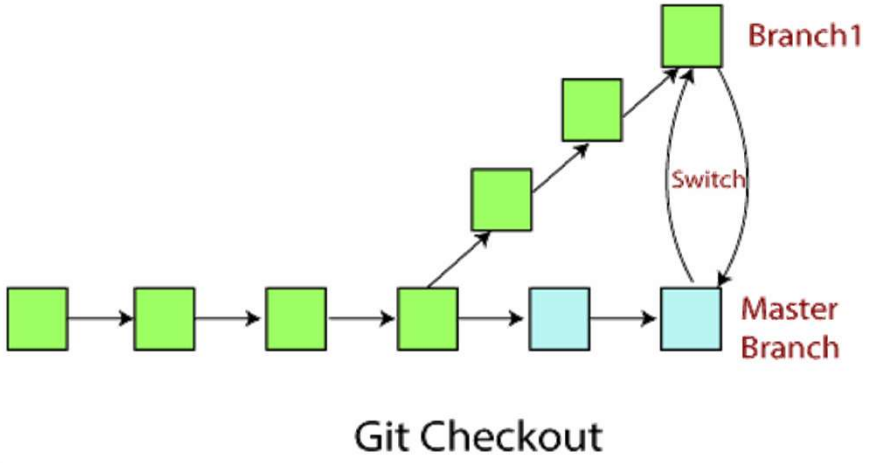
BASCULER ENTRE BRANCHES

Pour basculer sur une branche existante, il suffit de lancer la commande

git checkout nom_de_la_branche

Cela déplace HEAD pour le faire pointer vers la branche choisie.

1. A partir de ce moment, si je viens à modifier/créer un fichier alors je crée un nouveau nœud
2. La branche main pointe toujours sur le commit avant le changement de branche.

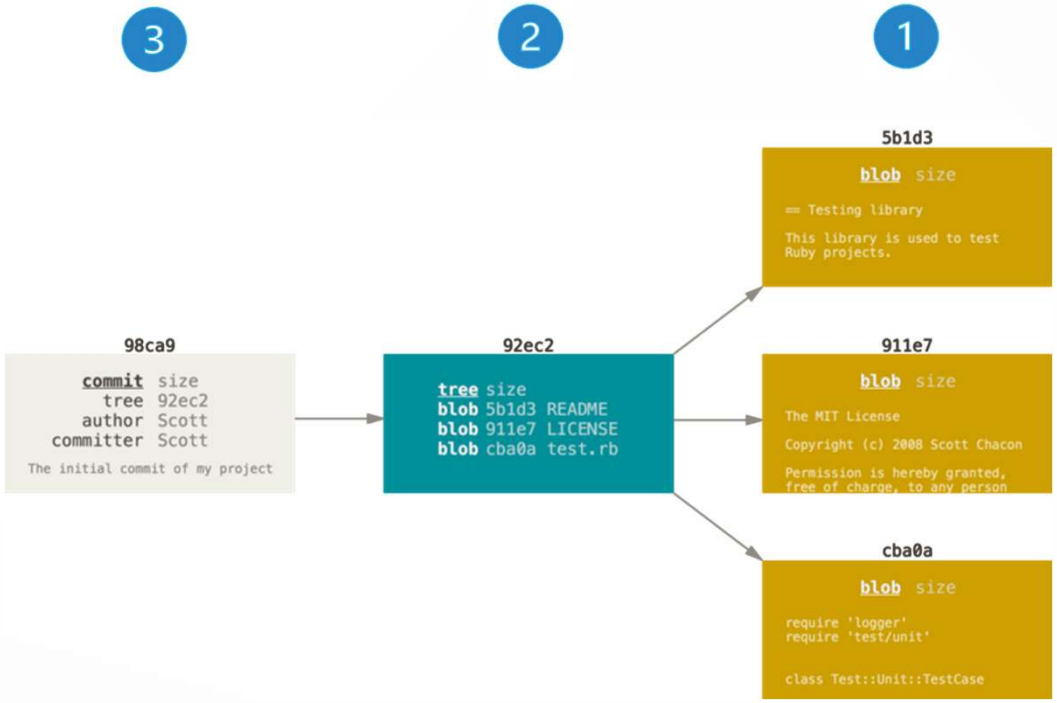


PAR L'EXEMPLE [1]

Pour visualiser ce concept, supposons que vous avez un répertoire contenant trois fichiers que vous indexez puis validez.

1. L'indexation des fichiers calcule une empreinte (checksum) pour chacun, stocke cette version du fichier dans le dépôt Git (Git les nomme blobs) et ajoute cette empreinte à la zone d'index (staging area) :
2. Lorsque vous créez le commit en lançant la commande git commit, Git calcule l'empreinte de chaque sous-répertoire (ici, seulement pour le répertoire racine) et stocke ces objets de type arbre dans le dépôt Git.
3. Git crée alors un objet commit qui contient les métadonnées et un pointeur vers l'arbre de la racine du projet de manière à pouvoir recréer l'instantané à tout moment.

Votre dépôt Git contient à présent cinq objets : un blob pour le contenu de chacun de vos trois fichiers, un arbre (tree) qui liste le contenu du répertoire et spécifie quels noms de fichiers sont attachés à quels blobs et enfin un objet commit portant le pointeur vers l'arbre de la racine ainsi que toutes les métadonnées attachées au commit.



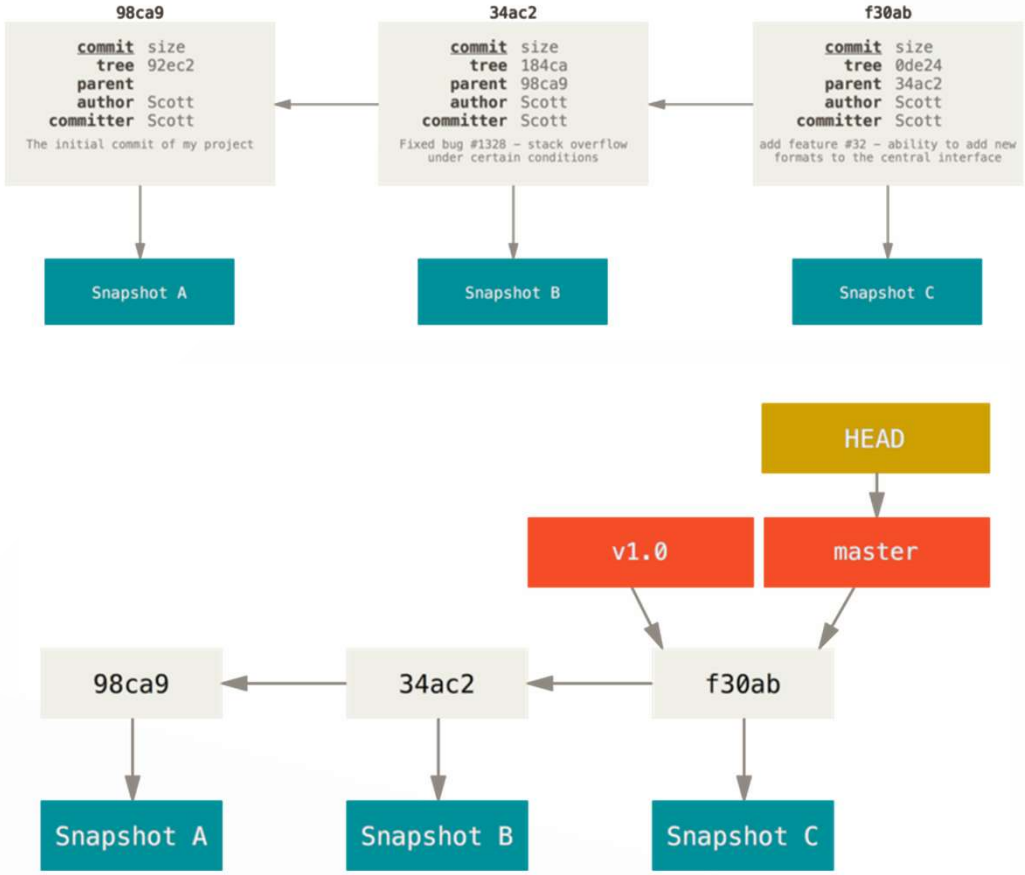
PAR L'EXEMPLE 2

Si on effectue des modifications et validez à nouveau, le prochain commit stock un pointeur vers le commit le précédant immédiatement

Une branche dans Git est simplement un pointeur léger et déplaçable vers un de ces commits. La branche par défaut dans Git s'appelle main.

Au fur et à mesure des validations, la branche main pointe vers le dernier des commits réalisés.

À chaque validation, le pointeur de la branche main avance automatiquement.



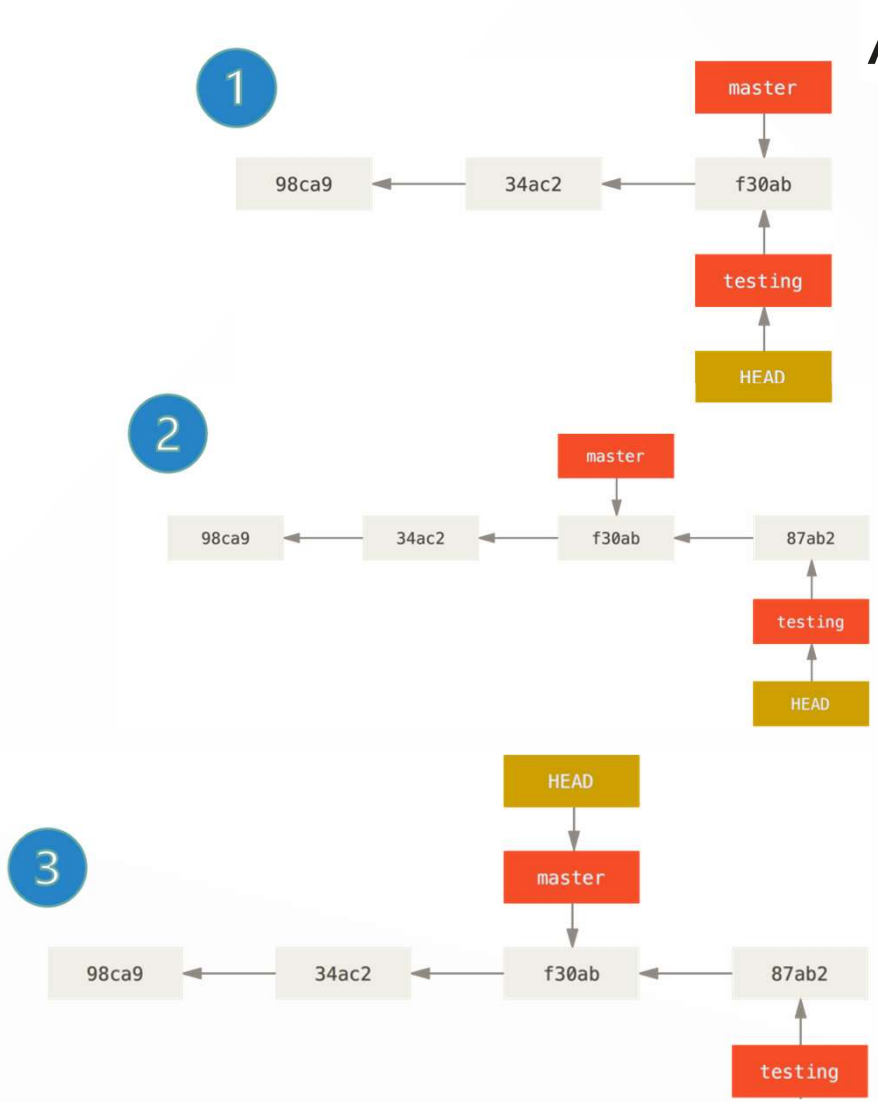
PAR L'EXEMPLE 3

Que se passe-t-il si vous créez une nouvelle branche ?

Eh bien, cela crée un nouveau pointeur pour vous. Supposons que vous créez une nouvelle branche nommée testing

PAR L'EXEMPLE 4

1. Si on bascule sur la nouvelle branche avec la commande git checkout testing
   - Cela déplace le HEAD pour le faire pointer vers la branche testing
2. Si j'effectue une modification
   - Notre branche testing a avancé tandis que la branche main pointe toujours sur le commit sur lequel vous étiez lorsque vous avez lancé la commande git checkout
3. Si je reviens sur ma branche main
   - Cette commande a remis le pointeur HEAD sur la branche main et a replacé les fichiers de votre répertoire de travail dans l'état du snapshot pointé par main.
   - Cela signifie aussi que les modifications que vous réalisez à partir de ce point divergeront de l'ancienne version du projet. Cette commande annule les modifications réalisées dans la branche testing pour vous permettre de repartir dans une autre direction.



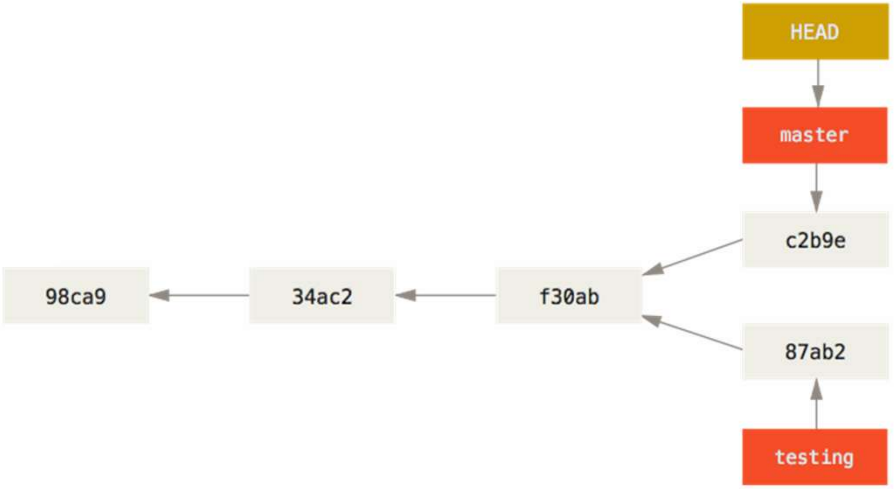
PAR L'EXEMPLE [5]

Si on réalise des modifications dans main et que nous validons à nouveau ?

Vous avez créé une branche et basculé dessus, y avez réalisé des modifications, puis vous avez rebasculé sur la branche principale et réalisé d'autres modifications.

Ces deux modifications sont isolées dans des branches séparées : vous pouvez basculer d'une branche à l'autre et les fusionner quand vous êtes prêt.

Et vous avez fait tout ceci avec de simples commandes : branch, checkout et commit.



BRANCHES ET FUSIONS (MERGES)

Prenons un exemple simple faisant intervenir des branches et des fusions (merges) que vous pourriez trouver dans le monde réel. Vous effectuez les tâches suivantes :

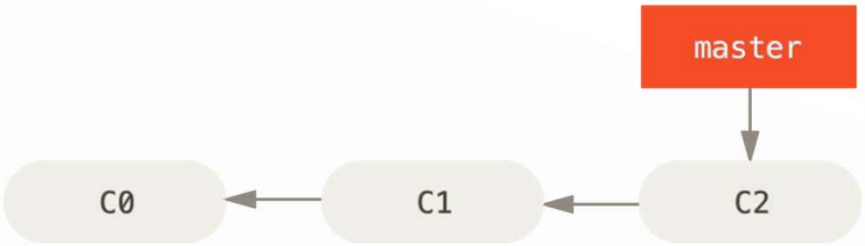
1. vous travaillez sur un site web
2. vous créez une branche pour un nouvel article en cours
3. vous commencez à travailler sur cette branche.

À cette étape, vous recevez un appel pour vous dire qu'un problème critique a été découvert et qu'il faut le régler au plus tôt. Vous faites donc ce qui suit :

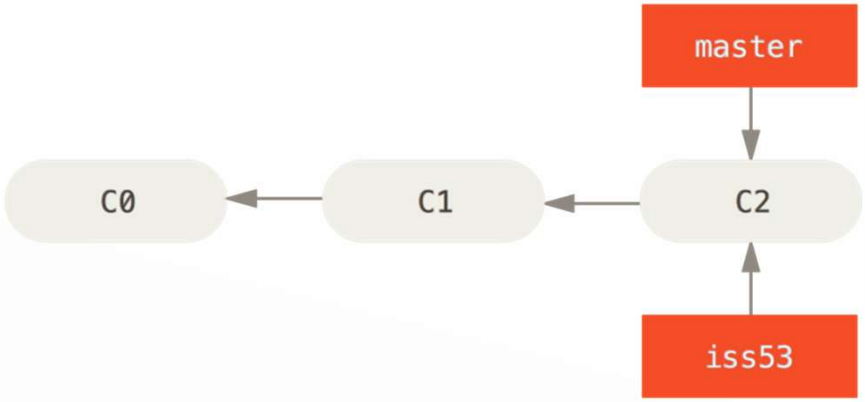
4. vous basculez sur la branche de production
5. vous créez une branche pour y ajouter le correctif
6. après l'avoir testé, vous fusionnez la branche du correctif et poussez le résultat en production
7. vous rebasculez sur la branche initiale et continuez votre travail.

BRANCHES ET FUSIONS (MERGES)

Etape 1(initial)

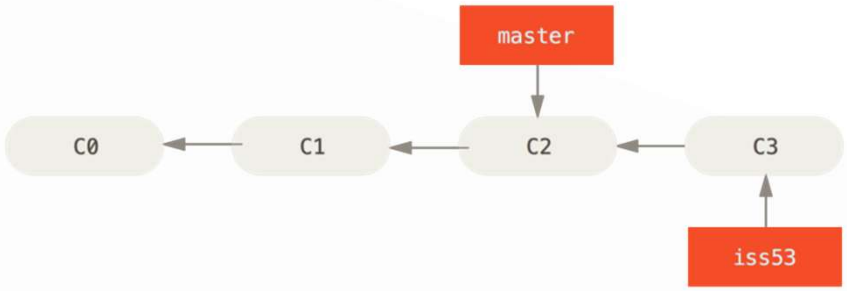


Etape 2 : Création d'une nouvelle branche



BRANCHES ET FUSIONS (MERGES)

Etape 3 : Travail en cours sur la branche



Le problème apparaît

Avec Git, vous n'avez pas à déployer en même temps votre correctif et les modifications déjà validées pour iss53 et vous n'avez pas non plus à vous fatiguer à annuler ces modifications avant de pouvoir appliquer votre correctif sur ce qu'il y a en production.

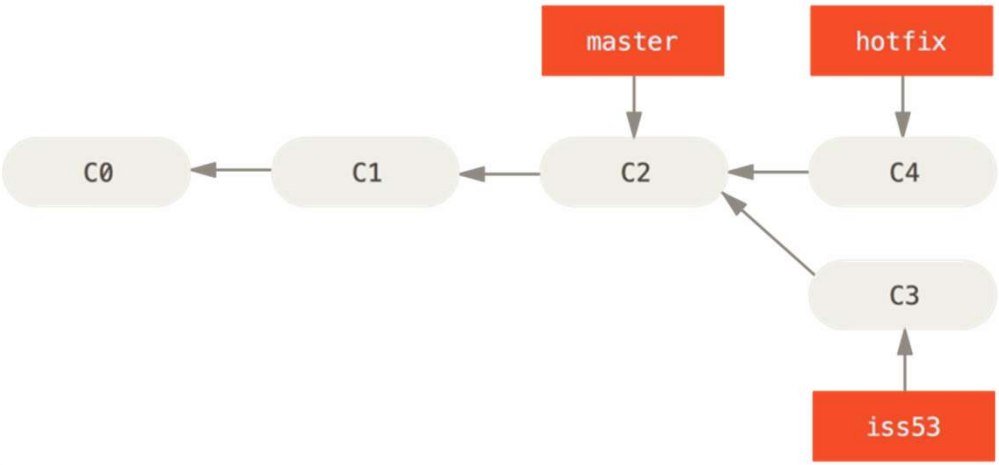
Tout ce que vous avez à faire, c'est simplement de rebasculer sur la branche main.

Attention : avant de le faire, notez que si votre copie de travail ou votre zone d'index contiennent des modifications non validées qui sont en conflit avec la branche que vous extrayez, Git ne vous laissera pas changer de branche. Le mieux est d'avoir votre copie de travail propre au moment de changer de branche.

ETAPE 4-5

À cet instant, votre répertoire de copie de travail est exactement dans l'état dans lequel vous l'aviez laissé avant de commencer à travailler sur le problème #53 et vous pouvez vous consacrer à votre correctif.

Création d'une branche pour le correctif

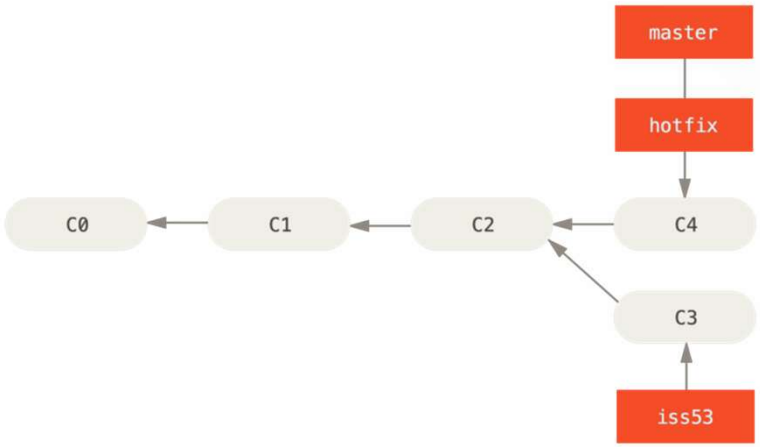


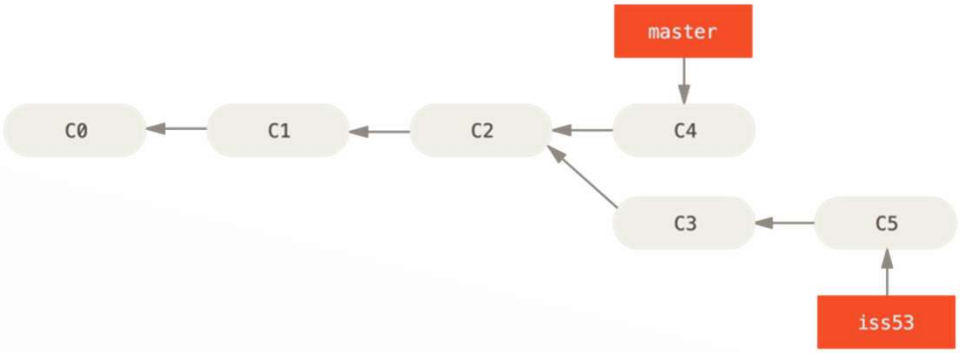
ETAPE 6

Vous pouvez lancer vos tests, vous assurer que la correction est efficace et la fusionner dans la branche main pour la déployer en production. Vous réalisez ceci au moyen de la commande git merge hotfix

Après le déploiement de votre correctif super-important, vous voilà prêt à retourner travailler sur le sujet qui vous occupait avant l'interruption.

Cependant, vous allez avant cela effacer la branche hotfix dont vous n'avez plus besoin puisque la branche main pointe au même endroit. Vous pouvez l'effacer avec l'option -d de la commande git branch : git branch -d hotfix

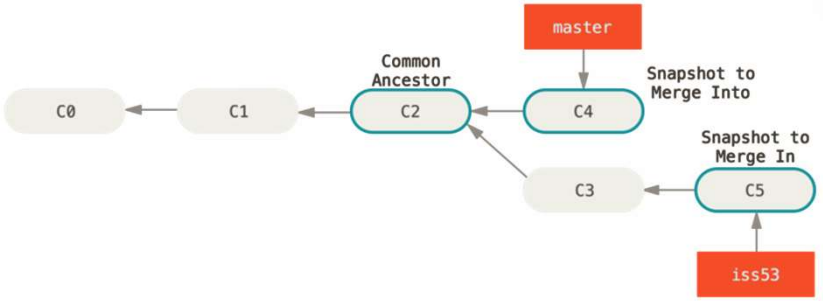


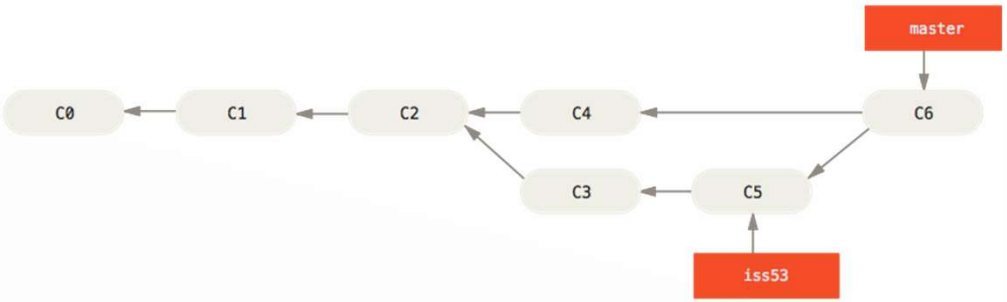


ETAPE 7

Supposons que vous ayez décidé que le travail sur le problème #53 était terminé et prêt à être fusionné dans la branche main

1. Bascule sur la branche principale : git checkout main
2. Fusion de la branche iss53 avec la branche principale : git merge iss53
3. Suppression de la branche iss53 : git branch -d iss53





CONFLITS DE FUSIONS (MERGE CONFLICTS)

Quelques fois, le processus ci-dessus ne se déroule pas aussi bien.

Si vous avez modifié différemment la même partie du même fichier dans les deux branches que vous souhaitez fusionner, Git ne sera pas capable de réaliser proprement la fusion.

Si votre résolution du problème #53 a modifié la même section de fichier que le hotfix.

Vous obtiendriez un conflit.

- Git arrête le processus le temps de résoudre le conflit. La commande git status vous indique où se trouve le conflit (listé avec unmerged dans le résultat de la commande).
- Cela marque dans le fichier les différences afin qu'on puisse manuellement effectuer le choix pour résoudre le conflit.

CONFLITS DE FUSIONS (MERGE CONFLICTS)

Après résolutions des conflits.

Il faut indexer les fichiers concernés avec la commande git add nom_du_fichier

Cela permet d'indiquer à git la résolution du conflit.

Finaliser la fusion

Enfin la commande git commit permet de finaliser la fusion

Un message de validation vous indique la réussite de la fusion.

GESTION DES BRANCHES

Git branch	
-v	Visualise la liste des derniers commits sur chaque branche
--merged	Filtre pour indiquer les branches fusionnées
--no-merged	Filtre pour indiquer les branches non fusionnées
--move old new	Modifie le nom d'une branche
--all	

BRANCHES DE SUIVI À DISTANCE

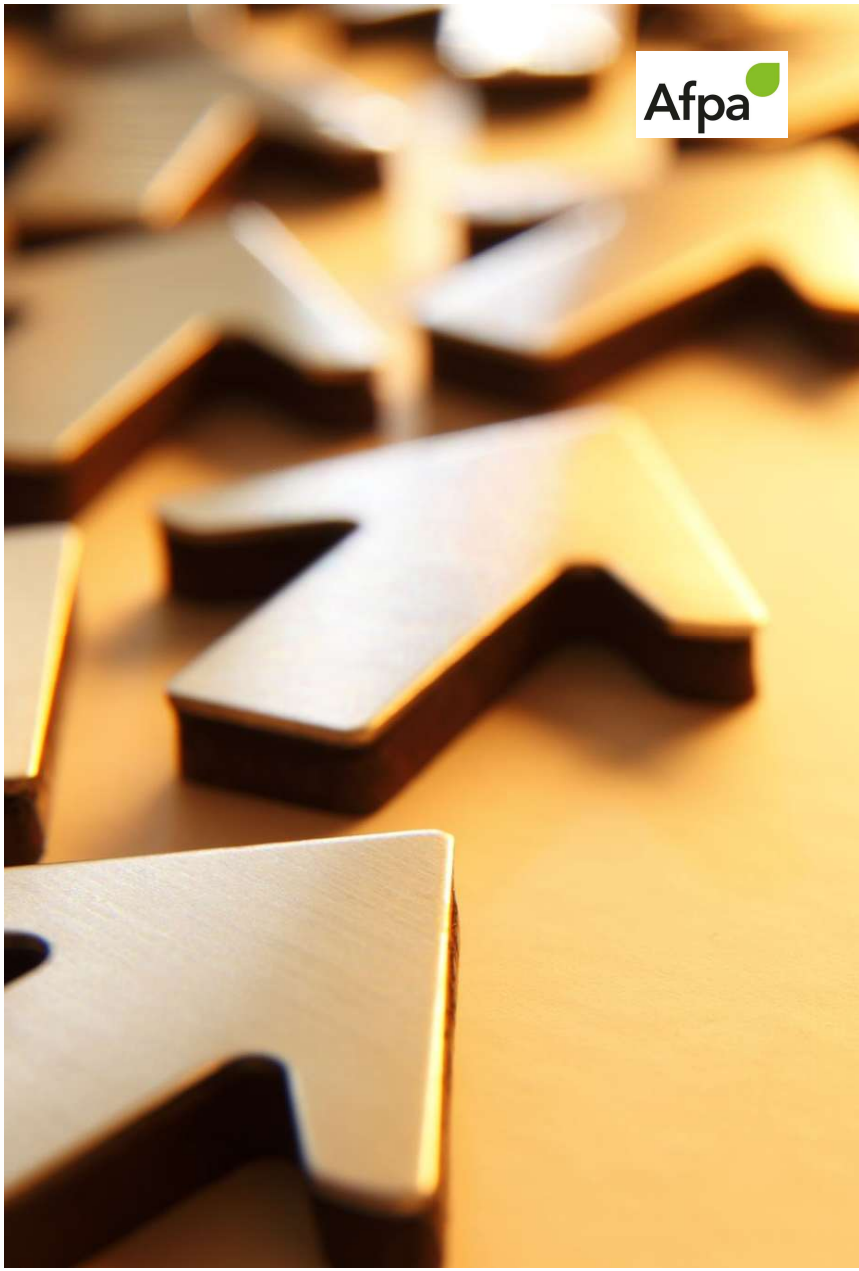
Les références distantes sont des références (pointeurs) vers les éléments de votre dépôt distant tels que les branches, les tags, etc…

Ce sont des branches locales qu'on ne peut pas modifier - elles sont modifiées automatiquement pour vous lors de communications réseau.

Les branches de suivi à distance agissent comme des marque-pages pour vous indiquer l'état des branches sur votre dépôt distant lors de votre dernière connexion.

Elles prennent la forme de (distant)/(branche).

Par exemple, si vous souhaitiez visualiser l'état de votre branche main sur le dépôt distant origin lors de votre dernière communication, il vous suffirait de vérifier la branche origin/main.



BRANCHES DE SUIVI À DISTANCE

Vous pouvez obtenir la liste complète de ces références distante avec la commande

- git ls-remote nom
- git remote show

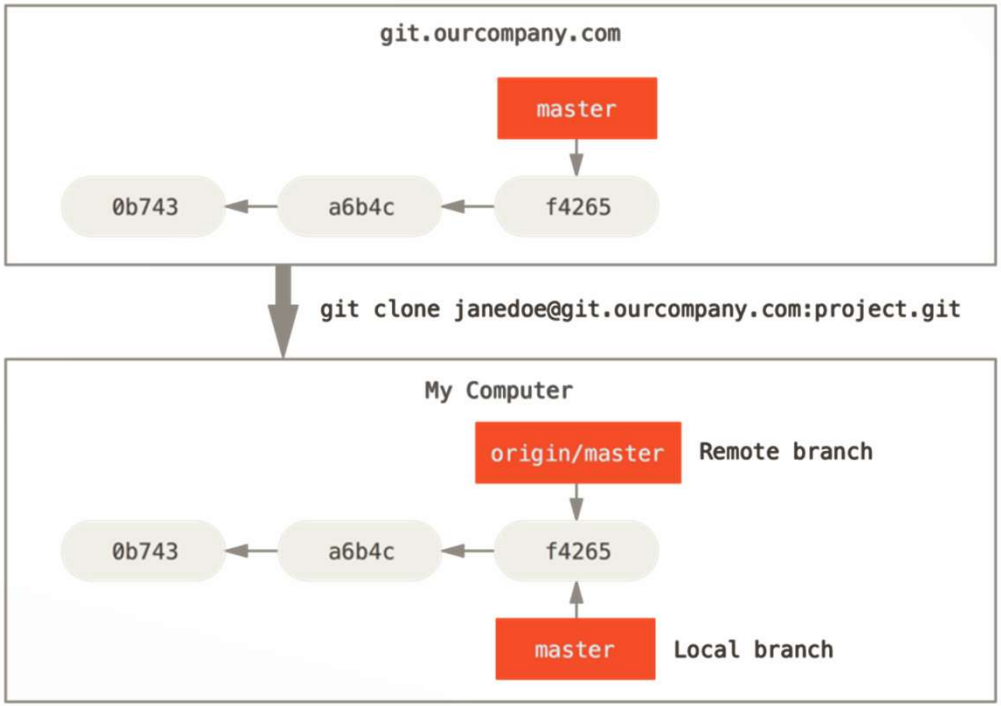
```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA/CoursDemo (main)
$ git ls-remote origin
c249f262ffb135175d2e025d171a8d89e203f191        HEAD
c249f262ffb135175d2e025d171a8d89e203f191        refs/heads/main

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA/CoursDemo (main)
$ git remote show
origin
```

ILLUSTRATION

Supposons que vous avez un serveur Git sur le réseau à l'adresse git.notresociete.com.

Si vous clonez à partir de ce serveur, la commande clone de Git le nomme automatiquement origin, tire tout son historique, crée un pointeur sur l'état actuel de la branche main et l'appelle localement origin/main.

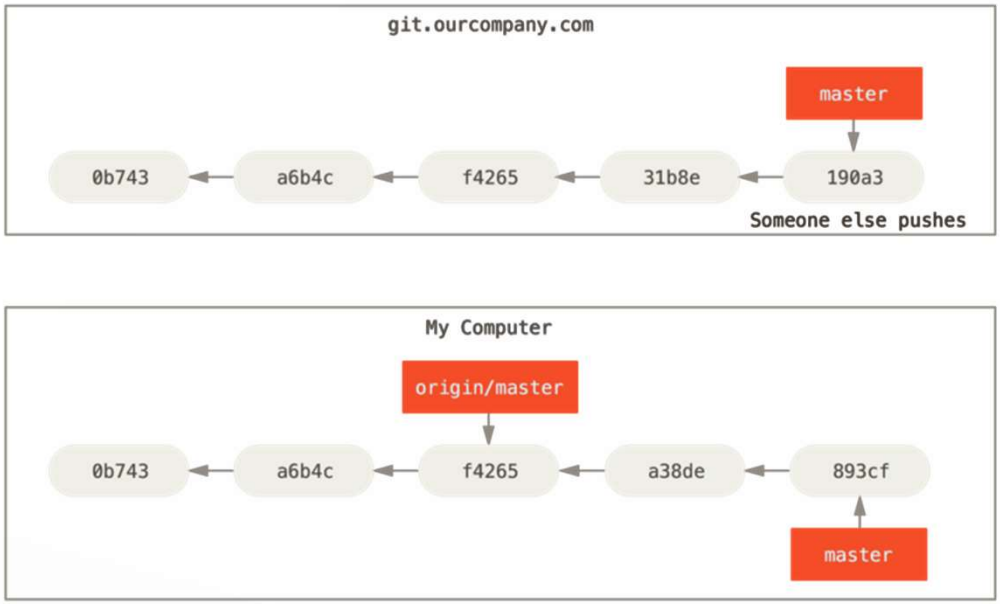
Git crée également votre propre branche main qui démarre au même endroit que la branche main d'origine, pour que vous puissiez commencer à travailler.



ILLUSTRATION

Si vous travaillez sur votre branche locale main et que dans le même temps, quelqu'un publie sur git.notresociete.com et met à jour cette même branche main, alors vos deux historiques divergent.

Tant que vous restez sans contact avec votre serveur distant, votre pointeur vers origin/main n'avance pas.

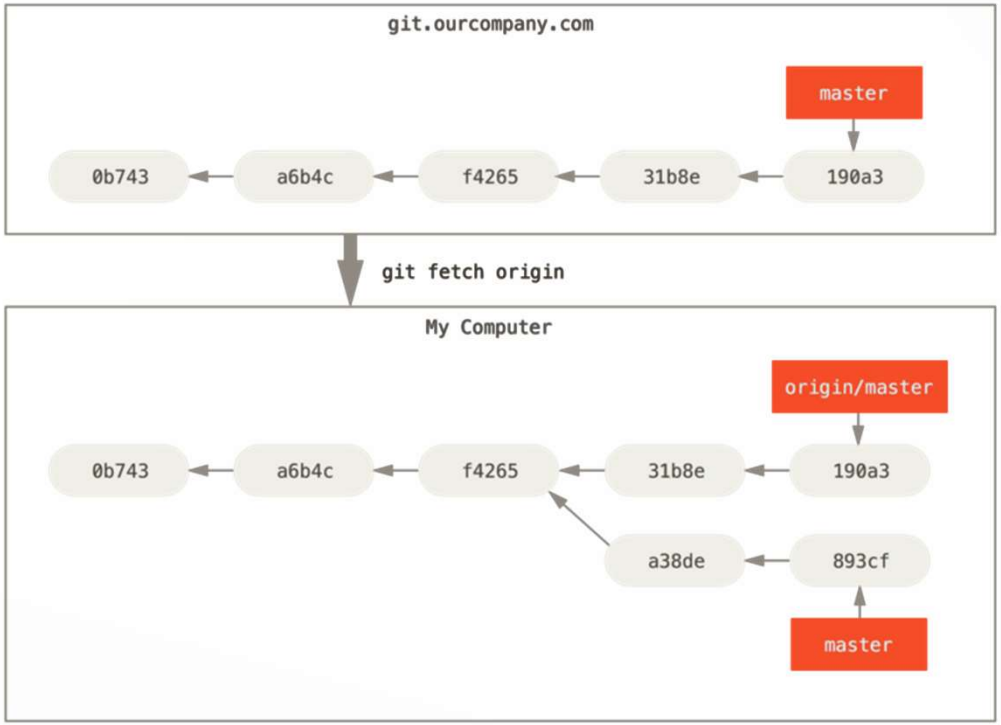


ILLUSTRATION

Lancez la commande git fetch <distant> pour synchroniser vos travaux (dans notre cas, git fetch origin).

Cette commande recherche le serveur hébergeant origin (dans notre cas, git.notresociete.com), y récupère toutes les nouvelles données et met à jour votre base de données locale en déplaçant votre pointeur origin/main vers une nouvelle position, plus à jour.

Git fetch met à jour vos branches de suivi à distance



ILLUSTRATION

Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur un serveur distant sur lequel vous avez accès en écriture.

Vos branches locales ne sont pas automatiquement synchronisées sur les serveurs distants — vous devez pousser explicitement les branches que vous souhaitez partager.

De cette manière, vous pouvez utiliser des branches privées pour le travail que vous ne souhaitez pas partager et ne pousser que les branches sur lesquelles vous souhaitez collaborer.

Si vous possédez une branche nommée correctionserveur sur laquelle vous souhaitez travailler avec d'autres, vous pouvez la pousser de la même manière que vous avez poussé votre première branche.

Lancez git push serveur_distant branche

Par exemple : git push origin correctionserveur

La prochaine fois qu'un de vos collègues récupère les données depuis le serveur (git fetch origin), il récupérera, au sein de la branche de suivi à distance origin/correctionserveur, une référence vers l'état de la branche correctionserveur sur le serveur.

Attention : Il est important de noter que lorsque vous récupérez une nouvelle branche depuis un serveur distant, vous ne créez pas automatiquement une copie locale éditable.

Pour fusionner ce travail dans votre branche de travail actuelle, vous pouvez lancer la commande :

- *git merge origin/correctionserveur*

Si vous souhaitez créer votre propre branche correctionserveur pour pouvoir y travailler, vous pouvez faire qu'elle repose sur le pointeur distant :

- *git checkout -b correctionserveur origin/correctionserveur*

SUIVI DES BRANCHES

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement ce qu'on appelle une "branche de suivi" (tracking branch) et la branche qu'elle suit est appelée "branche amont" (upstream branch).

Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante.

- Si vous vous trouvez sur une branche de suivi et que vous tapez git push, Git sélectionne automatiquement le serveur vers lequel pousser vos modifications.
  - De même, un git pull sur une de ces branches récupère toutes les références distantes et fusionne automatiquement la branche distante correspondante dans la branche actuelle.
- Lorsque vous clonez un dépôt, il crée généralement automatiquement une branche main qui suit origin/main. C'est pourquoi les commandes git push et git pull fonctionnent directement sans autre configuration.

SUIVI DES BRANCHES

Tirer une branche (Pulling)

git fetch récupère l'ensemble des changements présents sur serveur et qui n'ont pas déjà été rapatriés localement, elle ne modifie en rien votre répertoire de travail. elle récupère simplement les données pour vous et vous laisse les fusionner par vous-même.

Cependant, il existe une commande appelée git pull qui consiste essentiellement en un git fetch immédiatement suivi par un git merge dans la plupart des cas.

git pull va examiner quel serveur et quelle branche votre branche courante suit actuellement, synchroniser depuis ce serveur et ensuite essayer de fusionner cette branche distante avec la vôtre.

Il est généralement préférable de simplement utiliser les commandes fetch et merge explicitement plutôt pull qui peut s'avérer source de confusion.

Suppression de branches distantes

Supposons que vous en avez terminé avec une branche distante et l'avez fusionnée dans la branche main du serveur distant (ou la branche correspondant à votre code stable).

Vous pouvez effacer une branche distante en ajoutant l'option --delete à git push.

Si vous souhaitez effacer votre branche :

git push origin --delete nom_branche

Afpa

COMMITS ATOMIQUES
BONNES PRATIQUES

COMMITS ATOMIQUES

Une bonne pratique avec Git consiste à décomposer les modifications à enregistrer dans le dépôt en entités atomiques (ou unitaires).

Cette pratique permet de mieux suivre, et donc comprendre, les modifications apportées.

Elle permet également de plus facilement retrouver une modification, et de la renverser sans toucher au reste des modifications (quand c'est possible...).

Cependant si, par exemple, on ajoute une nouvelle fonction dans un code source, il faut pouvoir s'assurer du bon fonctionnement de cette fonction avant de déposer les modifications.

On va donc utiliser cette fonction par ailleurs dans nos sources, compiler le projet et vérifier que le résultat est satisfaisant.

Nous voulons maintenant séparer nos modifications en 2 commits :

1. le premier contiendra la définition de la nouvelle fonction,
2. le second contiendra l'utilisation de cette fonction.

Si ces modifications sont dans 2 fichiers séparés, alors vous savez déjà comment procéder.

Mais comment faire si tout est dans le même fichier ?

AJOUT INTERACTIF

L'option -p (--patch) de la commande git add permet de sélectionner interactivement les morceaux de code que nous voulons ajouter à l'index.

Avec cette option, git add va découper les modifications en morceaux, appelés hunks, qui seront affichés l'un après l'autre, et vous demander ce que vous voulez faire de chacun de ces hunks.

Stage this hunk [y,n,a,d,/,j,J,g,e,?]?

y - stage this hunk

n - do not stage this hunk

a - stage this and all the remaining hunks in the file

d - do not stage this hunk nor any of the remaining hunks in the file

g - select a hunk to go to

/ - search for a hunk matching the given regex

j - leave this hunk undecided, see next undecided hunk

J - leave this hunk undecided, see next hunk

k - leave this hunk undecided, see previous undecided hunk

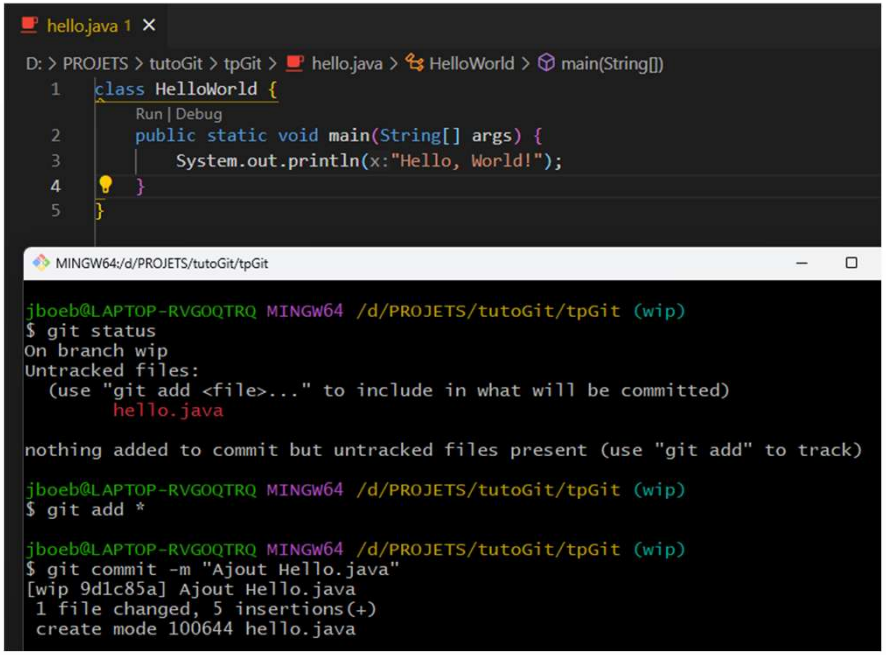
K - leave this hunk undecided, see previous hunk

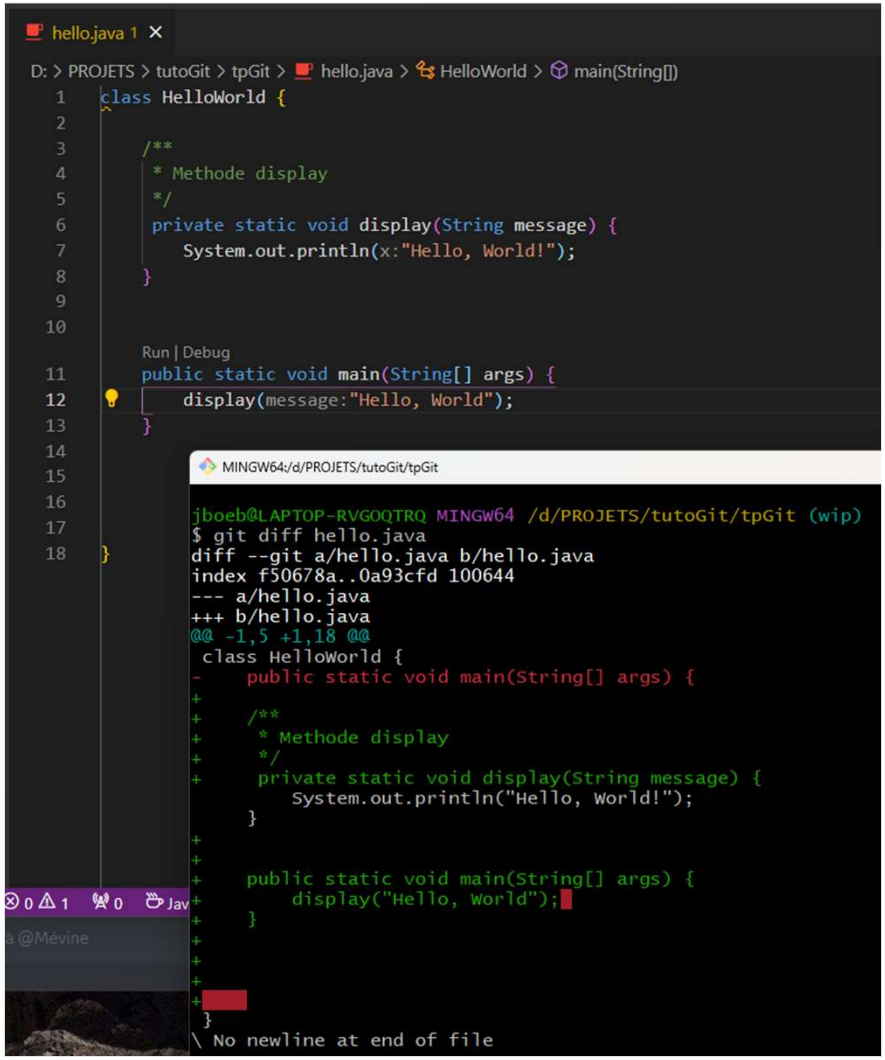
s - split the current hunk into smaller hunks

e - manually edit the current hunk

? - print help

EXEMPLE


hello.java 1
D: > PROJETS > tutoGit > tpGit > hello.java > HelloWorld > main(String[])
class HelloWorld {
Run | Debug
public static void main(String[] args) {
System.out.println(x:"Hello, World!");
}
}
MINGW64:/d/PROJETS/tutoGit/tpGit
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git status
On branch wip
Untracked files:
(use "git add <file>..." to include in what will be committed)
hello.java
nothing added to commit but untracked files present (use "git add" to track)
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git add *
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git commit -m "Ajout Hello.java"
[wip 9d1c85a] Ajout Hello.java
1 file changed, 5 insertions(+)
create mode 100644 hello.java


hello.java 1
D: > PROJETS > tutoGit > tpGit > hello.java > HelloWorld > main(String[])
class HelloWorld {
/**
* Methode display
*/
private static void display(String message) {
System.out.println(x:"Hello, World!");
}
Run | Debug
public static void main(String[] args) {
display(message:"Hello, World");
}
MINGW64:/d/PROJETS/tutoGit/tpGit
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git diff hello.java
diff --git a/hello.java b/hello.java
index f50678a..0a93cfd 100644
--- a/hello.java
+++ b/hello.java
@@ -1,5 +1,18 @@
class HelloWorld {
- public static void main(String[] args) {
+
+ /**
+ * Methode display
+ */
+ private static void display(String message) {
+ System.out.println("Hello, World!");
+ }
+
+
+ public static void main(String[] args) {
+ display("Hello, World");
+ }
+
+
+
+ }
\ No newline at end of file

Dans notre cas, les 2 modifications étant proches l'une de l'autre, git add -p les place dans le même hunk.

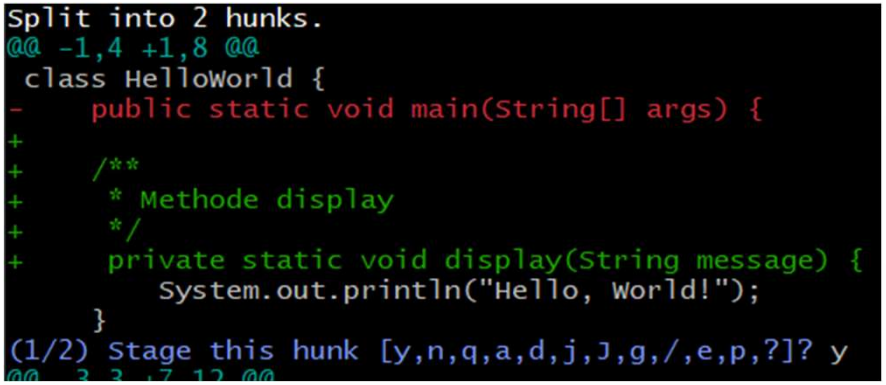
```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git add -p hello.java
diff --git a/hello.java b/hello.java
index f50678a..0a93cfd 100644
--- a/hello.java
+++ b/hello.java
@@ -1,5 +1,18 @@
 class HelloWorld {
-    public static void main(String[] args) {
+
+    /**
+     * Methode display
+     */
+     private static void display(String message) {
+        System.out.println("Hello, World!");
+    }
+
+
+    public static void main(String[] args) {
+        display("Hello, World");
+    }
+
+
+
 }
\ No newline at end of file
(1/1) Stage this hunk [y,n,q,a,d,s,e,p,?]?
```

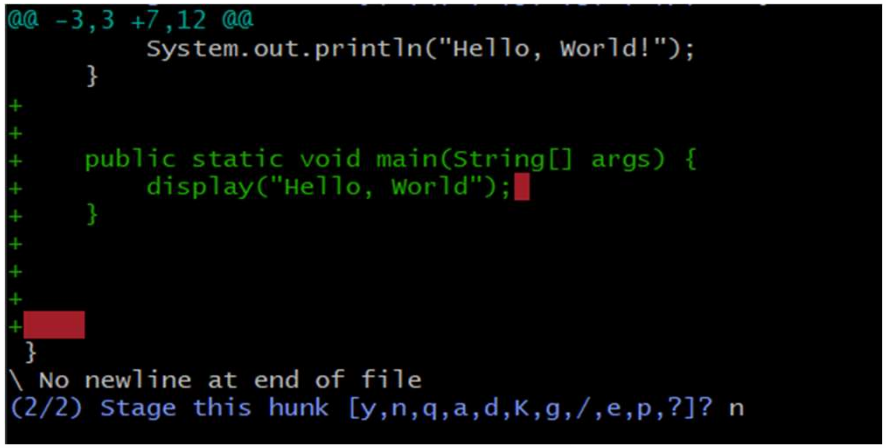
? [Pour obtenir de l'aide]

```
(1/1) Stage this hunk [y,n,q,a,d,s,e,p,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
s - split the current hunk into smaller hunks
e - manually edit the current hunk
p - print the current hunk
? - print help
(1/1) Stage this hunk [y,n,q,a,d,s,e,p,?]?
```

Il est donc possible de découper le morceau actuel en 2 hunks plus petits, à l'aide de la commande s.

Cette modification doit être incluse dans le 1er commit, vous répondez donc par y. Le hunk qui est affiché ensuite ne doit pas être inclus, vous répondez par n.





AVEC LES COMMANDES GIT STATUS, GIT DIFF --STAGED ET GIT DIFF, LE CODE CORRESPONDANT À L'AJOUT DE LA FONCTION EST BIEN DANS L'INDEX, PRÊT À ÊTRE VALIDÉ.

On a un fichier staged avec le 1er hunk prêt à être commiter

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git status
On branch wip
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   hello.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.java
```

On a un fichier non stagged avec le 2eme hunk en attente

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git diff --staged
diff --git a/hello.java b/hello.java
index f50678a..af6969d 100644
--- a/hello.java
+++ b/hello.java
@@ -1,5 +1,9 @@
 class HelloWorld {
-    public static void main(String[] args) {
+
+    /**
+     * Methode display
+     */
+    private static void display(String message) {
         System.out.println("Hello, World!");
     }
 }
\ No newline at end of file
```

MAIS QUE LE CODE D'UTILISATION DE CETTE FONCTION EST TOUJOURS DANS LA COPIE DE TRAVAIL, NON ENCORE INDEXÉ.

```
$ git diff
diff --git a/hello.java b/hello.java
index af6969d..0a93cfd 100644
--- a/hello.java
+++ b/hello.java
@@ -6,4 +6,13 @@ class HelloWorld {
      private static void display(String message) {
         System.out.println("Hello, World!");
      }
+
+
+     public static void main(String[] args) {
+         display("Hello, World");
+     }
+
+
+
+
 }
\ No newline at end of file
```

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git commit -m "Ajout methode display"
[wip e5840c1] Ajout methode display
 1 file changed, 5 insertions(+), 1 deletion(-)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git status
On branch wip
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.java

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git add hello.java

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git commit -m "utilisation methode display"
[wip aa179db] utilisation methode display
 1 file changed, 9 insertions(+)
```

COMMITS ATOMIQUES

Désindexation interactive

Parfois, on se rend compte après coup qu'une modification a été ajoutée à l'index par erreur.

Il est également possible de choisir les morceaux de code devant être désindexés avec git reset -p (qui est donc la commande opposée à git add -p).

Pour cela, la commande git reset -p, qui s'utilise comme git add -p. Il faudra donc découper le hunk en 2, et désindexer le hunk concerné.

Annulation interactive

Que ce soit lors de l'édition d'un texte ou du développement d'un code, il n'est pas rare que l'on désire abandonner les modifications que l'on vient de faire pour revenir à la version précédente.

la commande git checkout -p, qui s'utilise comme git add -p et git reset -p.

Il faudra donc découper en 2 le hunk proposé, accepter d'annuler les modifications correspondant au 1er morceau, et refuser pour le 2eme.

GUI POUR GIT

GITKRAKEN

GITKRAKEN

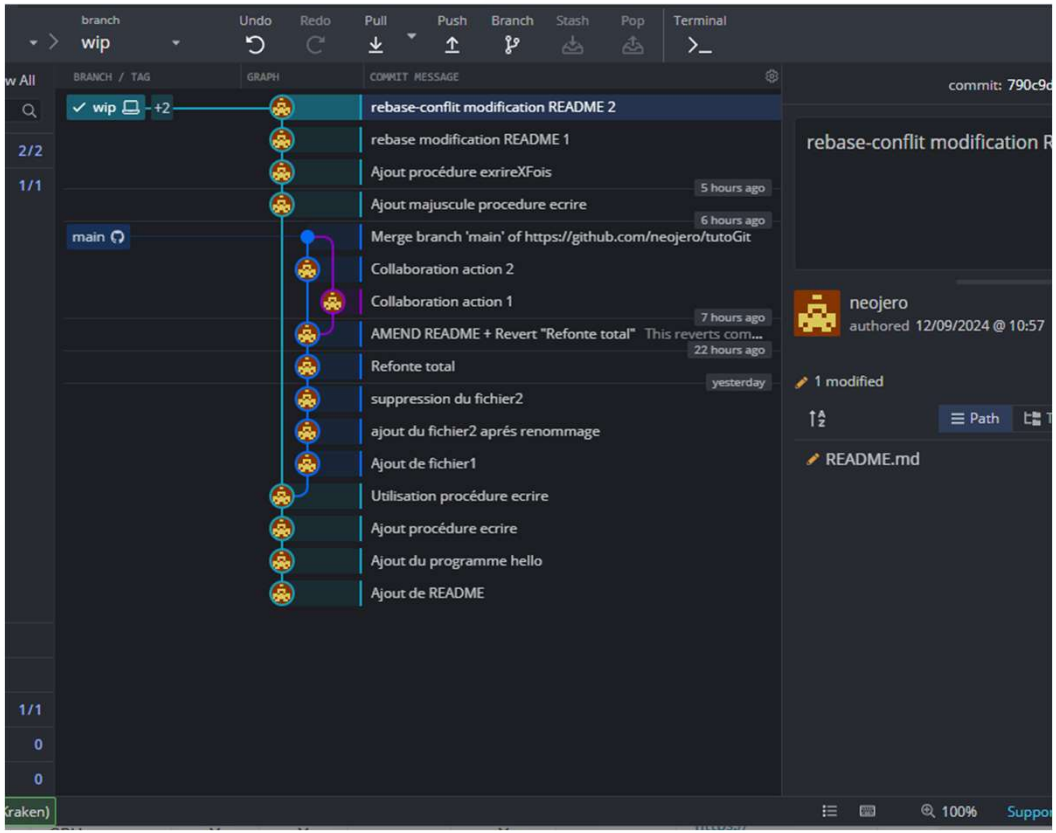
Vous avez pu constater que l'affichage de l'historique n'est pas optimal en ligne de commande.

Des outils nous permettent de gagner en efficacité.

- Je vais donc vous en présenter un de ces clients baptisé GitKraken. Sorti en 2016, ce client n'a eu de cesse d'évoluer et propose gratuitement tout le nécessaire pour les développeurs.
- On peut visuellement inspecter l'historique de tous ses commits et des branches du projet.
- Vous pouvez également faire du drag et drop ou rechercher dans les fichiers de vos dépôts.
- Si vous faites une boulette, pas de panique ! GitKraken propose une fonctionnalité « Undo », c'est-à-dire d'annulation de la dernière commande.
- Il intègre un éditeur pour fusionner vos fichiers en cas de conflits ainsi qu'un petit éditeur de code avec coloration syntaxique, et la possibilité de mettre 2 versions côte à côte pour inspecter les différences (diff).

GITKRAKEN

Voici une capture de mon dépôt à la fin du TP visualisé avec l'outil GitKraken.



LIENS UTILES

Git : https://git-scm.com/

Git cheatsheet : http://ndpsoftware.com/git-cheatsheet.html#loc=index

GitLab : https://about.gitlab.com/

GitHub : https://github.com

Simulation Git : https://learngitbranching.js.org/?locale=fr_FR

Git Cheat Sheet

Remember!
`git <COMMAND> --help`

Global configuration is stored in ~/.gitconfig.
`git config --help`

master is the default development branch.
origin is the default upstream repository.

Create

From existing data
```
cd ~/my_project_directory
git init
git add .
```

From existing repository
```
git clone ~/existing_repo ~/new/repo
git clone git://host.org/project.git
git clone ssh://user@host.org/project.git
```

Show

Files changed in working directory
`git status`

Changes made to tracked files
`git diff`

What changed between ID1 and ID2
`git diff <ID1> <ID2>`

History of changes
`git log`

History of changes for file with diffs
`git log -p <FILE> <DIRECTORY>`

Who changed what and when in a file
`git blame <FILE>`

A commit identified by ID
`git show <ID>`

A specific file from a specific ID
`git show <ID>:<FILE>`

All local branches
`git branch`
star (*) marks the current branch

Revert

Return to the last commited state
`git reset --hard`
This cannot be undone!

Revert the last commit
`git revert HEAD`
Creates a new commit

Revert specific commit
`git revert <ID>`
Creates a new commit

Fix the last commit
`git commit -a --amend`
(after editing the broken files)

Checkout the ID version of a file
`git checkout <ID> <FILE>`

Update

Fetch latest changes from origin
`git fetch`
(this does not merge them)

Pull latest changes from origin
`git pull`
(does a fetch followed by a merge)

Apply a patch that someone sent you
`git am -3 patch.mbox`
In case of conflict, resolve the conflict and
`git am --resolved`

Publish

Commit all your local changes
`git commit -a`

Prepare a patch for other developers
`git format-patch origin`

Push changes to origin
`git push`

Make a version or milestone
`git tag v1.0`

Branch

Switch to a branch
`git checkout <BRANCH>`

Merge BRANCH1 into BRANCH2
```
git checkout <BRANCH2>
git merge <BRANCH1>
```

Create branch BRANCH based on HEAD
`git branch <BRANCH>`

Create branch BRANCH based on OTHER and switch to it
`git checkout -b <BRANCH> <OTHER>`

Delete branch BRANCH
`git branche -d <BRANCH>`

Resolve merge conflicts

View merge conflicts
`git diff`

View merge conflicts against base file
`git diff --base <FILE>`

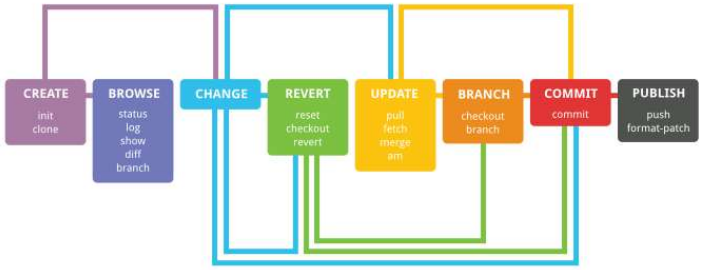
View merge conflicts against your changes
`git diff --ours <FILE>`

View merge conflicts against other changes
`git diff --theirs <FILE>`

Discard a conflicting patch
```
git reset --hard
git rebase --skip
```

After resolving conflicts, merge with
```
git add <CONFLICTING_FILE>
git rebase --continue
```

Workflow



2022

MERCI !

Jérôme BOEBION
Concepteur Développeur d'Applications
Version 1 - révision 2024

Afpa
se former, avancer