Fms Academy 2022
Formation Java Spring Angular
Module 4 : Git & GitHub

mohamed.el-babili@fms-ea.com

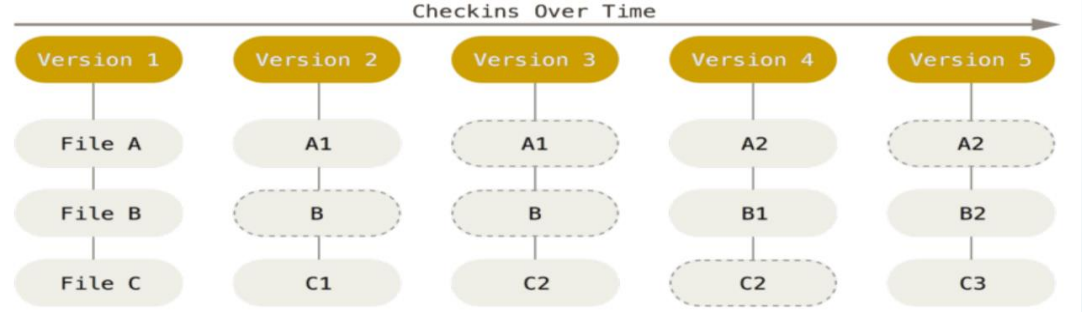
Sommaire

Git Intro

- Git est un gestionnaire de version qui enregistre l'évolution d'un fichier ou ensemble de fichiers au cours du temps, de sorte qu'on puisse rappeler une version antérieure d'un fichier ou d'un projet à tout moment.
- C'est un système de gestion de version (VCS pour Version Control System) permettant aussi de visualiser les changements au cours du temps, de voir qui a modifié qq chose qui pourrait causer pb et depuis quand.
- **Qui n'a jamais fait de bêtises et aimerait revenir en arrière pour retrouver du code ou des fichiers perdus par ex ?**
- Le meilleur moyen de comprendre l'intérêt de Git est encore de ne pas l'utiliser dans un projet.
- Que dire alors si nous sommes plusieurs développeurs à travailler sur le même projet ?
- Dans cette approche, les devs dupliquent complètement le dépôt, du coup si le serveur disparaissait, on pourrait le restaurer avec la copie d'un dev qui devient comme une sauvegarde.

Comment fonctionne Git (1)

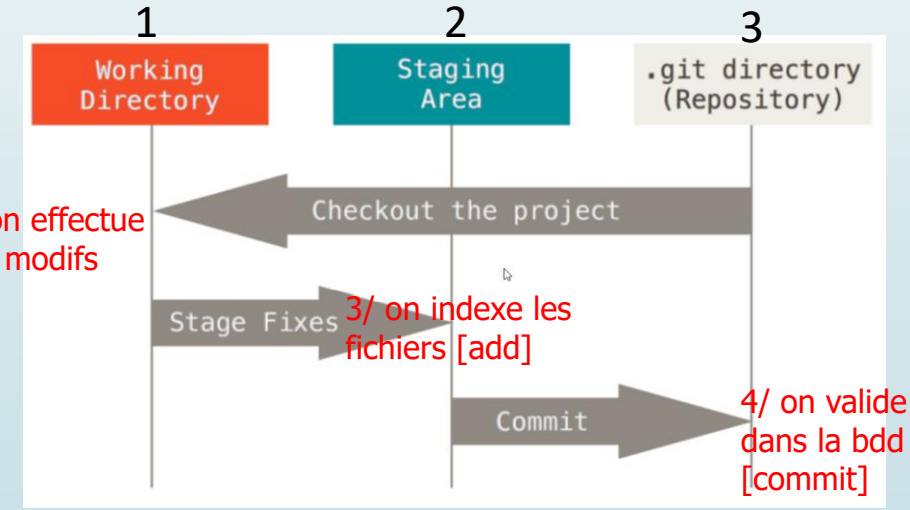
- Git gère ses données à la manière d'un flux instantané (photocopie)



- Les atouts de Git viennent de la possibilité offerte de travailler en locale sur la version de votre choix sans besoin de connexion, vous avez accès à l'historique du projet, vous pouvez comparer l'évolution d'un fichier sur une période donnée sans passer par un serveur…
- Puis une fois connecté, vous pourrez partager votre travail
- Git gère l'intégrité des données, aucune perte n'est possible lors d'un transfert par ex car il scrute la moindre de vos actions via une somme de contrôle.
- Le mécanisme qu'il utilise pour réaliser les contrôles est appelé une emprunte SHA-1 calculée en fonction du contenu du fichier ou de la structure du répertoire considéré.
- Git gère et stocke dans sa base de données, indexée par ses clés `24b9da6552252987aa493b52f8696cd6d3b00373`

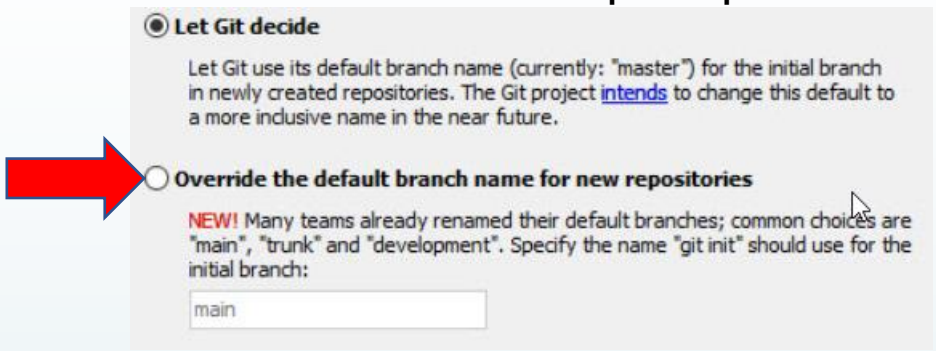
Comment fonctionne Git (2)

- Les 3 états d'un fichier : Modifié, Indexé et Validé
  - Modifié pour fichier modifié uniquement
  - Indexé pour fichier modifié et marqué prêt pour photocopie [add]
  - Validé pour fichier stocké en base de données locale [commit]
- Les 3 sections principales d'un projet Git :
  - **1/ Le répertoire de travail** est une extraction unique d'une version du projet pour utilisation et modification.
  - **2/ La zone d'index** est un simple fichier (rep git) qui stocke les infos pour la prochaine photocopie.
  - **3/ Le répertoire Git** est l'endroit ou Git stocke les méta données et la base de données des objets de votre projet (cloné ou pas)



Install & config [Windows]

- Télécharger et installer ici (64 bits) avec options par défaut sauf pour éditeur VSC + choix de la branche principale : main



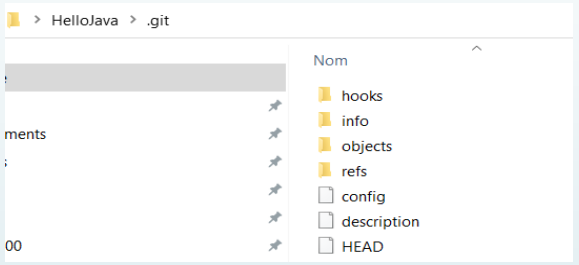
- Vérifier que Git est installé :
  - $ **git version**
- Sous windows, un fichier « C:\Users\you\.gitconfig » contient les infos de configuration. Il faut changer de user :
  - $ **git config** --global user.name "John Doe"
  - $ **git config** --global user.email johndoe@example.com
- Vérifier si c'est bon :
  - $ git config name
  - $ git config email

- $ **git help** → **affiche la liste des commandes**
- **$ git add -h → affiche les options de cette commande**

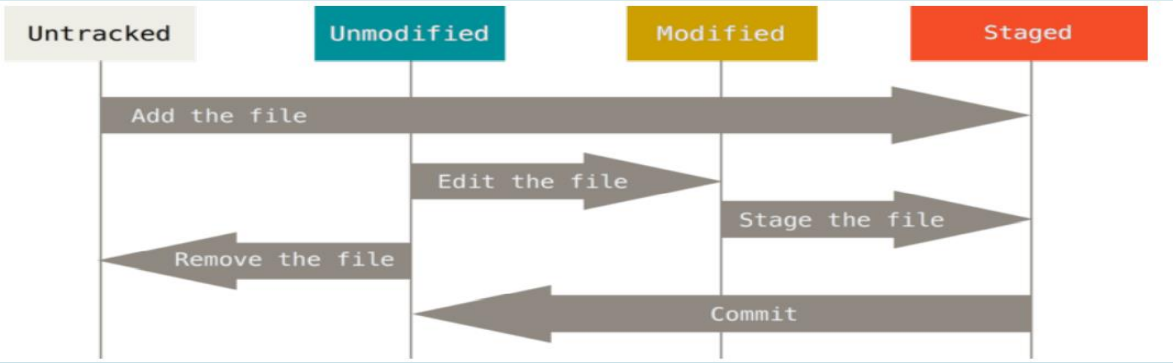
Git : Les principales commandes (1)

- 2 manières de démarrer un dépôt git :
  - Transformer un répertoire existant en dépôt Git
  - Cloner un dépôt existant sur un autre serveur
- Reprenons le 1er cas de figure en ajoutant un répertoire « HelloJava ». Positionnez vous dans le répertoire et exécuter la commande d'initialisation :

```
C:\Users\El-BabiliM\Desktop\HelloJava>git init
Initialized empty Git repository in C:/Users/El-BabiliM/Desktop/HelloJava/.git/
```



- $ **git init** → crée un sous répertoire .git qui contient tous Les fichiers nécessaires au dépôt.
- Ajouter une classe dans votre répertoire avec du code puis compiler et exécuter
- $ **git add** Hello.java → indique qu'on souhaite **suivre** le fichier Hello.java
- $ **git commit** -m « first commit » → indique qu'on a fait une photocopie poussée en bdd de git



Cycle de vie des états d'un fichier

Git : Les principales commandes (2)

- $ **git status** → pour vérifier l'état des fichiers

```
C:\Users\El-BabiliM\Desktop\HelloJava>git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Hello.class

nothing added to commit but untracked files present (use "git add" to track)

C:\Users\El-BabiliM\Desktop\HelloJava>git add Hello.class

C:\Users\El-BabiliM\Desktop\HelloJava>git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   Hello.class


C:\Users\El-BabiliM\Desktop\HelloJava>git commit -m "second commit"
[main a58c634] second commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Hello.class

C:\Users\El-BabiliM\Desktop\HelloJava>git status
On branch main
nothing to commit, working tree clean
```

Nb : ça n'a aucun intérêt de tracker Hello.class !

- $ **git restore** --staged Hello.class → le fichier n'est plus suivi

```
C:\Users\El-BabiliM\Desktop\HelloJava>git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    Hello.class

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Hello.class
```

- $ **git log** → pour visualiser l'historique des commits

```
C:\Users\El-BabiliM\Desktop\HelloJava>git log
commit a58c6345efc058070ae87735735affda3bf38d6b (HEAD -> main)
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Fri Jan 28 11:33:48 2022 +0100

    second commit

commit e525ff46f21b8d2232deb9ab850a26e0bbfaefc1
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Fri Jan 28 11:29:54 2022 +0100

    first commit

C:\Users\El-BabiliM\Desktop\HelloJava>git reset e525ff46f21b8d2232deb9ab850a26e0bbfaefc1

C:\Users\El-BabiliM\Desktop\HelloJava>git log
commit e525ff46f21b8d2232deb9ab850a26e0bbfaefc1 (HEAD -> main)
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Fri Jan 28 11:29:54 2022 +0100

    first commit
```

On peut supprimer un commit de 2 façons :
- **Git revert** (garde l'historique)
- **Git reset** (supprime toute trace de commits à la suite)

Git : Les principales commandes (3)

- Nous avons effectué des modifications sur un fichier

```
C:\Users\El-BabiliM\Desktop\HelloJava>git status
On branch main
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
```

- 2 options ici :
  - $ git restore Hello.java → pour supprimer les modifications que nous venons de faire
    - → les modifications seront définitivement perdues (avec Git on peut quasiment tout récupérer sauf ici)
  - **$** git add + git commit → Afin que ces modifications soient suivies et insérer en base
- Git add permet d'ajouter du contenu pour la prochaine validation aussi au cas ou cette commande est appelé une 1ère fois puis suivie de modification :
  - On nous indique qu'il faut faire un commit
  - Et aussi faire git add + git commit

```
C:\Users\El-BabiliM\Desktop\HelloJava>git status
On branch main
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
```

- → Lancer à nouveau git add avant de commit
- **La règle est à chaque modification qu'on effectue, faire suivre : Git add + git commit**
- Git add . Permet d'indexer tous les fichiers du répertoire

Git : Les principales commandes (4)

- Nous avons la possibilité **d'ignorer des fichiers** comme par exemple Hello.class, pour cela, il faut ajouter un fichier .**gitignore** à la racine du répertoire de travail, il contient des patrons standards de fichiers ou expressions régulières simplifiées comme ci-dessous :

```
# pas de byte code

*.class
```

- Si on souhaite **connaître ce qui a changé dans les fichiers**, il y a la commande **git diff** qui permet de savoir ce qui a été modifié sans être indexé (avant git add)
- Si vous souhaitez **visualiser les modifications indexées** (après git add) : git diff --staged
  - → Compare les fichiers indexés à la dernière photocopie
- Après avoir indexé un fichier, vous pouvez lancer un commit comme suit : **git commit** qui a pour effet de lancer l'éditeur par défaut en vous invitant à saisir un message de commit.
  - → git commit –v fait de même en affichant les modifications
- Commande qui permet de **cumuler git add et git commit** uniquement si le fichier est déjà indexé (git add) : **git commit –a –m « double action » ou -am**
- Pour **supprimer un fichier indexé**, il faut le supprimer du répertoire puis : **git rm « fichier »**
- **Git log** permet **d'afficher l'historique des commits** avec à chaque commit une série d'info :
  - **Email de l'auteur**
  - **Date**
  - **Message de commit**

```
C:\Users\El-BabiliM\Desktop\HelloJava>git log
commit f3a8b29fcb20c13a43b80182ca34e9e9a86706f8 (HEAD -> main)
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Wed Feb 2 16:43:25 2022 +0100

    ajout methode addition

commit 6b99376d422cb8a5d6c25ff924a0efaf049478bb
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Wed Feb 2 16:42:07 2022 +0100

    gestion des opérations

commit 3040ff372badcf5b68039405c03ab7ce7b61fdd1
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Wed Feb 2 16:37:09 2022 +0100

    first commit
```

Git : Les principales commandes (5)

- **Git log –p** montre les différences introduites entre chaque validation -2 limite la sortie aux 2 entrées les + récentes
- **Git log –-stat** affiche des states pour chaque fichiers modifiés
- **Git log –-pretty=oneline** affiche le journal des logs différemment

```
C:\Users\El-BabiliM\Desktop\HelloJava>git log --pretty=oneline
f3a8b29fcb20c13a43b80182ca34e9e9a86706f8 (HEAD -> main) ajout methode addition
6b99376d422cb8a5d6c25ff924a0efaf049478bb gestion des opérations
3040ff372badcf5b68039405c03ab7ce7b61fdd1 first commit
```

- **Git log –-pretty=format:** "%h - %an, %ar : %s"
  - Permet de décrire précisément le format de sortie

```
C:\Users\El-BabiliM\Desktop\HelloJava>Git log --pretty=format:"%h - %an, %ar : %s"
f3a8b29 - elbabili, 48 minutes ago : ajout methode addition
6b99376 - elbabili, 49 minutes ago : gestion des opérations
3040ff3 - elbabili, 54 minutes ago : first commit
```

- **Multiples combinaisons** : tous les commits de elbabili entre le 2 et le 3 février 2022, pas de commit de fusion
- git log --pretty="%h - %s" --author=`elbabili' --since="2022-02-02" \ --before=" 2022-02-03" --no-merges -- t/

  - **COMMENT COMPLETER DES ACTIONS PROPREMENT LOCALEMENT ?**

- Après un commit, on se rend compte qu'un fichier n'avez pas été indexé, il est possible de modifier le dernier commit :
- **git add fichier_oublie**
- **git commit --amend**

  - **COMMENT SE POSITIONNER SUR UN ETAT PRECIS ?**

- Vous souhaitez revenir sur une version stable car vous avez introduit de nombreuses erreurs par exemple :
- À l'aide de git log vous pouvez repérer l'état stable sur lequel vous souhaitez vous positionner, copier le SHA puis
- **git checkout 5cee65e14b2e5fe65d4b188fa4a19571b6de386a**
- NB : cela ne veut pas dire que vous avez perdu l'état pb, vous pouvez y revenir sauf si vous l'avez supprimé

Travailler avec des dépôts distants

- Un dépôt distant est soit votre travail local soit celui d'un autre dev. Ce dépôt se trouve généralement sur des sites web tels que GitHub ou GitLab. Les dépôts distants sont donc des versions de votre projet local avec des droits spécifiques (lecture/écriture). Vous pouvez partager vos projets en les poussant en ligne et les devs qui le souhaitent peuvent cloner votre dépôt et inversement, ceci afin de collaborer sur des projets.
- **Git remote -v** pour visualiser les serveurs distants enregistrés

```
C:\Users\El-BabiliM\angular-workspace\airbus-app-ngrx>git remote -v
origin  https://github.com/elbabili/fms-ngrx-aircrafts.git (fetch)
origin  https://github.com/elbabili/fms-ngrx-aircrafts.git (push)
```

- Afin de travailler sur un dépôt distant, il faut soit :
  - 1/ Cloner localement : git clone https://github.com/elbabili/fms-ngrx-aircrafts.git
  - 2/ Créer un dépôt en ligne vide puis l'associer au dépôt local :
    - Git remote add origin https://github.com/elbabili/Test-Git.git
      - **Cela sous entend que vous avez un compte sur GitHub**
      - Origin correspond à l'url donc au dépôt distant
    - Puis git push -u origin main afin de pousser votre travail
      - **Attention, si un autre dev a poussé avant vous, vous devez d'abord pull avant de push**
    - Afin que cela fonctionne, il faut avoir des droits d'accès en écriture
      - https
      - Ssh
      - GitHub Cli
    - Git pull récupère les données depuis le serveur cloné et fusionne automatiquement une branche distante dans votre branche locale.
- **Git remote show origin** pour visualiser plus d'infos d'un dépôt distant, ex :

```
C:\Users\El-BabiliM\Desktop\HelloJava>git remote show origin
* remote origin
  Fetch URL: https://github.com/elbabili/Test-Git.git
  Push  URL: https://github.com/elbabili/Test-Git.git
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

Retirer et renommer des dépôts distants

Git remote rename origin toto pour modifier le nom d'un dépôt distant

Git remote rm origin pour retirer un dépôt distant suite à un changement de serveur par ex

Gestion des Etiquettes ou Tag

La commande Tag permet notamment de gérer les versions, il en existe 2 types :

- **Légères** : pointeur sur un commit spécifique

  git tag v1.0

- **Annotées** : stockées en tant qu'objets dans la bdd git (nom, email, date, message d'étiquetage, signature…)

  git tag –a –m 'version 1.1' **-a pour annoté** **-m pour ajout message**

Git tag va afficher tous les tags

v0.1

v1.3

v1.4

Git show v0.1 affiche des infos sur le commit associé

Git tag –a v1.2 f7e9e2b0e permet d'associer un tag

Avec un commit à posteriori

Git push origin v1.2 transfère le tag vers le serveur

Git push origin --tags transfère tous les tags

Git tag –d v1.3 supprime un tag en local puis serveur

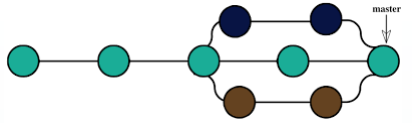
```
C:\Users\El-BabiliM\Desktop\HelloJava>git push origin :refs/tags/v1.0
To https://github.com/elbabili/Test-Git.git
 - [deleted]         v1.0

C:\Users\El-BabiliM\Desktop\HelloJava>git push origin --delete v1.1
To https://github.com/elbabili/Test-Git.git
 - [deleted]         v1.1
```

```
C:\Users\El-BabiliM\Desktop\HelloJava>git tag -a v1.2 f7e9e2b0e

C:\Users\El-BabiliM\Desktop\HelloJava>git tag
v1.0
v1.1
v1.2

C:\Users\El-BabiliM\Desktop\HelloJava>git log --pretty=oneline
119427fc9b3889756f1f8ae8cdeafd9773fdda93 (HEAD -> main, origin/main) update readme
f7e9e2b0e75fdda88df428e9606d6eb1d8cb0a57 (tag: v1.2) create readme
6ac2f58d0e25077adebff260c83c22b78d431f29 add rational class
5cee65e14b2e5fe65d4b188fa4a19571b6de386a (tag: v1.1) new method mul
f3a8b29fcb20c13a43b80182ca34e9e9a86706f8 ajout methode addition
6b99376d422cb8a5d6c25ff924a0efaf049478bb gestion des opérations
3040ff372badcf5b68039405c03ab7ce7b61fdd1 (tag: v1.0) first commit

C:\Users\El-BabiliM\Desktop\HelloJava>git show v1.2
tag v1.2
Tagger: elbabili <elbabilimohamed@gmail.com>
Date:   Fri Feb 4 16:41:20 2022 +0100

version 1.2

commit f7e9e2b0e75fdda88df428e9606d6eb1d8cb0a57 (tag: v1.2)
Author: elbabili <34126012+elbabili@users.noreply.github.com>
Date:   Fri Feb 4 13:54:06 2022 +0100

    create readme

diff --git a/README.md b/README.md
new file mode 100644
index 0000000..ff7d933
--- /dev/null
+++ b/README.md
@@ -0,0 +1,3 @@
+# Test-Git
+
+# c'est ok !
```

COMMENT SE POSITIONNER SUR UNE VERSION PRECISE ? **Git checkout v1.1**

NB : Attention, si vous réaliser un commit sans l'associer à aucune branche, il ne sera plus visible sauf si vous créer une branche associée par ex ici à des corrections particulières : **git checkout –b V1.1.X**

Gestion des Branches avec Git (1)



Sans doute un des outils de Git, le plus puissant parce qu'allégé, rapide et simple d'utilisation. L'idée ici est de pouvoir créer une dimension parallèle sous forme de branche qui permet de coder et tester sans impacter la branche principale puis de fusionner avec celle-ci le travail effectué.

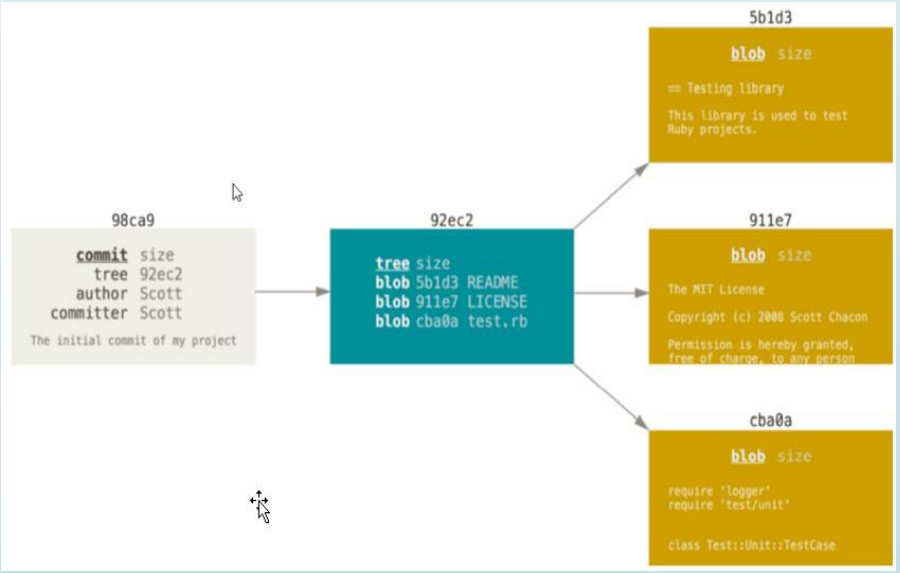
Sur un commit, git stocke un objet commit qui contient un pointeur (SHA) vers la photocopie ou instantané(snapshot) du contenu que vous avez indexé. Cet objet contient également :

- Nom
- Prénom
- message de commit
- Des pointeurs vers le ou les commits précédents. (pas de commit parent pour le commit initial)
- un parent pour un commit normal et de multiples parents pour un commit qui résulte de la fusion d'une ou plusieurs branches

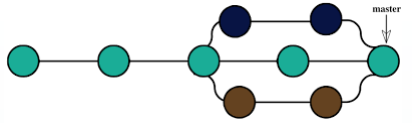
Pour comprendre, prenons l'exemple d'un répertoire contenant 3 fichiers

- Indexation des fichiers [add] calcule une empreinte (checksum) pour chacun à l'aide de la fonction de hachage SHA-1
- Stockage des contenus de chaque fichiers(blobs) dans le dépôt git
- Ajout des 3 empruntes à la zone d'index (staging area)
- Git commit a pour conséquence l'ajout d'un arbre (Tree) : qui liste le contenu du répertoire et spécifie quels noms de fichiers sont attachés à quels blobs

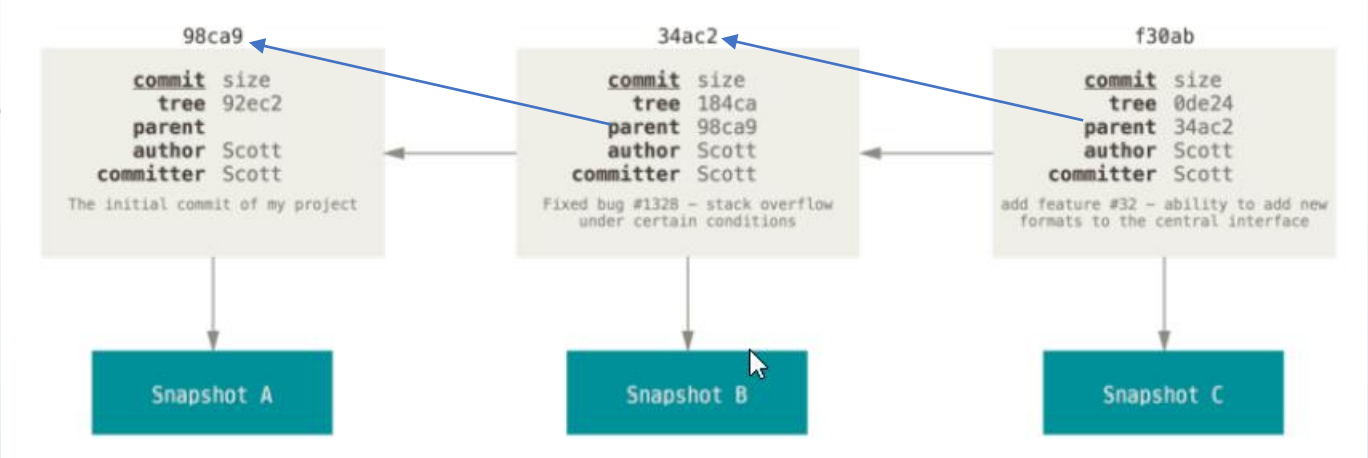
Git crée alors un objet commit qui contient les méta données et un pointeur vers l'arbre de la racine du projet de sorte à pouvoir récréer l'instantané à tout moment



Gestion des Branches avec Git (2)

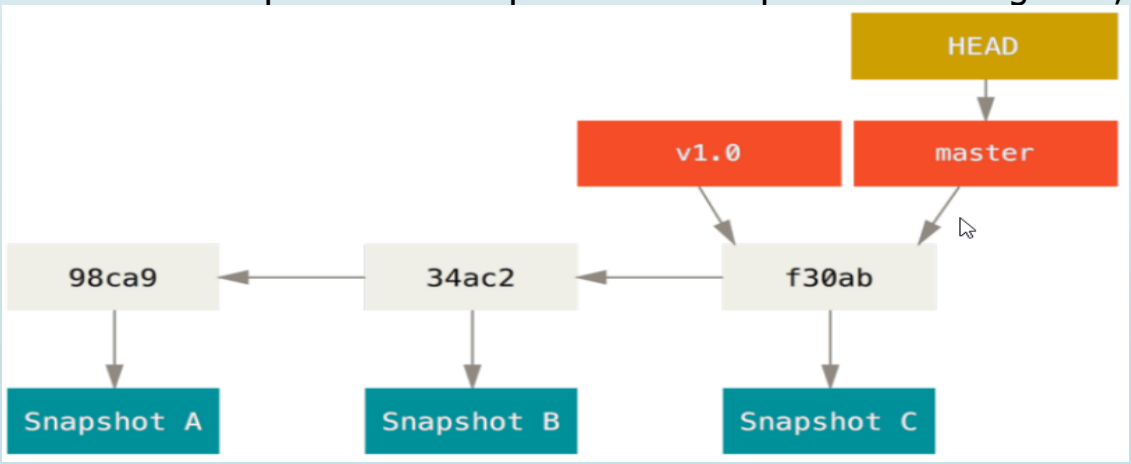


Si vous faites des modifs et validez à nouveau, le prochain commit stocke un pointeur vers le précédent

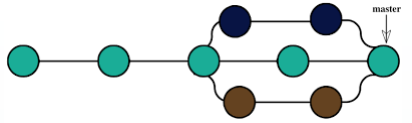


Une branche dans git est simplement un pointeur léger et déplaçable vers un de ces commits

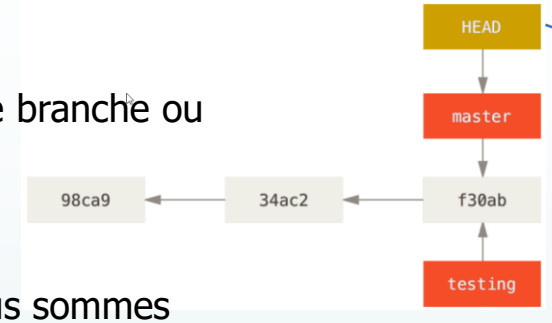
La branche par défaut s'appelle master ou main. Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés. A chaque validation, le pointeur de la branche master avance automatiquement. La branche master n'a rien de spécial hormis qu'elle est crée par défaut sur git init, on peut lui changer son nom : main



Gestion des Branches avec Git (3)



Git branch testing Pour créer une nouvelle branche ou pointeur vers le commit courant :

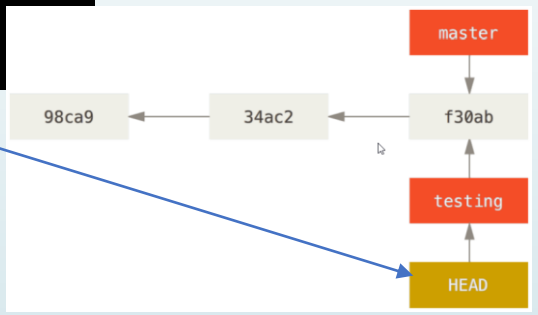


Git connait la branche ou nous sommes à l'aide du pointeur head

git log –decorate permet de savoir ou nous sommes

```
C:\Users\El-BabiliM\Desktop\HelloJava>git log --decorate
commit 119427fc9b3889756f1f8ae8cdeafd9773fdda93 (HEAD -> main, origin/main, testing)
Author: elbabili <34126012+elbabili@users.noreply.github.com>
Date:   Fri Feb 4 14:03:29 2022 +0100

    update readme
```

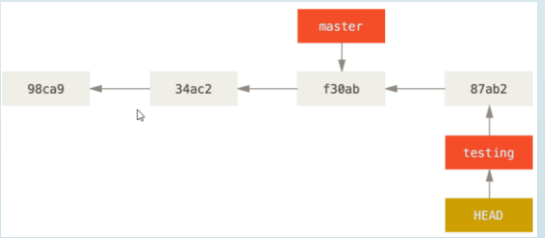
Ici main et testing pointent sur le même commit



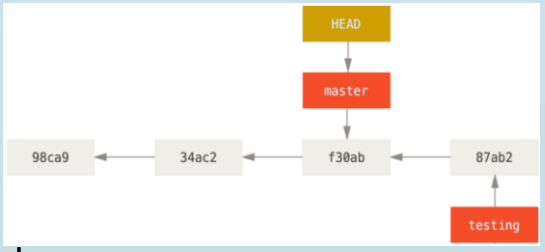
Git checkout testing permet de basculer sur une autre branche

```
C:\Users\El-BabiliM\Desktop\HelloJava>git log --decorate
commit 119427fc9b3889756f1f8ae8cdeafd9773fdda93 (HEAD -> testing, origin/main, main)
Author: elbabili <34126012+elbabili@users.noreply.github.com>
Date:   Fri Feb 4 14:03:29 2022 +0100

    update readme
```

Git commit –am « msg » décale le pointeur head sur le nouveau commit :

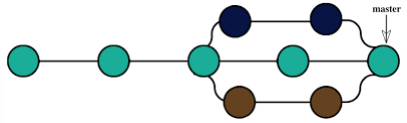



```
C:\Users\El-BabiliM\Desktop\HelloJava>git log --decorate
commit eb504012b919bceab9ef11a0a53459149eef919d (HEAD -> testing)
Author: elbabili <elbabilimohamed@gmail.com>
Date:   Mon Feb 7 17:09:03 2022 +0100

    modif branche testing

commit 119427fc9b3889756f1f8ae8cdeafd9773fdda93 (origin/main, main)
Author: elbabili <34126012+elbabili@users.noreply.github.com>
Date:   Fri Feb 4 14:03:29 2022 +0100

    update readme
```

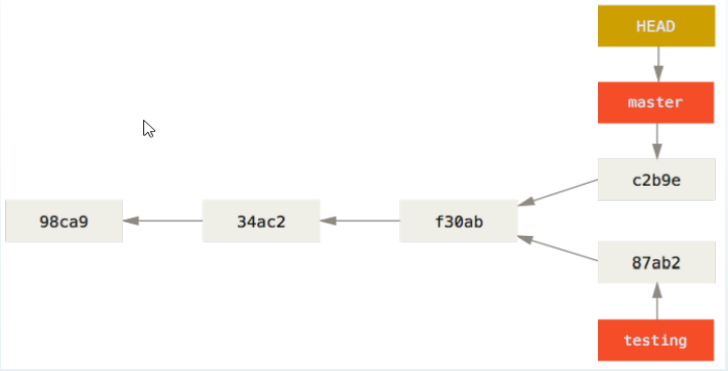


Git checkout main permet de revenir sur la branche main, du coup, tout ce qui a été fait sur la branche testing n'apparaitra plus ici et peut être considéré comme perdu si vous ne fusionnez pas les 2 branches, idéal pour tester une fonctionnalité sans valider.

Gestion des Branches avec Git (4)



Lorsque vous changez de branche, les fichiers de votre répertoire de travail sont modifiés. Si vous basculez vers une branche plus ancienne, votre répertoire sera remis dans l'état dans lequel il était lors du dernier commit sur cette branche. Si git ne peut pas effectuer cette action proprement, il ne vous laissera pas changer de branche.



Git log testing permet d'afficher l'historique sur cette branche à partir d'une autre branche

git log –all permet d'afficher toutes les branches

git log --oneline --decorate --graph –-all va en + afficher les endroits ou sont positionnés les pointeurs de branche ainsi que la manière dont l'historique a diverge

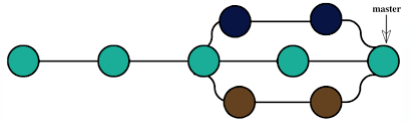
```
* eb50401 (testing) modif branche testing
* 119427f (HEAD -> main, origin/main) update readme
* f7e9e2b (tag: v1.2) create readme
* 6ac2f58 add rational class
| * 2eca0df (origin/v1.1.X, v1.1.X) autre dimension
|/
* 5cee65e (tag: v1.1) new method mul
* f3a8b29 ajout methode addition
* 6b99376 gestion des opérations
* 3040ff3 first commit
```

Une branche git n'est en fait qu'un simple fichier contenant les 40 caractères de l'empreinte SHA-1 du commit sur lequel elle pointe !

git checkout -b « new branch » permet de créer une nouvelle branche et de basculer directement dessus.

Git switch équivaut à git checkout aussi on peut switcher sur une branche qu'on vient de créer git switch -c « new branch »

Branches & fusions : les bases (1)



Imaginons un scénario qui devrait sans doute vous arriver très vite. Vous travaillez actuellement sur une nouvelle fonctionnalité(feature). Vous avez donc créer une nouvelle branche puis après 2 heures de code, vous recevez un appel qui indique un pb critique en production qui doit être réglé au plus vite :

1/ vous basculez sur la branche de production

2/ vous créer une nouvelle branche à partir de là puis ajouter le correctif

3/ après avoir testé, vous fusionnez la branche du correctif et poussez le résultat en production

4/ vous pouvez supprimez la branche devenu inutile

5/ vous rebasculez sur la branche initiale et continuer votre travail du jour

Reprenons, Vous travailliez sur une nouvelle tache qui a donné son nom à la branche de travail : **iss53**

→ **git checkout -b iss53**

→ **git commit –am « ajout tache c3 »**

→ **INCIDENT**

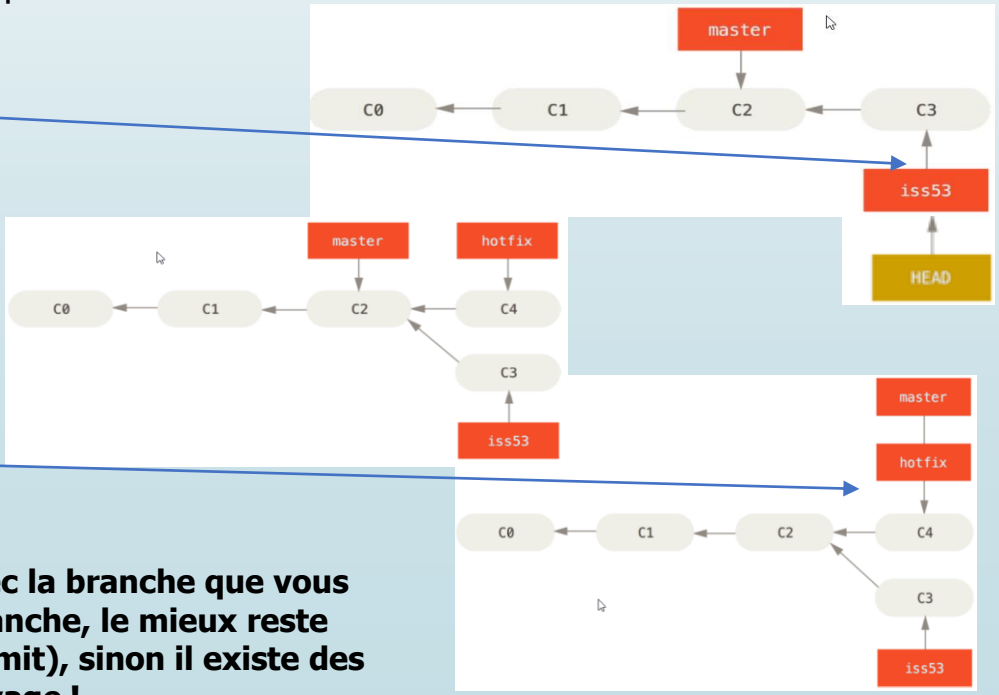
→ **Git checkout main**

→ **Git branch hotfix + codage/test + commit C4**

→ **Git checkout main**

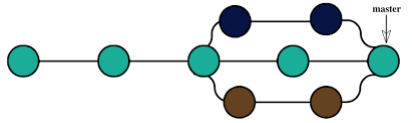
→ **Git merge hotfix**

→ **Git branch –d hotfix**



NB : En cas de modifs non validées en conflit avec la branche que vous extrayez, Git ne vous laissera pas changer de branche, le mieux reste d'avoir une copie de travail propre (git add/commit), sinon il existe des moyens de contourner cela : remissage et nettoyage !

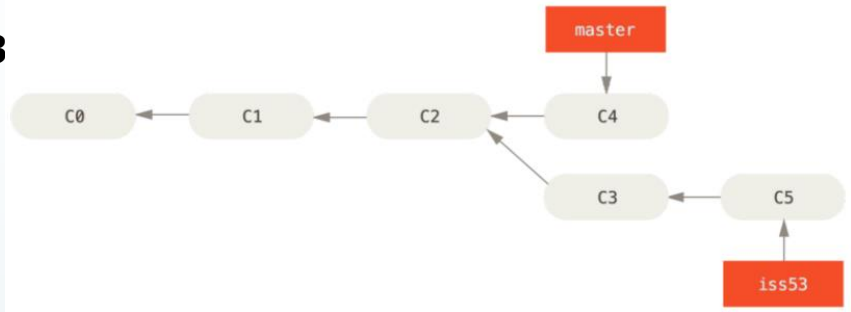
Branches & fusions : les bases (2)



Après avoir gérer l'incident, retour sur votre branche de travail **iss53**

Notez ici que le travail effectué sur hotfix et fusionné sur la branche principale n'apparait pas ici.

Pour ce faire(récupérer les dernières modifs dans le projet), on peut fusionner la branche main dans iss53 : **git merge main**

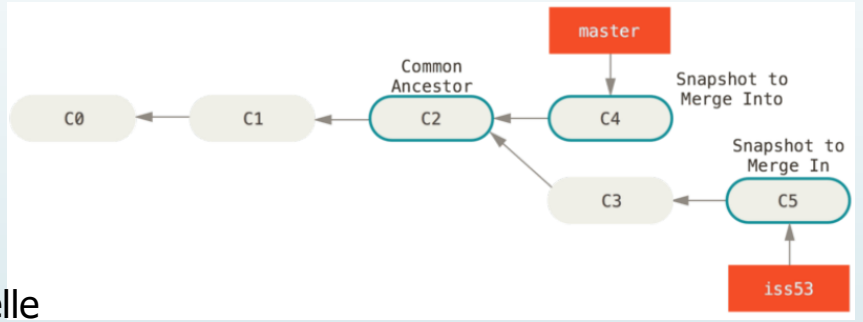


L'étape suivante consiste à fusionner iss53 à main pour partager notre travail avec le reste de l' équipe → pull :

Git checkout main

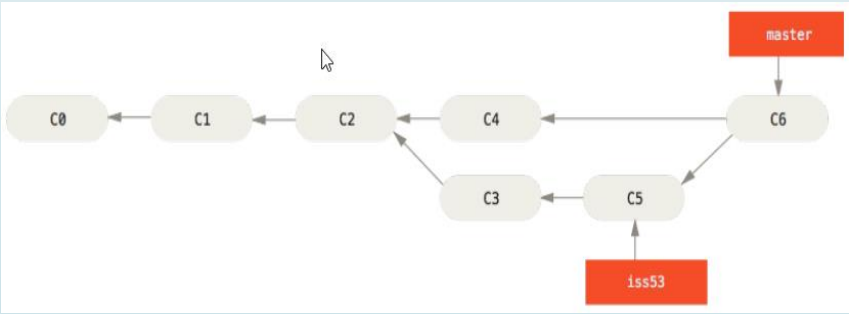
Git merge iss53 : three way merge, à partir de C2, C4 et C5

Au lieu de déplacer le pointeur de branche, git crée un nouvel instantané(commit) qui résulte de la fusion à trois sources. On appelle ceci un commit de fusion (merge commit) car il a plusieurs parents.

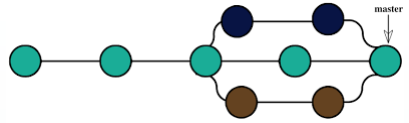


Git branch –d iss53

Tache en cours : DONE !



Conflits de fusions



Il arrive que la fusion échoue par exemple si vous avez modifié le même code du même fichier dans les 2 branches que vous souhaitez fusionner. Git n'étant pas en mesure de réaliser l'action, vous obtenez un conflit qu'il faut résoudre avant de relancer la fusion.

```
C:\Users\El-BabiliM\Desktop\HelloJava>git merge testing
Auto-merging Hello.java
CONFLICT (content): Merge conflict in Hello.java
Automatic merge failed; fix conflicts and then commit the result.
```

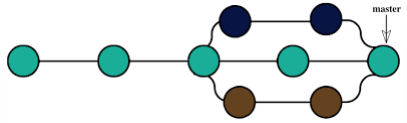
Vous pouvez obtenir des informations sur la nature du conflit, quels sont les fichiers qui n'ont pas fusionnés via **git status**

Afin de vous aider à résoudre les conflits, git ajoute des marques de résolution de conflits dans les fichiers concernés, vous pouvez les visualiser à l'aide de votre ide préféré (configuré à l'install) ou github Desktop par ex, après quoi il faudra faire des choix suivi de commit de part et d'autre avant de relancer la fusion. Assurez-vous que tout est rentré dans l'ordre : **git status**

```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
        if(age > 17 && age < 26) {
=======
        if(age >= 18 && age <= 25) {
>>>>>>> testing (Incoming Change)
```

Des options de modifs automatiques sont proposées, vous pouvez les sélectionner ou pas

Infos complémentaires sur les branches



Git branch pour avoir la liste des branches, celle qui est précédée d'un * correspondant à la branche active

Git branch –v pour visualiser la liste des derniers commits sur chaque branche

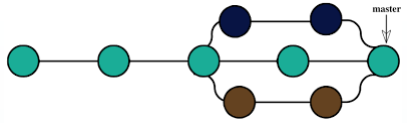
Git branch –-merged ou **–-no-merged** pour filter les branches fusionnés ou pas avec main

Git branch –move ancien nouveau pour change de nom de branche

git push --set-upstream origin nouveau pour maj branche depot

Changer le nom de la branche master….

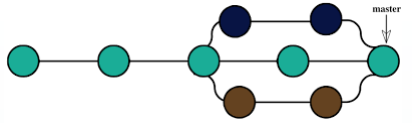
Bonnes pratiques avec les branches



Parmi les bonnes pratiques, on retrouve une branche **master** ou **main** qui contient la version mise en production et une branche **developer** qui sert de travail aux développeurs. En effet, les développeurs vont créer des branches à partir de celle-ci puis fusionner leurs travaux(branches/topics), et une fois stable et testée, la branche dev est fusionné avec la branche principale, le tout représentant un cycle de développement.

Il existe de nombreuses stratégies en fonction de la taille et du type de projet aussi, il faut se rapprocher de l'équipe de dev et(ou) du techlead pour connaître celle que vous devez utiliser.

Branches de suivi à distance (1)



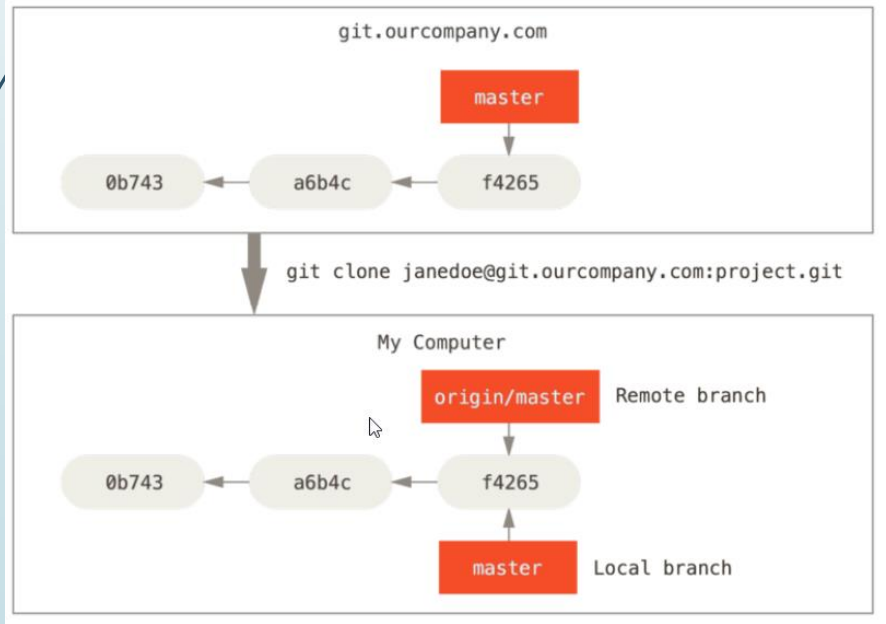
Git ls-remote ou **git remote show** pour visualiser l'ensemble des références distantes

Néanmoins, ont peut aussi utiliser **Les branches de suivi à distance** qui sont des références (des pointeurs) vers l'état des branches sur votre dépôt distant. Ce sont des branches locales qu'on ne peut pas modifier; elles le sont automatiquement lors de communication réseau. En effet, à chaque connexion, elles agissent comme des marques pages pour indiquer l'état des branches distantes.

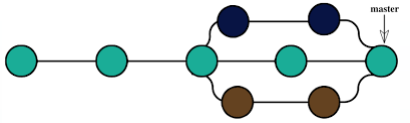
Elles prennent la forme : **serveur distant/branche**, par ex si vous souhaitez visualiser l'état de la branche master sur le dépôt distant origin, il suffit de vérifier la branche **: origin/master**

De même, un collègue peut publier une branche **iss53** sur le serveur qui porte le même nom que la votre en local, à la différence que la branche sur le serveur pointe sur **origin/iss53**

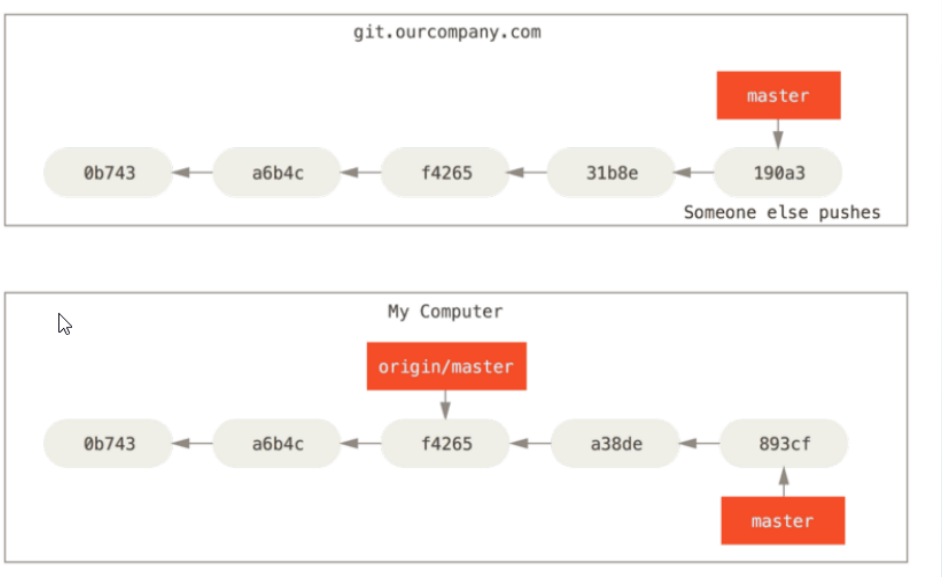
Exemple : on clone un dépôt distant qui est nommé automatiquement origin, tout l'historique est récupéré, un pointeur est crée sur l'état actuel de la branche master : origin/master, git crée aussi notre propre branche master afin de pouvoir travailler.



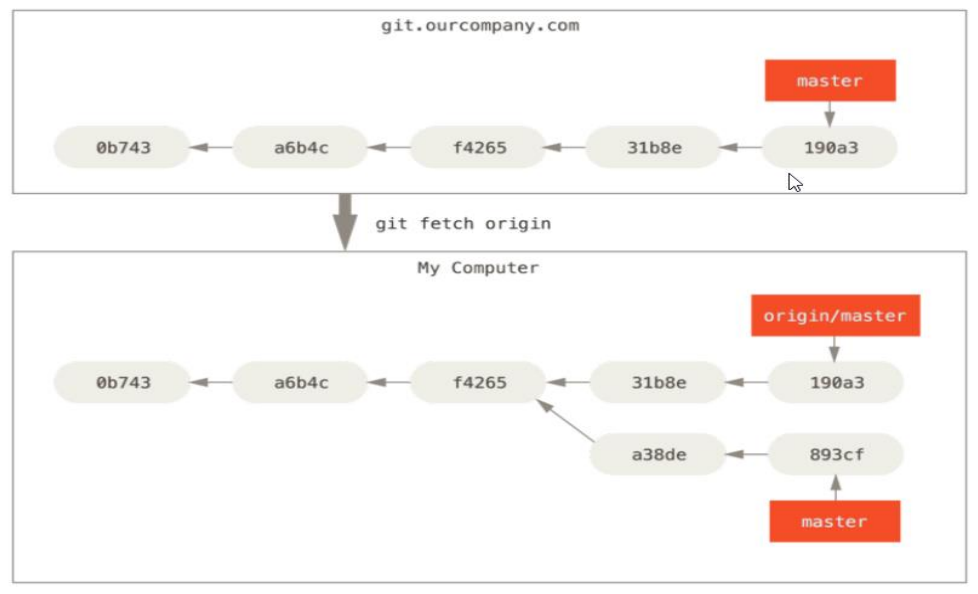
Branches de suivi à distance (2)



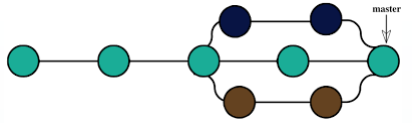
Supposons que qq publie sur le dépôt distant et met à jour la branche master, que localement, vous aussi faites des modifs, les historiques divergent :



Git Fetch origin pour synchroniser les travaux, la commande recherche le serveur hébergeant origin (git.ourcompany.com), y récupère toutes les nouvelles données et met à jour votre bdd locale en déplaçant votre pointeur origin/master.



Pousser des branches



Lorsque vous souhaitez partager une branche avec votre équipe ou plus, vous devez la pousser sur un serveur distant (accès en écriture). En effet, vos branches locales ne sont pas automatiquement synchronisés sur le serveur distant.

C'est le cas par exemple d'une branche « **correctif52** » sur laquelle vous travaillez et que vous souhaitez partager avec tout ou partie de l'équipe afin qu'ils collaborent aussi : Git push (serveur distant) (branche)

Git push origin correctif52

```
C:\Users\El-BabiliM\Desktop\HelloJava>git push origin correctif52
Enumerating objects: 19, done.
Counting objects: 100% (19/19), done.
Delta compression using up to 16 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (17/17), 1.75 KiB | 595.00 KiB/s, done.
Total 17 (delta 12), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (12/12), completed with 2 local objects.
remote:
remote: Create a pull request for 'correctif52' on GitHub by visiting:
remote:      https://github.com/elbabili/Test-Git/pull/new/correctif52
remote:
To https://github.com/elbabili/Test-Git.git
 * [new branch]      correctif52 -> correctif52
```

Delors, lorsqu'un collègue récupère les données depuis le serveur : **Git fetch origin**

Il aura une référence ou pointeur vers la branche correctif52, ce n'est donc pas une copie locale modifiable.

Pour fusionner ce travail dans votre branche actuelle : **git merge origin/correctif52**

To be continued : Rebase …

Utiliser Git avec Eclipse

1/ Créer un repo dans github

2/ ouvrir la perspective git dans Eclipse

3/ click sur "clone a Git repository" et insérer les données du repo crée + id + pwd

4/ Crée un projet sur Eclipse

5/ Projet Eclipse / clic droit / share project / Repository cloné.git

6/ Crée vos classes puis faire votre premier commit

7/ projet java / clic droit / team / commit [GitStaging] / sélectionner tout puis déplacer dans staged + ajouter un message dans commit message + commit and push et suivre les étapes...

8/ Vérifier sur Github si tout est bon

Il existe une variante ici lorsqu'on récupère un projet existant pour lui apporter des modifs

1/ clone a git repo avec le lien à cloner

2/ import project puis exécuter/modifier

Noter que vous pourriez forker ce projet puis mettre à jour sur votre github

Travailler sur un WorkFlow d'équipe avec GitHub ou GitLab

Dans le cas de travail en équipe, il existe 2 stratégies pour travailler sur la même dépôt :

→ Centraliser autour du techlead qui a des droits lui permettant de valider les **pull request ou merge request** venant des devs lorsqu'ils souhaitent soumettre leurs productions, c'est le techlead qui fait la revue de code avant de valider ou pas, c'est lui aussi qui gère les conflits et choisi de garder telle variante de telle branche. Pour ce faire le techlead doit parfaitement maitriser tous les projets dont il a la charge.

→ L'autre stratégie met toute l'équipe au même niveau, chacun devra en cas de conflit, les régler individuellement en communiquant avec les autres.

Process GitHub

- les personnes souhaitant travailler sur votre repo doivent se connecter dessus : git remote add "url du dépot distant crée"

- ces personnes pourront récupérer localement le repo distant grâce à : git pull origin master

- afin qu'ils puissent apporter des modifs, il faut leur donner un accès en écriture : Repo/Settings/Nom des personnes

- elles pourront effectuer des modifs et rapatrier celles-ci grâce à : git push origin master

- le reste de l'équipe pourra donc récupérer les mises à jour : git pull origin master

Process GitLab

- On peut travailler en ligne ou bien installer gitlab sur le serveur de l'entreprise

- création des utilisateurs en ligne (individuel) ou sur le serveur (admin système)

- création d'un ou plusieurs groupes suivie d'invitations personnelles pour se connecter

- chacun utilisateur doit cloner le projet de son groupe puis vérifier que c'est bon

- en cas de modif : git add .

- git commit -a -m "j'ai changé qqchose"

- git switch -c main pour se positionner sur la branche main

- git push origin main / git pull origin main

Intégration continue ….

Les entreprises qui utilisent Git, GitHub & GitLab

GitHub

100 M*
de repositories dans le monde

40 M*
de développeurs dans le monde

2.1 M*
d'entreprises dans le monde

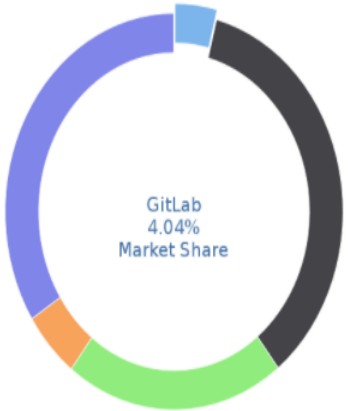
Adopté par plus de 2.1 millions* d'organisations

airbnb SAP IBM PayPal Bloomberg
Spotify Walmart

en août 2019

GitLab
Software Configuration Management

10,252
Companies using GitLab



powered by enlyft.com

Software Configuration Management

GitLab (4.04%)
Atlassian JIRA (35.41%)
GitHub (20.96%)
Bugzilla (5.51%)
View other alternative products

Next steps

- Java SE 8
- Algorithmique
- **Git/GitHub**
- La Programmation orientée objet

Ressources

- https://git-scm.com/
- **Livre ProGit :** Git Scott Chacon & Ben Straub