Travaux Pratique

Apprendre, Utiliser et comprendre Git

Note d'utilisation

La réalisation du TP s'effectue en ligne de commandes afin de manipuler Git et d'analyser les différentes étapes.

- Attention : Ce TP nécessite de manipuler un dossier sous le contrôle de Git. Il se peut qu'à un moment, on se perds dans les manipulations d'où y aller étape par étape.

- Les captures résultats proposées sont là pour vous aider en donnant les commande à saisir et des informations et indications.
- Attention : les identifiants à saisir seront ceux de votre dépôt.

Contenu du TP

1. Les bases
    1. initialisation (init, status)
    2. les 3 espaces: workdir, index, dépot (add, commit, log, diff)
2. Remonter le temps (checkout, revert, reset)
3. Dépôt distant
    1. partager son dépôt (bare, remote)
    2. utiliser un dépôt distant (clone, push, fetch, pull)
4. Les branches (branch, checkout, merge, fast-forward, rebase)
5. Gestion des branches distantes, tags

TRAVAUX PRATIQUE GIT

Les bases de Git

LES BASES

- Initialiser un nouveau répertoire **TpGit** avec Git
  - git init
- Création d'un document
  - Pour l'exercice, créer README.md
- Contrôler l'état du dépôt
  - git status 1
  - *Pour savoir où on en est à tout moment dans notre dépôt.*
- Ajouter ce fichier dans l'index
  - git add README.md
  - git status 2
  - ***L'index*** *peut être vu comme une zone de transit en attente de validation*
  - ***git status*** *montre ici que le fichier est passé dans l'index. Il est prêt à être validé, ce qui enregistrera dans un dépôt local la version du fichier se trouvant actuellement dans la zone d'index.*
  - *Note : comme il s'agit d'un nouveau fichier, la commande permettant le désindexage (pour sortir le fichier de l'index) est proposée : git rm --cached ….*

1

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit
$ git init
Initialized empty Git repository in D:/PROJETS/tutoGit/tpGit/.git/

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main

No commits yet

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

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

2

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git add README.md

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
```

LES BASES
Mon premier commit

- Commiter le fichier
  - *Pour valider le nouveau fichier et ainsi créer son 1er commit*
  - git commit -m "Ajout de Readme"
  - git status
  - *Vérifiez que git status vous dit que plus rien n'est à valider.* 3
  - *Le message est une information pour permettre une description de votre commit dans l'historique.* ***Il est donc important que ce message soit explicite.***
  - *Le commit n'est qu'une validation (un enregistrement) local, qui ne nécessite pas de serveur sur le réseau.*

3

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "Ajout de README"
[main (root-commit) 0d4f59b] Ajout de README
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean
```

1ère partie

1. Ajouter un nouveau fichier hello.py dans votre repertoire de travail avec le contenu print ("Hello World")

```
D: > PROJETS > tutoGit > TPGit2025 > hello.py
  1    print("Hello World")
```

2. Commiter ce fichier avec le message "Ajout du programme hello" 4

3. Ajouter du contenu dans votre fichier Hello.py

```
def ecrire(chaine):
    print (chaine)

print ("Hello World")
```

4. Contrôler le status de votre workdir 5

5. Visualisez les modifications apportées
   - git diff 6 [en vert, il indique les différences entre les 2 versions de votre fichier]
   - Ajouter la modification à l'index sans la valider (**pas de commit donc**) et revisualisez les modifications 7

4

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello.py

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

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "Ajout du programme hello"
[main ef6b67a] Ajout du programme hello
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

5

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ 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.py

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

6

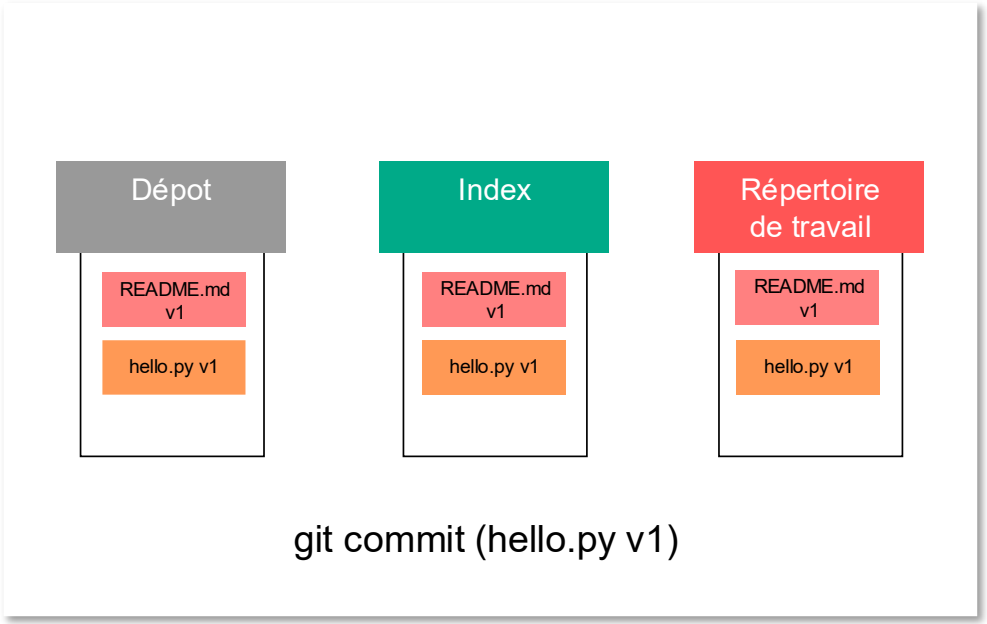
```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff
diff --git a/hello.py b/hello.py
index df94cd9..f966ae9 100644
--- a/hello.py
+++ b/hello.py
@@ -1 +1,4 @@
+def ecrire(chaine):
+    print(chaine)
+
 print("hello World")
\ No newline at end of file
```

7

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git add hello.py

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

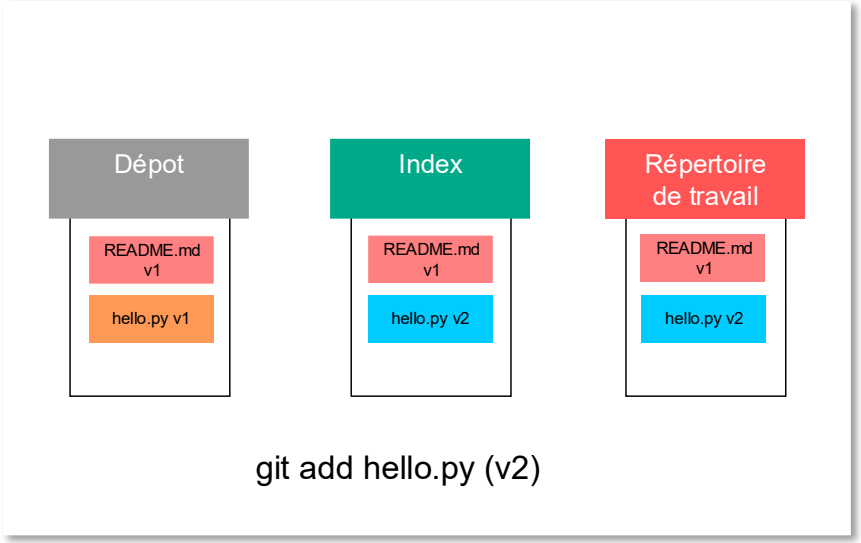
Les trois espaces (RAPPEL)

- GIT comporte 3 espaces, on parle des 3 arbres de GIT
  - votre répertoire de travail (workdir, en anglais) : c'est là que vous faites toutes vos modifications.
  - l'index (index ou stage, en anglais): zone de transit avant la validation des modifications.
  - le dépôt (repository, en anglais): l'ensemble des modifications validées, reliées entre elles.
- Donc pour le moment, votre dépôt et votre répertoire de travail sont la même chose.



- Les trois espaces (RAPPEL)

- Remarquez que maintenant git diff ne voit aucune différence car, par défaut, il compare l'espace de travail avec l'index.
- Pour visualiser les différences entre l'index et le dernier commit, utilisez git diff --staged (pour rappel, le stage est l'espace d'index). 8
- Votre dépôt ressemble à cela schématiquement.
  - Votre répertoire de travail et l'index sont identiques (on a juste ajouté dans l'index la modification)
  - Votre dépôt contient une différence car on n'a pas encore commité.



8

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff --staged
diff --git a/hello.py b/hello.py
index df94cd9..f966ae9 100644
--- a/hello.py
+++ b/hello.py
@@ -1 +1,4 @@
+def ecrire(chaine):
+    print(chaine)
+
 print("hello World")
\ No newline at end of file
```

- Modifier le programme pour utiliser la fonction ecrire() plutôt que print()

```python
def ecrire(chaine):
    print (chaine)

ecrire ("Hello World")
```

- Afficher l'état courant (status)

9

- *On remarque ici 2 modifications sur notre fichier : une modification indexée (l'ajout de la fonction) et une autre non (l'utilisation de la fonction)*

- Visualisez les différences entre les espaces :

10

1. **espace de travail <-> index** (pour rappel : comportement par défaut de git diff)
2. **index <-> dépôt** (dernier commit) (pour rappel : git diff --staged)
3. **espace de travail <-> dépôt** (git diff head)
    - Remarquez que cette diff est la fusion des deux autres. Chaque espace est une version différente du fichier hello.py

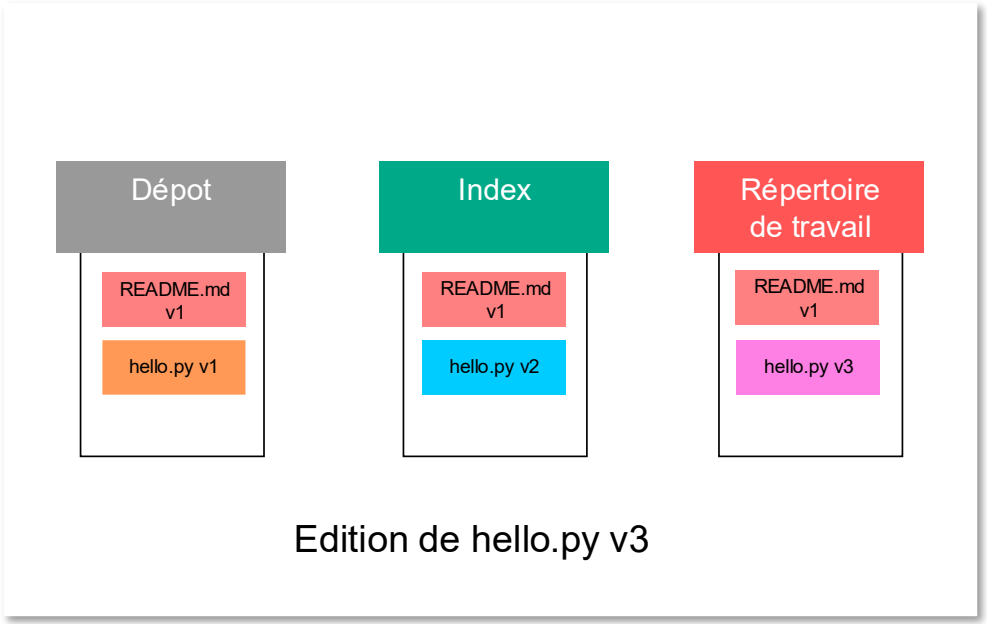
9

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

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.py
```

10

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff
diff --git a/hello.py b/hello.py
index f966ae9..80a5a17 100644
--- a/hello.py
+++ b/hello.py
@@ -1,4 +1,4 @@
 def ecrire(chaine):
     print(chaine)

-print("hello World")
\ No newline at end of file
+ecrire("hello World")
\ No newline at end of file

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff --staged
diff --git a/hello.py b/hello.py
index df94cd9..f966ae9 100644
--- a/hello.py
+++ b/hello.py
@@ -1 +1,4 @@
+def ecrire(chaine):
+    print(chaine)
+
 print("hello World")
\ No newline at end of file
```

- Chaque espace contient une version différente de fichier hello.py .



Les bases
Conclusion

- Vous allez bien séparer les choses :
  1. la création de la fonction est dans l'index, on la valide premièrement : tapez git commit -m "Ajout procédure ecrire" (1)
  2. puis on crée un commit pour son utilisation : tapez git add hello.py suivi de git commit -m "Utilisation procédure ecrire". (2)
- Utilisez git status (n'hésitez pas à user et abuser de cette commande !) et git diff à chaque étape pour suivre ce qui ce passe.

(1)

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

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.py


jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "Ajout procédure ecrire"
[main 538eb85] Ajout procédure ecrire
 1 file changed, 3 insertions(+)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ 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.py

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

(2)

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ 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.py

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

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "Utilisation procédure ecrire"
[main ac598ca] Utilisation procédure ecrire
 1 file changed, 1 insertion(+), 1 deletion(-)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean
```

Les bases

Renommage et Suppression

Les bases
Renommage et Suppression

- Ajouter un fichier non vide fichier1 et commitez-le 1
- Puis, observez ce que vous dit git status pour chacune des étapes suivantes :
  - mv fichier1 fichier2
  - git rm fichier1
  - git add fichier2 2
- *Git détecte que les 2 fichiers (1 supprimé, 1 ajouté) ont le même contenu. Il en déduit que c'est un renommage.*
- *Le renommage peut se faire aussi plus directement :*
  - *git mv fichier1 fichier2*
- Commiter ce renommage avec le message "ajout du fichier2 aprés renommage" 3
- Pour supprimer fichier2, 2 manières de faire :
  - rm fichier2
  - git rm fichier2
  - git commit avec le message "suppression du fichier2" 4
- *Ou plus direct :*
  - *git rm fichier2*
  - *git commit*

1

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fichier1

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git add fichier1

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "Ajout de fichier1"
[main cbc3942] Ajout de fichier1
 1 file changed, 3 insertions(+)
 create mode 100644 fichier1

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean
```

2

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ mv fichier1 fichier2

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    fichier1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fichier2

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git rm fichier1
rm 'fichier1'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    fichier1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        fichier2


jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git add fichier2

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    fichier1 -> fichier2
```

Captures résultats

3

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    fichier1 -> fichier2


jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "ajout du fichier2 aprés renommage"
[main 1336ab4] ajout du fichier2 aprés renommage
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename fichier1 => fichier2 (100%)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean
```

4

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ rm fichier2

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    fichier2

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git rm fichier2
rm 'fichier2'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    fichier2


jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "suppression du fichier2"
[main e085932] suppression du fichier2
 1 file changed, 3 deletions(-)
 delete mode 100644 fichier2

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean
```

Les bases

Affichage de l'historique

Les bases

Affichage de l'historique

- Vous avez déjà vu la commande git status. Elle vous permet de voir où vous en êtes et vous donne des indications utiles.
- Vous aurez également besoin régulièrement de voir l'historique de vos commits. Pour cela vous allez utiliser git log (q pour quitter) . 5
- Pour un affichage plus concis, vous pouvez utiliser l'option --oneline 6
  - *Les identifiants des commits sont réduits aux 7 premiers caractères. C'est suffisant la plupart du temps. Cette longueur est ajustée si nécessaire, pour les très gros projets notamment.*
  - Comme c'est un affichage assez pratique, faites-en un alias :
    - *git config --global alias.lg "log --oneline"*
    - *git lg pour lancer l'alias* 7
- Pour avoir plus de détail sur un commit particulier vous pouvez utiliser git show suivi de l'identifiant du commit 8

Captures résultats

5

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git log
commit e085932baae5c724330688e0b178bf16e283a00b (HEAD -> main)
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 14:46:28 2024 +0200

    suppression du fichier2

commit 1336ab44be6a544acf59a286cd307d8a00277a51
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 14:44:32 2024 +0200

    ajout du fichier2 aprés renommage

commit cbc39422ad524f77b8e98933df1bf74991cfaa1b
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 14:38:10 2024 +0200

    Ajout de fichier1

commit ac598ca9c50269e4858d22ce1283d963e5b0f6e5
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 14:33:17 2024 +0200

    Utilisation procédure ecrire

commit 538eb85d57b155bfbacc80a0ab6b789935e0bf4b
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 14:31:10 2024 +0200

    Ajout procédure ecrire

commit ef6b67a44952983e98145d260e9e1756de79c062
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 13:56:14 2024 +0200

    Ajout du programme hello

commit 0d4f59bb624586d23f05fc8883caeaeb457361e2
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 13:50:33 2024 +0200

    Ajout de README
```

6

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git log --oneline
e085932 (HEAD -> main) suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

7

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit (main)
$ git config --global alias.lg "log --oneline"

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit (main)
$ git lg
872ab39 (HEAD -> main) suppression fichier2
6ed4913 fichier2
434990f ajout fichier1
0d98bf4 ajout modification hello.py
e524e43 aprés les 3 branches
92e30f8 ajout hello.py
0ed3180 initial commit
```

Captures résultats

8

Exemple de git show …

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git show ac598ca
commit ac598ca9c50269e4858d22ce1283d963e5b0f6e5
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 14:33:17 2024 +0200

    Utilisation procédure ecrire

diff --git a/hello.py b/hello.py
index f966ae9..80a5a17 100644
--- a/hello.py
+++ b/hello.py
@@ -1,4 +1,4 @@
 def ecrire(chaine):
     print(chaine)

-print("hello World")
\ No newline at end of file
+ecrire("hello World")
\ No newline at end of file
```

Retour au TP : revenir en arrière

Remonter le temps

Vous décidez de procéder à une refonte totale de votre programme en commençant par le supprimer.

- rm hello.py
- git rm hello.py
- Commiter cette suppression avec le message "Refonte Total"
- Puis afficher l'historique simplifié de votre dépôt (1)
- Votre historique devrait être le suivant (2)
- *Cette suppression de hello.py est un peu brutale, non ? Vous auriez peut-être voulu garder une trace de votre programme... et vous n'avez pas fait de fichier .old ! Ni un répertoire d'archive !*
- *Grâce à Git, vous pouvez consultez très facilement les anciennes versions de votre dépôt à l'aide de la commande **git checkout**, puisqu'elle permet de remettre le contenu du répertoire de travail dans un état précédent.*

Captures résultats

1

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ rm hello.py

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git rm hello.py
rm 'hello.py'

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


jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "refonte total"
[main 262d3d2] refonte total
 1 file changed, 4 deletions(-)
 delete mode 100644 hello.py
```

2

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
262d3d2 (HEAD -> main) refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Remonter le temps (1)

En cas de suppression brutale d'un fichier, vous pouvez consulter très facilement les anciennes versions de votre dépôt à l'aide de la commande git checkout puisqu'elle permet de remettre le contenu du répertoire de travail dans un état précédent.

- Repérez l'id du commit précédant la suppression du fichier grâce à git log, et procédez au déplacement de HEAD avec checkout (3)
  - D'où l'importance d'avoir des messages de commit clair et précis
  - *Regardez le contenu de votre répertoire de travail : le fichier hello.py est à nouveau présent. Vous pouvez, à nouveau, visualiser son contenu.*
- Si vous affichez l'historique, vous pouvez constater que tous les commits suivants ont disparu (tout du moins en apparence). (4)
  - Pour les retrouver, nous savons comment procéder : il faut replacer HEAD sur la tête de l'historique enregistré dans le dépôt : git checkout main (5)

Captures résultats

3

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git checkout e085932
Note: switching to 'e085932'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at e085932 suppression du fichier2
```

4

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit ((e085932...))
$ git lg
e085932 (HEAD) suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

5

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit ((e085932...))
$ git checkout main
Previous HEAD position was e085932 suppression du fichier2
Switched to branch 'main'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
262d3d2 (HEAD -> main) refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Restaurer un fichier

Avec git checkout, vous pouvez aussi récupérer un état particulier d'un fichier donné.

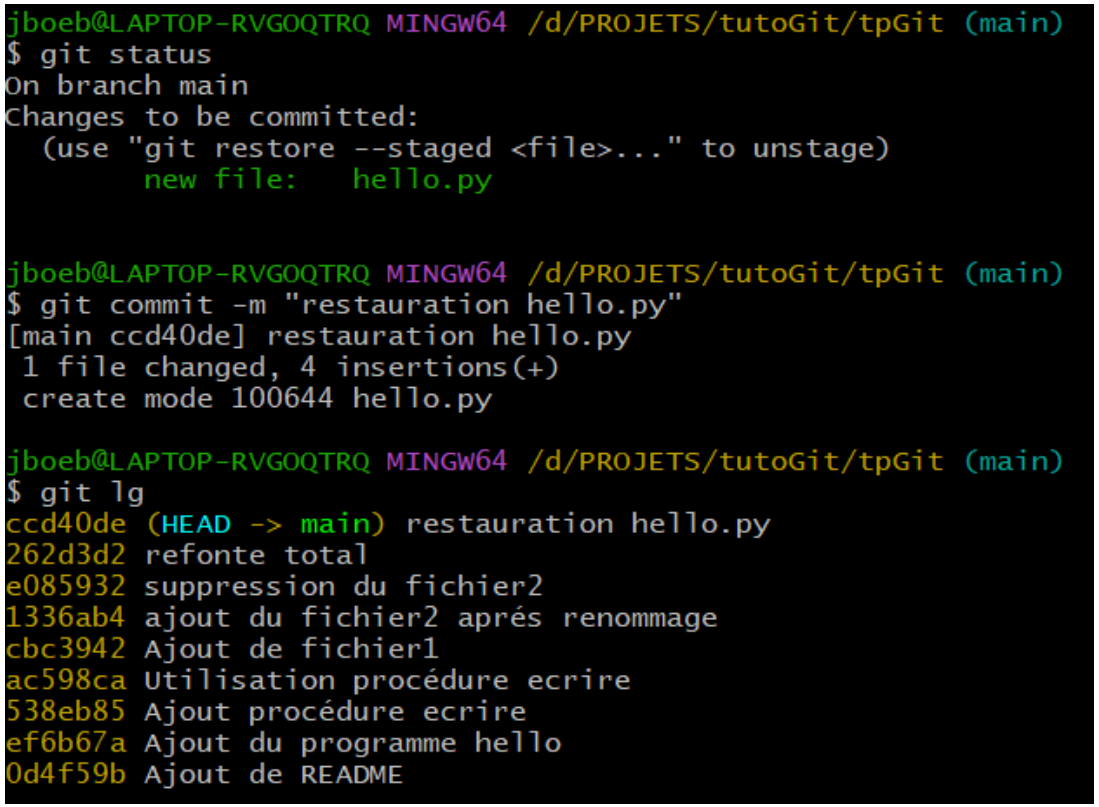
Cela va nous permettre de *récupérer* notre fichier **hello.py**, en utilisant (à adapter à votre historique)

- Git checkout n°Commit hello.py
  - Littéralement, cela signifie : met dans le répertoire de travail le fichier hello.py dans l'état où il était au commit.

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git checkout e085932 hello.py
Updated 1 path from f4520dc

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
262d3d2 (HEAD -> main) refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Restaurer un fichier

- git status montre que la commande précédente a mis la modification dans l'index. Vous pouvez maintenant faire un nouveau commit pour restaurer complètement le fichier avec le message "Restauration hello.py".
- *git checkout permet également de retrouver un commit qui a introduit un problème.*



Annuler des commits

Vous décidez, après réflexion, que le commit restaurant hello.py n'était pas une bonne idée.

- Il faudrait donc supprimer ce commit de l'historique. Cette opération peut être réalisée avec la commande git reset :
- Git reset HEAD^
  - ***Rappel : HEAD^ référence le commit précédant HEAD.***
- Git status montre que hello.py est toujours présent dans votre espace de travail mais qu'il est maintenant non suivi.

Captures résultats

⑧

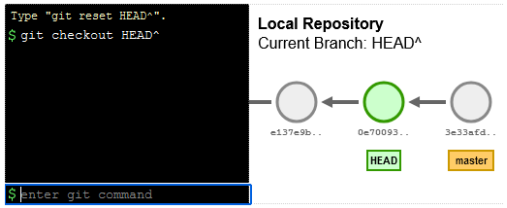
```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
ccd40de (HEAD -> main) restauration hello.py
262d3d2 refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git reset HEAD^

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello.py

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

Git reset

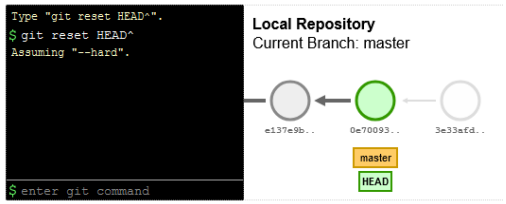
À l'aide du simulateur explainGit, vous pouvez observer la différence entre la commande git checkout que nous avons déjà vue, et la commande git reset

https://onlywei.github.io/explain-git-with-d3/#commit

1. git checkout HEAD^ : déplacement de la référence HEAD mais sans toucher à l'historique enregistré (et synchronisation du répertoire de travail)



2. git reset HEAD^ : déplacement de la référence de la tête de l'historique (main), ce qui force un déplacement de HEAD (mais sans modification du répertoire de travail)



git reset va donc déplacer en arrière la référence qui pointe sur le dernier commit de l'historique (le main). C'est comme si ce commit n'avait jamais été effectué. Avant ce commit, le fichier hello.py n'existait plus, il n'était donc plus suivi par Git. C'est bien dans cet état que nous sommes revenu

Attention au reset --hard!

Le mode --hard de git reset permet d'aller plus loin dans le retour en arrière, en combinant le git reset avec une synchronisation du répertoire de travail (comme le fait git checkout).

Il y a cependant une différence essentielle avec git checkout : la synchronisation est forcée, toutes les modifications sur les fichiers suivis sont supprimées (git checkout ne peut pas être exécuté si il existe des modifications non validées).

Pour tester ce mode :

1. Re-validez hello.py en créant un nouveau commit. (9)
2. Utilisez le mode --hard : git reset --hard HEAD^ (10)
3. Constatez que les modifications sont perdues (**le fichier hello.py a disparu !**)

Ce mode est à utiliser en connaissance de cause : c'est l'un des rares cas où vous pouvez perdre des données de manière irréversible !

Captures résultats

9

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git add hello.py

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit -m "2éme restauration hello.py"
[main 5dd7d8a] 2éme restauration hello.py
 1 file changed, 4 insertions(+)
 create mode 100644 hello.py

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git status
On branch main
nothing to commit, working tree clean

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
5dd7d8a (HEAD -> main) 2éme restauration hello.py
262d3d2 refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

10

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git reset --hard HEAD^
HEAD is now at 262d3d2 refonte total

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
262d3d2 (HEAD -> main) refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Supprimer plusieurs commits

git reset permet aussi de supprimer plusieurs commits.

Pour cela il y a plusieurs possibilités

(ne les essayez pas, nous avons besoin de conserver le dépôt dans son état actuel pour la suite…).

- Git reset HEAD^^^^^
  - Supprimer les 5 derniers commits
- Git reset HEAD~5
  - Ecriture compressée
- Git reset f1fa39c
  - Supprimer tous les commits après le commit ayant l'id
- Git reset f1fa39v^
  - Supprimer le commit ayant l'id et ceux qui le suivent (donc revenir au commit avant l'id)

A retenir

Les commandes git checkout et git reset pourraient sembler équivalentes de prime abord puisqu'elles peuvent produire le même résultat apparent sur votre répertoire de travail.

- git checkout agit sur le répertoire de travail : le répertoire de travail est restauré dans l'état enregistré dans le commit sélectionné.
  - Pour cela, Git peut modifier vos fichiers, et en ajouter ou supprimer d'autres (Git ne touche cependant pas aux fichiers non suivis, fort heureusement).
  - Cette commande ne modifie pas l'historique enregistré dans le dépôt (la référence main n'est pas modifiée). On peut donc revenir à la dernière version avec git checkout main.
- git reset agit sur l'historique : la référence de tête (main) est déplacée, ce qui supprime des commits de l'historique.
  - Avec l'option --soft
    - le contenu du répertoire de travail n'est pas modifié : les différences entre le répertoire de travail et le nouveau commit de tête sont considérées comme des modifications placées dans l'index, mais non encore validées.
    - ***Résultat*** : on revient juste au moment où l'on avait fait le git add mais sans avoir encore fait le commit.
  - Dans sa version de base (option --mixed par défaut)
    - le contenu du répertoire de travail n'est toujours pas modifié : les différences entre le répertoire de travail et le nouveau commit de tête sont considérées comme non indexées.
    - ***Résultat*** : on revient un cran plus tôt, juste au moment où l'on avait modifié les fichiers mais sans avoir encore fait le git add.
  - Dans sa version --hard
    - le répertoire de travail est également synchronisé avec le nouveau commit de tête.
    - ***Résultat*** : on revient un cran de plus en arrière, avant le moment où l'on avait modifié les fichiers.

A retenir

Cette différence entre reset et checkout fait que leur fréquence d'utilisation est très dissemblable.

ATTENTION : Comme pour toute commande modifiant l'historique, git reset ne doit pas être utilisé sur des historiques partagés avec d'autres contributeurs.

- git checkout <commit_id> est une commande utilisée très régulièrement lorsque l'on travaille avec des branches, comme vous le verrez bientôt. C'est l'usage principal de cette commande.
- git checkout <un_fichier> est occasionnellement utilisée pour supprimer des modifications que l'on vient d'ajouter à un fichier.
- git reset est rarement utilisée. Elle ne sert, a priori, que lorsque l'on décide d'abandonner un travail en cours et de revenir en arrière.

Point d'étape du TP

Après la manipulation du git reset, nous revalidons notre fichier hello.py.

Vous devriez obtenir un état de votre dossier Git suivant :

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
ff7a8b5 (HEAD -> main) Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Remonter le temps (2) avec revert

Dans le chapitre Revenir en arrière (1), nous avons vu une première méthode permettant de *renverser* un commit.

- le fichier hello.py supprimé lors d'un commit a été retrouvé avec git checkout, puis remis dans le dépôt avec un git commit.

- Cette manipulation consiste donc à créer un commit qui inverse les modifications introduites par un ancien commit, afin qu'au final elles soient annulées dans la version courante du dépôt.
  - C'est une opération classique, qui porte le nom de revert en anglais.
- Git dispose de la commande git revert pour effectuer en une seule opération ce que nous avions fait dans Revenir en arrière (1).

1. Effectuer Git revert <id>
   1. Git va créer un nouveau commit et vous demande donc d'écrire le commentaire associé (prenez le commentaire proposé par défaut) 1

Captures résultat

1

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
ff7a8b5 (HEAD -> main) Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git revert ff7a8b5
[main 6c29431] Revert "Refonte total"
 1 file changed, 4 insertions(+)
 create mode 100644 hello.py
```

Git log après revert

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
6c29431 (HEAD -> main) Revert "Refonte total"
ff7a8b5 Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Amender le dernier commit

Le dernier commit restaure donc le fichier hello.py.

Ce commit ne vous convient cependant pas totalement parce que vous aviez également mis le fichier README.md à jour pour indiquer que vous étiez en cours de refonte de votre projet.

1. Commencez par modifier votre README.md pour commenter différemment la refonte en cours.
2. git status vous indique qu'il y a une modification non encore validée. Pour différentes raisons, vous voulez ajouter cette modification directement dans le même commit que celui qui renverse la suppression de hello.py.
   - *Avec ce que vous connaissez pour l'instant, il faudrait supprimer le dernier commit pour le recréer en utilisant la suite de commandes suivantes (ne le faîte pas) :*
     1. *git reset HEAD^*
     2. *git add hello.py README.md*
     3. *git commit (avec un nouveau commentaire)*
3. Ce type d'opération consistant à **amender** le dernier commit est assez standard, et Git propose de la réaliser en une seule action avec l'option '--amend' de git commit.

   Ajoutez d'abord README.md à l'index, puis exécutez le commit avec amendement :

   1. git add README.md
   2. git commit --amend
   3. *Ajouter le message dans l'éditeur : "AMEND README + Revert "Refonte total"*

2

4. modifier le message du dernier commit. constatez avec git lg que Git a remplacé le dernier commit (son identifiant est différent) avec git show HEAD constatez que la modification de README.md a bien été prise en compte en plus de la réintégration de hello.py.

3

Captures résultats

2

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ 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:   README.md

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git add README.md

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git commit --amend
[main 29f161b] AMEND README + Revert "Refonte total"
 Date: Wed Sep 11 16:50:23 2024 +0200
 2 files changed, 5 insertions(+)
 create mode 100644 hello.py
```

3

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
29f161b (HEAD -> main) AMEND README + Revert "Refonte total"
ff7a8b5 Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git show HEAD
commit 29f161b921ec9145aa03be176891a4fd887fab57 (HEAD -> main)
Author: neojero <jboebion@gmail.com>
Date:   Wed Sep 11 16:50:23 2024 +0200

    AMEND README + Revert "Refonte total"

    This reverts commit ff7a8b5d2e9215cd009f8c0b1fa7b6638fb649b3.

diff --git a/README.md b/README.md
index e69de29..cd88fc4 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+# refonte du projet
\ No newline at end of file
diff --git a/hello.py b/hello.py
new file mode 100644
index 0000000..80a5a17
--- /dev/null
+++ b/hello.py
@@ -0,0 +1,4 @@
+def ecrire(chaine):
+    print(chaine)
+
+ecrire("hello World")
\ No newline at end of file
```

Cas fréquents d'amendement

Il y a 2 cas fréquents qui amènent à devoir amender le dernier commit :

1. Vous avez oublié d'ajouter les modifications d'un fichier dans le commit :
   - Faîtes le git add nécessaire, puis un git commit --amend
2. Vous voulez modifier, après coup, le commentaire du commit :
   - Faîtes un git commit --amend

- Il existe un autre cas, plus rare : il est possible lorsque vous faîtes un commit de spécifier le nom de l'auteur des modifications (si ce n'est pas vous) à l'aide de la commande :
  - git commit --author="son nom<son@adresse.mail>".
- Assez souvent dans ce cas, on oublie (par habitude de faire un simple git commit) de spécifier le nom de l'auteur.
  - On peut alors corriger cet oubli par amendement : git commit --amend --author="son nom<son@adresse.mail>"

TRAVAUX PRATIQUE GIT

Les dépôts distants

Partager son dépôt

Git est un VCS distribué :

- Cela signifie que tous les collaborateurs au projet possèdent une copie du dépôt sur leur machine.
- Centraliser sur un serveur (par exemple) un dépôt qui fera foi n'est pas une obligation. Tout dépend de votre organisation de travail.

- Il est à noter qu'en conséquence le dépôt central n'est pas un point critique du système. En particulier, une sauvegarde n'est pas indispensable (mais reste une bonne pratique !) puisque chaque collaborateur possède une copie.
- Attention ! Il ne faut pas modifier l'historique des modifications qui ont été partagées. Par exemple ne faites pas de git reset.
  - Vos collaborateurs auront peut-être commencé à faire des modifications à partir d'un commit que vous avez supprimé. Si vous voulez annuler un commit, préférez git revert.

Partage par simple copie

Une manière simple de partager son dépot et de le copier sur un support (clé USB, par exemple), de créer une archive Zip et de le transmettre.

Cela reste à la marge, car nous avons entre les mains un outil pour partager, travailler de manière collaborative.

- En décompressant le dossier et en effectuant un git log, vous constaterez que vous venez de créer un clone
- Vous pouvez aussi faire la même chose avec la commande git clone :
  - Git clone tpgit tpgit-clone

Utiliser un dépôt distant et clonage

Pour collaborer sur un projet, on utilise un dépôt dît de référence, hébergé sur un serveur Git tel que GitHub ou GitLab pour ne citer qu'eux.

1. Créer un dépôt sur GitHub
2. Lier votre dépôt distant avec votre dépôt local avec git remote
3. Pousser votre dépôt local vers le dépôt distant avec git push

1

4. Créer un nouveau dossier tutoGit-clone
5. Cloner votre dépôt distant dans ce nouveau dossier
    1. Une copie du dépôt tutoGit du serveur est alors créée dans un répertoire local tutoGit-clone
        - *par défaut, Git créé un répertoire en réutilisant le nom du dépôt mais sans le suffixe .git*
    2. Entrez dans ce répertoire tutoGit cloné et vérifiez avec git log que tout l'historique est bien présent sur votre machine.
    3. Vérifiez qu'un lien vers le dépôt distant a été créé : git remote -v

2

        - git clone crée par défaut un lien nommé 'origin'.
        - *Bien entendu, dans notre cas cette opération n'était pas réellement nécessaire puisque nous avions déjà une version locale du dépôt...*

Captures résultats

1

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git branch -M main

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git remote add origin https://github.com/neojero/tutoGit.git

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git push -u origin main
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Delta compression using up to 12 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (22/22), 2.08 KiB | 2.08 MiB/s, done.
Total 22 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/neojero/tutoGit.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

2

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit-clone/tutoGit (main)
$ git remote -v
origin  https://github.com/neojero/tutoGit.git (fetch)
origin  https://github.com/neojero/tutoGit.git (push)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit-clone/tutoGit (main)
$ git lg
29f161b (HEAD -> main, origin/main, origin/HEAD) AMEND README + Revert "Refonte
total"
ff7a8b5 Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Travail collaboratif

Simulation de travail en équipe

Travail collaboratif

Pour la suite, vous allez travailler en équipe idéalement en binôme pour simuler un développement collaboratif.

Choisissez si votre projet Git ou celui de votre collègue servira de dépôt référence (c'est à dire de dépôt partagé entre vous).

- S'il s'agit de votre projet, alors vous en avez déjà créé un clone.
- S'il s'agit de celui de votre binôme, alors il vous faut en créer une copie en local sur votre machine.

- Modifications collaboratives (1)
  - Vous allez maintenant à tour de rôle faire une modification.
  - Le premier membre,
    - modifie le fichier README.md
    - crée un commit avec le message "Collaboration action 1"
    - puis pousse vers le dépôt distant.

Collaborateur 1

Travail collaboratif

Collaborateur 2

Que s'est-il passé jusqu'à présent ?

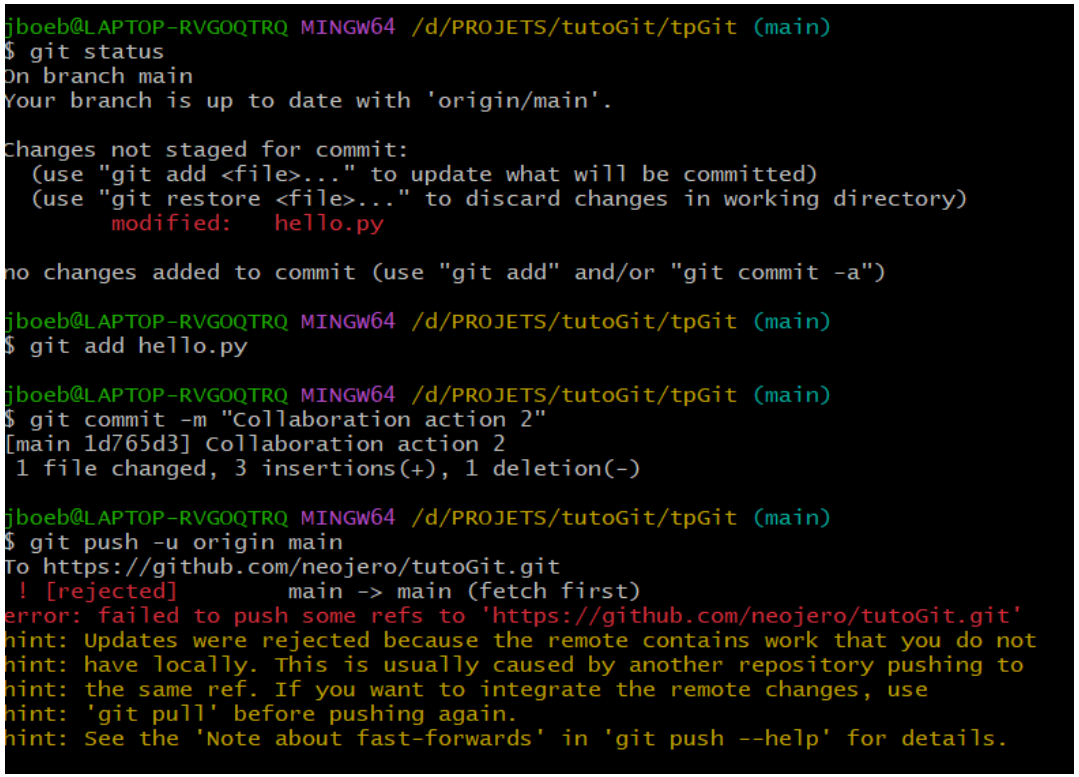
- Lorsque vous avez poussé votre dépôt local vers votre dépôt distant Git a créé une copie exacte de votre historique
  - les mêmes commits avec les mêmes identifiants sont dans le dépôt hébergé sur le serveur.
- Créez un commit : git commit.
  - **Résultat***:* Il y a maintenant un commit supplémentaire dans votre historique local.
- Effectuez un git push.
  - **Résultat**: Git compare votre historique local avec l'historique distant (en se basant sur les identifiants des commits). Par rapport à la tête (main) de l'historique distant, il y a un commit supplémentaire dans votre historique local. Il est recopié tel quel, et la tête de l'historique distant est modifiée.

Modifications collaboratives (2)

- De son côté, le deuxième membre du binôme
  - ajoute un commentaire au fichier hello.py,
  - crée un commit avec le message "Collaboration action 2"
  - Et le pousse.
- Que se passe-t-il selon vous ? (1)

Pourquoi Git refuse-t-il de le faire ?

- Le dépôt distant compte un commit que vous n'avez pas en local. Git ne sait donc pas quoi faire avec le commit que vous voulez pousser : écraser le commit existant sur le dépôt distant ? Fusionner les 2 commits et régler les conflits ?
- Non, Git ne veut/peut pas prendre ce genre de décision.



1

Pull puis Push

Il faut donc que vous récupériez d'abord les changements du dépôt distant et que vous régliez vous-même les éventuels problèmes.

- Git pull
  - Un nouveau commit qui fusionne les 2 modifications (celles du commit distant et celles de votre commit local) va être créé. Vous êtes invité à modifier (ou non) son message. (Prendre le message par défaut)
- Git push
  - Cette fois-ci, vous pouvez pousser votre modification dans le dépôt distant

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 310 bytes | 77.00 KiB/s, done.
From https://github.com/neojero/tutoGit
   29f161b..0587325  main       -> origin/main
Merge made by the 'ort' strategy.
 README.md | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git push -u origin main
Enumerating objects: 9, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 12 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 639 bytes | 639.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/neojero/tutoGit.git
   0587325..a85b157  main -> main
branch 'main' set up to track 'origin/main'.

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
a85b157 (HEAD -> main, origin/main) Merge branch 'main' of https://github.com/neojero/tutoGit
1d765d3 Collaboration action 2
0587325 Collaboration action 1
29f161b AMEND README + Revert "Refonte total"
ff7a8b5 Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Explications

Tout comme Git gère une référence vers la tête de l'historique local (main), il gère également une référence vers ce qu'il pense être la tête de l'historique distant (origin/main).

- Avoir cette référence permet à Git de pouvoir exécuter beaucoup d'opérations sans avoir besoin de se connecter au serveur distant. C'est toute sa force !
- Cette référence n'est mise à jour que lorsque vous utilisez les commandes distantes (pull, push et fetch).

- un git commit.
  - **Résultat** : Ce commit est ajouté à la fin de votre historique local (main), qui n'est donc maintenant plus cohérent avec le dépôt distant.
- un git push (il ne se passe rien...).
  - **Résultat** : Git se rend compte qu'il y a une différence entre votre origin/main et le vrai dernier commit distant. Il faut d'abord procéder à une synchronisation...
- un git pull
  - **Résultat** : Git récupère d'abord le commit distant supplémentaire, qu'il attache à votre origin/main et il déplace la référence. Vous avez alors en local en copie exacte de l'historique distant. Une divergence a été introduite dans votre historique local, que Git résout en procédant à une fusion avec votre commit (celui référencé par main). Un nouveau commit est créé dans votre historique, résultat de cette fusion.
- un git push.
  - **Résultat** : Le dépôt distant est mis à jour,

Travail collaboratif

Modifications collaboratives (3)

- Pour que tous les dépôts soient synchronisés, il reste à faire un git pull sur l'autre machine.
- Chaque collaborateur peut contrôler avec un git log que leur historique est identique.

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit-clone/tutoGit (main)
$ git pull
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 5 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (5/5), 619 bytes | 103.00 KiB/s, done.
From https://github.com/neojero/tutoGit
   0587325..a85b157  main       -> origin/main
Updating 0587325..a85b157
Fast-forward
 hello.py | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit-clone/tutoGit (main)
$ git lg
a85b157 (HEAD -> main, origin/main, origin/HEAD) Merge branch 'main' of https://
github.com/neojero/tutoGit
1d765d3 Collaboration action 2
0587325 Collaboration action 1
29f161b AMEND README + Revert "Refonte total"
ff7a8b5 Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

TRAVAUX PRATIQUE GIT

Les branches

Les branches

Une branche est une succession de commits, avec une référence nommée désignant la tête de cette succession.

Lorsque que l'on initialise un dépôt Git, la branche par défaut se nomme main.

- Lorsqu'on veut faire évoluer notre projet, cela peut avoir des effets de bord non désirés et créer des bugs. Nous voudrions pouvoir garder intacte la version stable courante tant que nous n'avons pas testé correctement notre nouvelle version. Il nous faut une branche, qui soit gérée séparément de la version stable.
- Une bonne pratique est ainsi de créer une nouvelle branche pour chaque évolution, une nouvelle fonctionnalité, une amélioration ou une correction de bug. Cela donne une grande souplesse et tranquillité d'esprit. Si l'évolution s'avère inutile ou même catastrophique, ce n'est pas grave, il suffira d'abandonner la branche et de la supprimer sans que cela ait d'impact sur la branche contenant la version stable.
- Git offre des outils puissants permettant de gérer les évolutions sous forme de branches. Une branche est rattachée à un commit donné, point de départ du travail sur l'évolution. Ce commit peut faire partie de la branche main ou de toute autre branche.
- Un historique complet est sous forme d'arbre, ou plus exactement d'un graphe acyclique direct (Direct Acyclic Graph en anglais).

Création d'une branche

L'utilisation des branches est utile même dans un contexte local.

Nous allons donc reprendre notre dépôt initial.

- Nous allons commencer par un peu de ménage dans notre dépôt en supprimant (option --hard pour vraiment tout oublier) tous les commits suivant celui qui intègre l'utilisation de la procédure ecrire(). 1
- Vous devriez avoir avant reset un historique qui ressemble à :

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
a85b157 (HEAD -> main, origin/main) Merge branch 'main' of https://github.com/neojero/tutoGit
1d765d3 Collaboration action 2
0587325 Collaboration action 1
29f161b AMEND README + Revert "Refonte total"
ff7a8b5 Refonte total
e085932 suppression du fichier2
1336ab4 ajout du fichier2 aprés renommage
cbc3942 Ajout de fichier1
ac598ca Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Capture résultats

Git reset --hard <id>

1

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git reset --hard ac598ca
HEAD is now at ac598ca Utilisation procédure ecrire

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
ac598ca (HEAD -> main) Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

Création d'une branche

Ensuite, nous allons créer une nouvelle branche nommée wip

- Git branch wip
  - Nous obtenons l'historique suivant :

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git branch wip

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg
ac598ca (HEAD -> main, wip) Utilisation procédure ecrire
538eb85 Ajout procédure ecrire
ef6b67a Ajout du programme hello
0d4f59b Ajout de README
```

  - Nous avons simplement créé une nouvelle référence, nommée wip, placée pour l'instant sur le dernier commit (celui pointé par HEAD).

Se positionner sur une branche

Conceptuellement, se placer sur une branche, c'est faire en sorte que notre répertoire de travail soit en lien avec le dernier commit de cette branche.

Nous savons que ce lien est défini par la position de la référence HEAD et nous connaissons la commande qui permet de manipuler cette référence :

- c'est la commande git checkout

- Pour se placer dans la branche wip, il suffit donc de déplacer HEAD sur le commit référence par wip
  - Git checkout wip
  - Constatez l'historique de votre dépôt ?
    - La référence HEAD pointe maintenant effectivement sur wip.
    - **Résultat** : tout nouveau commit sera ajouté après wip et non plus après main. **C'est cela qui va créer une branche, une succession séparée de commit.**

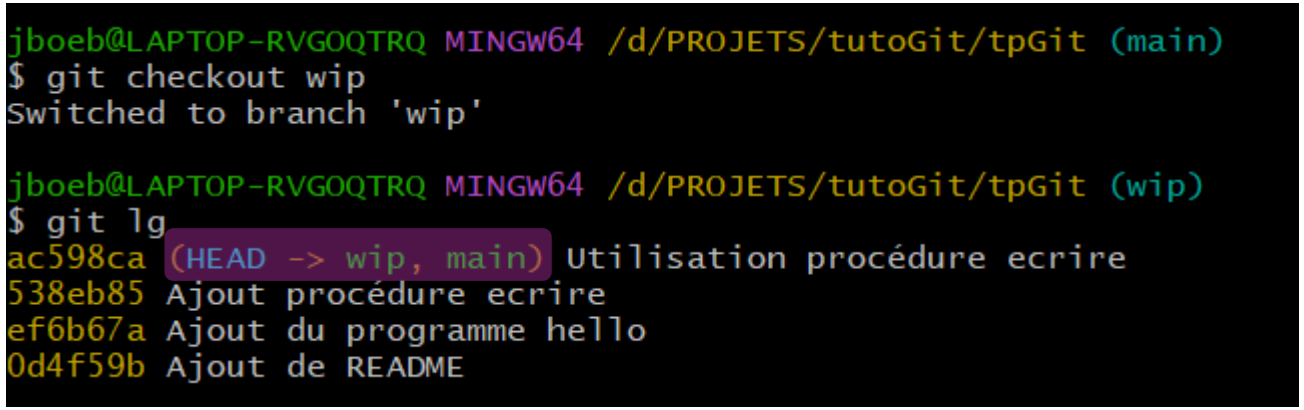
2

  - *Note : vous pouvez effectuer les deux actions en une seule commande (création + déplacement) :*
    - *Git checkout -b wip*

Captures résultats

Vous pouvez constater que le HEAD pointe désormais sur wip et plus sur la branche main

2



Explications (ne pas exécuter)

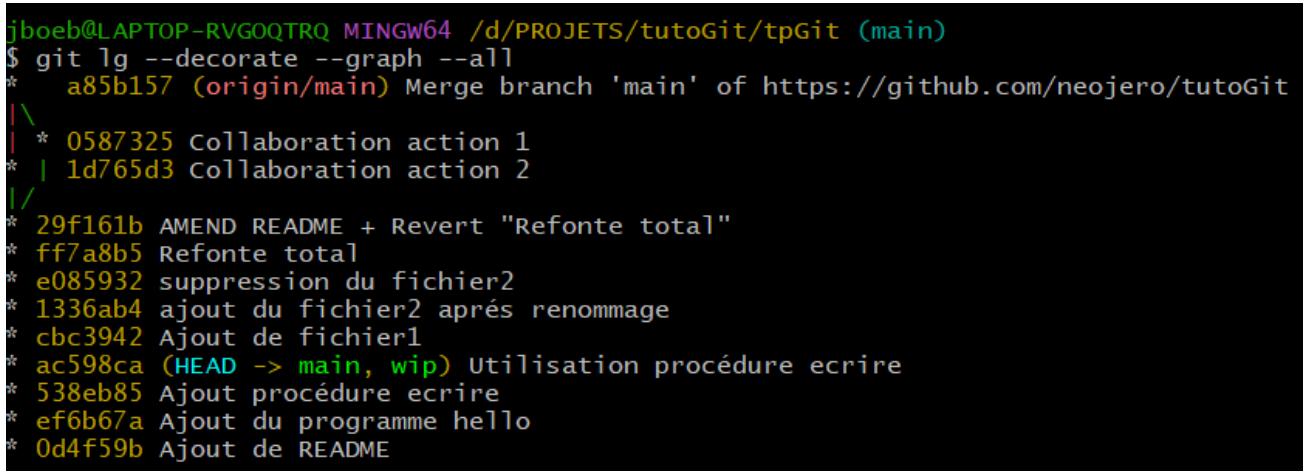
Pour bien comprendre…

- N'oubliez pas qu'en plus de déplacer la référence HEAD, l'objectif de la commande git checkout est surtout de modifier le contenu du répertoire de travail pour qu'il reflète le contenu du commit pointé par HEAD.
- Ainsi, en changeant de branche vous accédez à des versions de vos fichiers pouvant être totalement différentes, et ce extrêmement rapidement comme vous aurez bientôt l'occasion de le tester.

- git branch wip crée une nouvelle référence qui pointe sur le commit référence par HEAD (qui lui pointe sur main)
- git checkout wip fait pointer HEAD sur wip.
- git commit
  - ajoute un commit après le HEAD, et donc fait progresser la référence wip (toujours pointé par HEAD).
- git commit
  - nouveau commit sur wip
- git checkout main
  - fait revenir HEAD sur le commit référencé par main

Visualisation avancée de l'historique

- Visualisation avancée de l'historique :
  - Pour visualiser l'arbre des commits, vous pouvez utiliser git log avec l'option --graph.
  - Et pour voir toutes les branches, vous pouvez utiliser l'option --all.



Fusion de branche

Lorsque vous avez terminé le travail pour lequel vous avez créé une branche, il est nécessaire de reporter les modifications effectuées sur la branche principale (la branche main).

Cela se fait par une opération de fusion (merge en anglais) et Git est très puissant de ce point de vue.

- La solution la plus légère sera toujours prise. Cela revient souvent à un simple déplacement d'étiquette et à la création d'un nouveau commit de fusion.
- Parfois, ce sera plus compliqué, il faudra résoudre des conflits, mais git vous rendra la tâche plus facile en vous guidant pas à pas.

- Fusion fast-forward
- Le premier cas de fusion que nous allons voir est le plus simple puisqu'aucun nouveau commit ne sera créé !
    1. Sur la branche wip, qui a été créée, plusieurs commits ont été réalisés. La branche main, quant à elle, n'a pas évoluée.
    2. Les commits de la branche wip sont donc tous situés dans une séquence qui suit la référence main.
    3. **Pour fusionner une branche A sur une branche B, il faut d'abord se placer sur la branche B. Nous sommes donc revenus sur la branche main sur laquelle nous voulons fusionner la branche wip**
    4. L'opération de fusion : git merge wip.

Mise en pratique (1)

Nous allons ajouter une fonctionnalité très utile à notre code :

- la possibilité d'écrire plusieurs fois un texte.

- Vérifiez que vous êtes sur la branche wip à l'aide de la commande git branch
- Editer et Ajouter la fonction suivante à hello.py

```
def ecrireXFois(x, chaine):
    for i in range(x):
        ecrire(chaine)
```

- Commiter la modification
  - Message "Ajout procédure ecrireXFois" 3
- Retourner que la branche main
- Affichez l'historique avec l'option --all 4
- Editez le contenu de hello.py
  - Que constatez-vous ?

Captures résultats

3

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git checkout wip
Switched to branch 'wip'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git branch
  main
* wip

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.py

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.py

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git commit -m "Ajout procédure exrireXFois"
[wip a0786d1] Ajout procédure exrireXFois
 1 file changed, 5 insertions(+)
```

4

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git checkout main
Switched to branch 'main'
Your branch is behind 'origin/main' by 8 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg --decorate --graph --all
* a0786d1 (wip) Ajout procédure exrireXFois
| *   a85b157 (origin/main) Merge branch 'main' of https://github.com/neojero/tutoGit
| |\
| | * 0587325 Collaboration action 1
| * | 1d765d3 Collaboration action 2
| |/
| * 29f161b AMEND README + Revert "Refonte total"
| * ff7a8b5 Refonte total
| * e085932 suppression du fichier2
| * 1336ab4 ajout du fichier2 aprés renommage
| * cbc3942 Ajout de fichier1
|/
* ac598ca (HEAD -> main) Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

Après le checkout, La fonction ecrireXFois() n'est plus là ! Git l'a supprimée ? Non, mais il fait ce que vous lui avez demandé, un 'checkout', c'est-à-dire qu'il a modifié le contenu de votre répertoire de travail pour qu'il reflète le dernier commit de la branche 'main'. Et sur cette branche, la nouvelle fonction n'existe pas dans hello.py !

Mise en pratique (1)

Pour vous rassurer, retournez sur la branche wip (git checkout wip) et vérifiez le fichier.

- Ouf, tout est là ! Retournez maintenant sur main.

- Il faut donc fusionner nos deux branches
  - Git merge wip 5
- Affichez l'historique avec l'option --all 6
  - la fusion n'a nécessité aucun nouveau commit (dù au fait que main n'a pas évolué pendant le développement de la nouvelle fonctionnalité).
  - Ce mode de fusion par simple déplacement de la référence de tête s'appelle un merge fast-forward (c'est ce que Git a indiqué dans son message de réponse [capture 5]). Seul l'étiquette main s'est déplacée.
  - C'est le comportement par défaut de git merge mais, au besoin, vous avez la possibilité de forcer la création d'un commit de fusion en utilisant l'option --no-ff.
    - L'intérêt est de conserver dans l'historique une trace du fait que certains commits ont été réalisés sur une branche.

Capture résultats

5

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git merge wip
Updating ac598ca..a0786d1
Fast-forward
 hello.py | 5 +++++
 1 file changed, 5 insertions(+)
```

6

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg --decorate --graph --all
* a0786d1 (HEAD -> main, wip) Ajout procédure exrireXFois
|   a85b157 (origin/main) Merge branch 'main' of https://github.com/neojero/tutoGit
| |\
| | * 0587325 Collaboration action 1
| * | 1d765d3 Collaboration action 2
| |/
| * 29f161b AMEND README + Revert "Refonte total"
| * ff7a8b5 Refonte total
| * e085932 suppression du fichier2
| * 1336ab4 ajout du fichier2 aprés renommage
| * cbc3942 Ajout de fichier1
|/
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

Fusion sans conflit

vous avez commencé à travailler sur une branche wip et effectué quelques commits.

- Pour répondre à un besoin urgent, vous êtes retourné sur la branche main (la branche stable) et vous avez réalisé un commit (pour corriger un bug, par exemple).

Vous êtes ensuite retourné sur wip pour continuer votre travail.

- Il est temps maintenant d'intégrer le résultat sur la branche main, et vous y retournez :
  - git merge wip
  - La branche wip ne faisant plus suite directement à la branche main, un fast-forward (un déplacement de référence) ne sera plus possible.
    - Git doit combiner les modifications de deux branches en une seule, c'est l'opération de fusion à proprement parler.
    - Un commit contenant le résultat de cette fusion est créé et ajouté à la branche main.
    - Ce commit à 2 parents. C'est la caractéristique des commits de fusion.

Mise en pratique (2)

Commençons par revenir en arrière dans l'historique avec un git reset --hard HEAD^, pour supprimer le 'merge fast-forward' que nous avions créé.

- **Interprétation** : nous sommes sur la branche main (HEAD pointe sur main) et il y a un commit supplémentaire sur la branche wip.
  - *Notez au passage qu'il n'a pas été supprimé, malgré l'exécution de la commande git reset. En effet, tant qu'un commit possède une référence (ici wip) il est visible dans l'historique.*
- Dans main, vous allez modifier le programme pour qu'il affiche le message en majuscules :

```
def ecrire(chaine):
    print (chaine).upper()
```

Mise en pratique (2)

1. Ajoutez cette modification dans l'index
2. commitez-la avec le message "Ajout majuscule procédure écrire"

- Vous pouvez voir la différence qui existe entre le contenu de la branche 'wip' et la vôtre à l'aide de git diff wip 7
- Comment interprétez-vous ce résultat ?
  - Dans votre branche main, par rapport à wip, la fonction ecrireXFois() n'existe pas (d'où les '-') et la ligne 'print' est différente. **C'est cohérent.**
  - Fusionnez la branche wip : git merge wip
    - Que constatez-vous ? 8

Captures résultats

7

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff wip
diff --git a/hello.py b/hello.py
index 3e0418c..f06d130 100644
--- a/hello.py
+++ b/hello.py
@@ -1,9 +1,4 @@
 def ecrire(chaine):
-    print(chaine)
-
-# procedure ecrire plusieurs fois
-def ecrireXFois(x,chaine):
-    for i in range(x):
-        ecrire(chaine)
+    print(chaine).upper()

 ecrire("hello World")
\ No newline at end of file
```

8

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git merge wip
Auto-merging hello.py
Merge made by the 'ort' strategy.
 hello.py | 5 +++++
 1 file changed, 5 insertions(+)
```

Mise en pratique (2)

Vous êtes invité à modifier le message du commit de fusion.

Le 'fast-forward' n'est pas possible dans ce cas puisque les 2 branches ont évolué parallèlement. **Il y a création d'un commit.**

- Par contre vous n'avez pas eu de conflit à régler.

En effet, les 2 modifications ne concernent pas les mêmes lignes, comme nous l'avons vu en faisant le git diff wip.

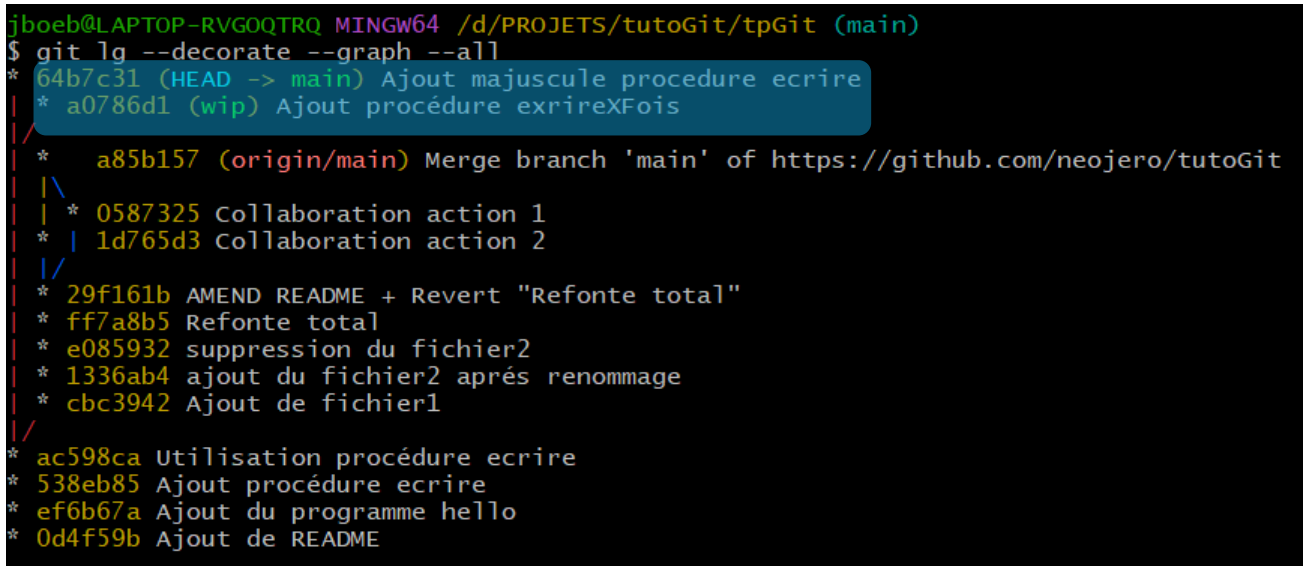
```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg --decorate --graph
*   19cdd9e (HEAD -> main) Merge branch 'wip'
|\
| * a0786d1 (wip) Ajout procédure exrireXFois
* | 64b7c31 Ajout majuscule procedure ecrire
|/
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

Fusion avec conflit

Effectuons un git reset --hard HEAD^ pour annuler notre commit de fusion.

Nous revenons à l'historique ci-contre.

- Nous avons donc bien un commit particulier sur chacune des deux branches.



Mise en pratique (3)

Nous allons maintenant provoquer volontairement un conflit, pour voir comment se comporte Git dans ce cas au moment de la fusion.

- Pour cela, nous allons modifier la même ligne du fichier hello.py dans les 2 branches.
  - Dans main, nous avons déjà modifiés la fonction ecrire() pour qu'elle affiche en majuscules.
  - Dans wip, effectuons donc un commit dans lequel la fonction ecrire() affiche en minuscules.

- Placez-vous dans wip et modifiez la procédure ecrire() comme cela :

```
def ecrire(chaine):
    print (chaine).lower()
```

Mise en pratique (3)

1. Commitez la modification avec le message "Ajout minuscule procédure ecrire)"
2. Retournez dans main
3. Effectuer un affichage de la différence des deux branches 9

- Tentez une fusion de la branche wip
  - Que se passe-t-il ? 10

- Git a détecté un conflit, et a arrêté l'opération de fusion.
  - git status vous le rappelle (le prompt Git fait de même avec l'indication '|MERGING')

Captures résultats

9

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git checkout main
Switched to branch 'main'
Your branch and 'origin/main' have diverged,
and have 1 and 8 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git diff wip
diff --git a/hello.py b/hello.py
index 2a6c0d0..f06d130 100644
--- a/hello.py
+++ b/hello.py
@@ -1,9 +1,4 @@
 def ecrire(chaine):
-    print(chaine).lower()
-
-# procedure ecrire plusieurs fois
-def ecrireXFois(x,chaine):
-    for i in range(x):
-        ecrire(chaine)
+    print(chaine).upper()

 ecrire("hello World")
\ No newline at end of file
```

Il faut résoudre le conflit pour pouvoir poursuivre la fusion. Pour cela, éditez le fichier hello.py comme Git vous y invite…

10

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git merge wip
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main|MERGING)
$ git status
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 8 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

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

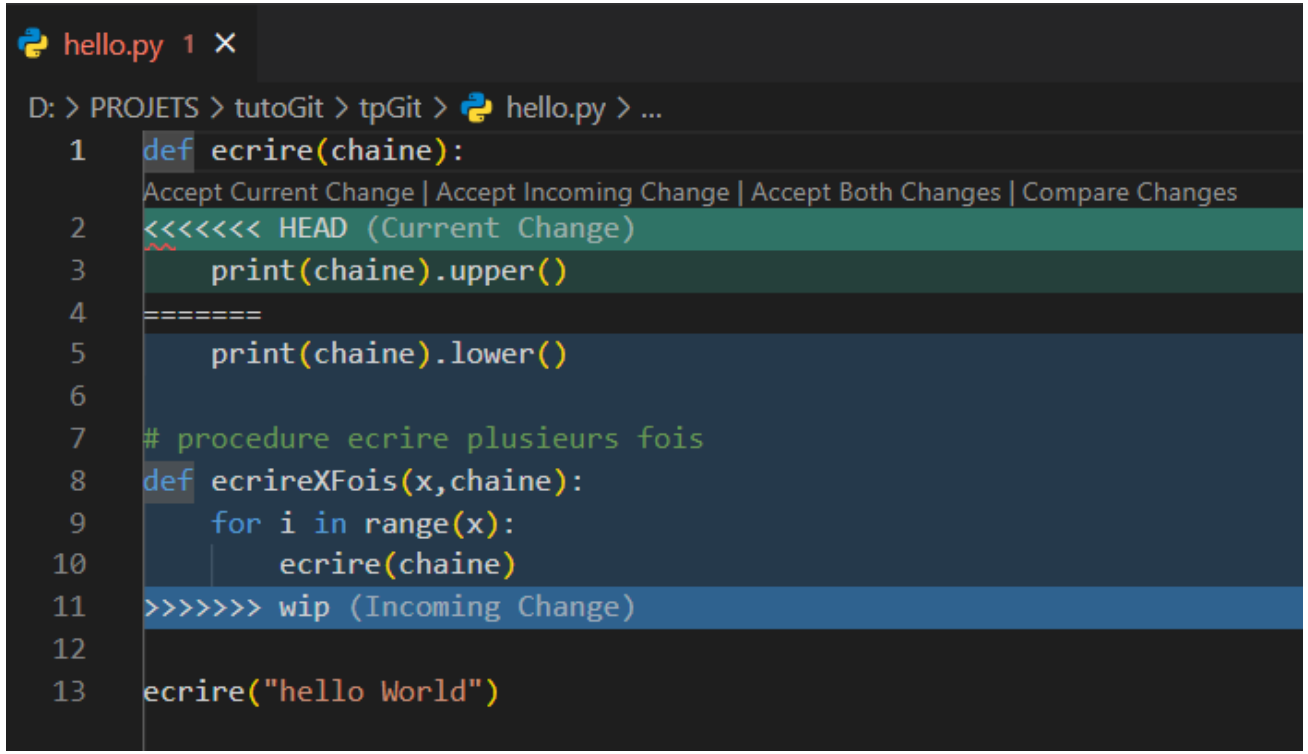
Mise en pratique (3)

Dans votre IDE, le balisage symbolise le conflit à résoudre.

- La première section concerne la modification de la branche courante (HEAD, qui pointe sur main) et la seconde, celle de wip.

Il suffit maintenant de supprimer le balisage de conflit et la modification que vous ne voulez pas garder. Par exemple, supprimer l'affichage en minuscules.

- Il suffit de cliquer dans l'IDE sur accept current change



Mise en pratique (3)

Il ne reste plus qu'a commiter les changements

- Git sait que l'on est en cours de fusion, il vous propose donc un message de commit par défaut (si vous le souhaitez).
- Une fois le commit réalisé, la fusion est terminée, et l'historique devient :

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main|MERGING)
$ git status
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 8 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

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

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main|MERGING)
$ git add hello.py

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main|MERGING)
$ git commit
[main e4de6ef] Merge branch 'wip'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg --decorate --graph --all
*   e4de6ef (HEAD -> main) Merge branch 'wip'
|\
| * ce11ce2 (wip) Ajout minuscule procedure ecrire
| * a0786d1 Ajout procédure exrireXFois
* | 64b7c31 Ajout majuscule procedure ecrire
|/
| *   a85b157 (origin/main) Merge branch 'main' of https://github.com/neojero/tutoGit
| |\
| | * 0587325 Collaboration action 1
| * | 1d765d3 Collaboration action 2
| |/
| * 29f161b AMEND README + Revert "Refonte total"
| * ff7a8b5 Refonte total
| * e085932 suppression du fichier2
| * 1336ab4 ajout du fichier2 aprés renommage
| * cbc3942 Ajout de fichier1
|/
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

Les branches

rebaser

Rebaser

Fusionnez avec Git c'est plutôt facile mais dans certains cas, il n'est pas souhaitable d'encombrer l'historique en gardant la trace de toutes les branches fusionnées.

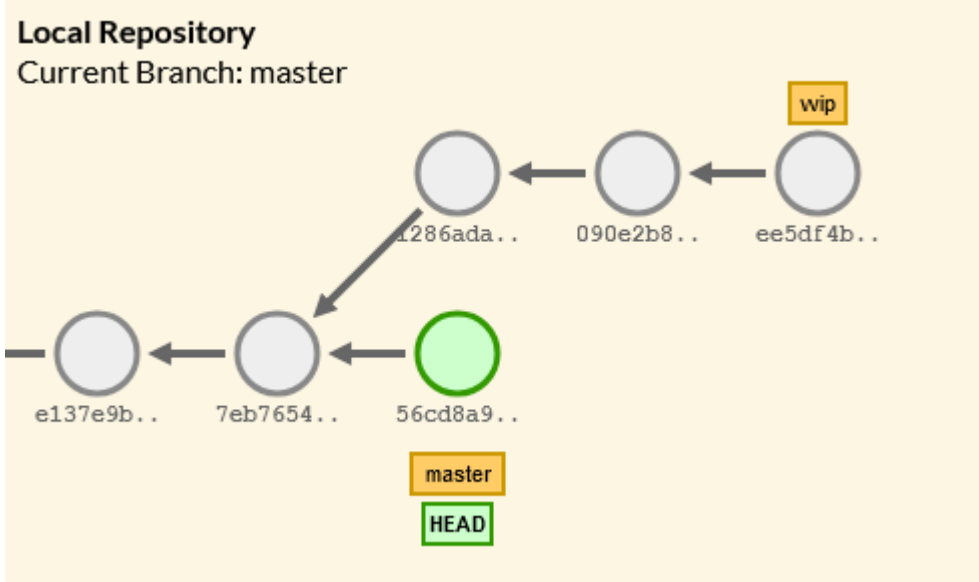
- L'idéal serait alors de faire, quel que soit le contexte, l'équivalent d'un merge fast-forward afin d'obtenir, après fusion, un arbre linéaire.

Avec Git, c'est possible malgré la multiplication des branches et les conflits grâce à git rebase.

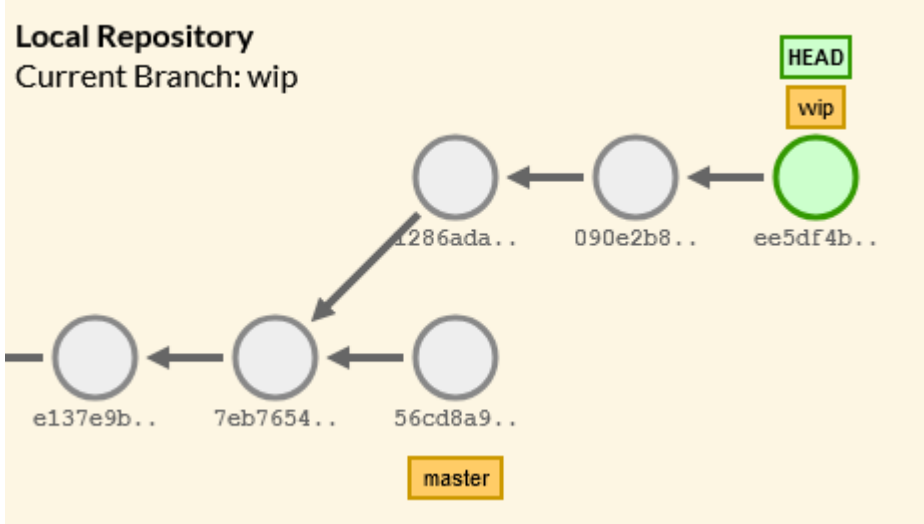
- Attention ! L'utilisation de git rebase implique la modification de l'historique. Il faut donc s'assurer que la partie que l'on affecte est bien locale.
- git rebase, comme son nom l'indique, change la base de la branche courante, c'est à dire le commit à partir duquel la branche a démarré.
  - La nouvelle base devient le dernier commit de la branche passée en paramètre de git rebase.
  - C'est pour cette raison que la branche que l'on rebase ne doit pas avoir été partagée (pas poussée).

Exemple dans cet exemple c'est master, mais avec main c'est pareil

Dépôt à l'origine

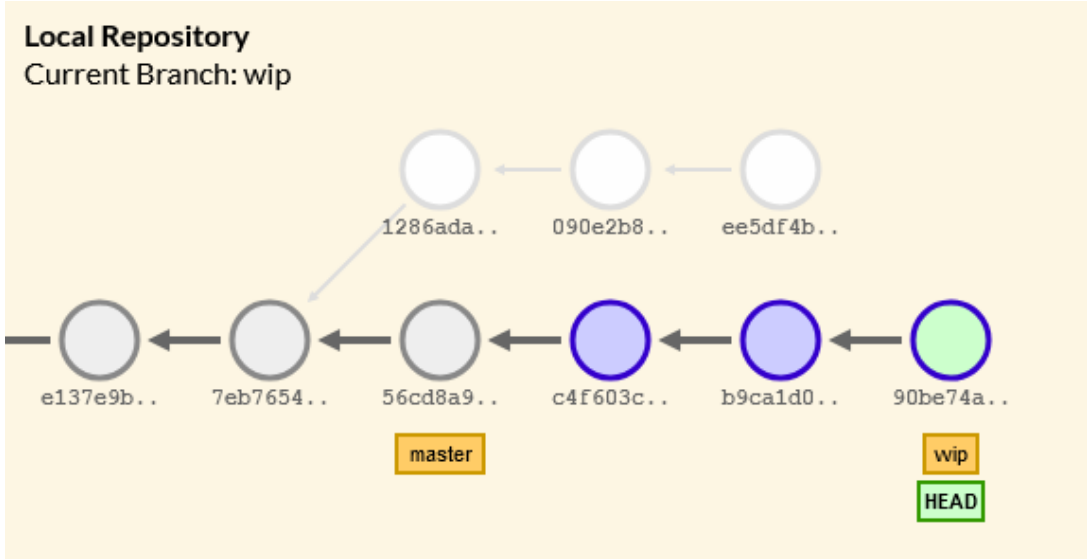


Déplacement du HEAD sur wip



Exemple

Rebase sur master



Conclusion

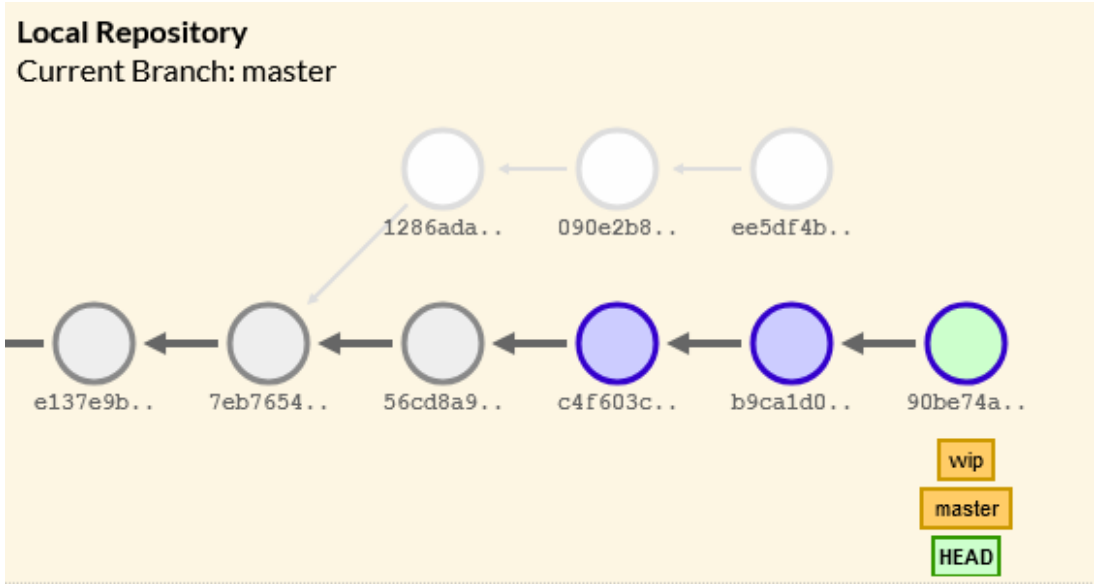
- Tout se passe comme si les commits de la branche wip avaient été déplacés à la fin de la branche main.
- En fait, si vous regardez bien le résultat, Git a appliqué tour à tour chacun des commits de la branche wip sur la tête de la branche main.
- Il a donc créé de nouveaux commits lors du rebase, ce que l'on observe par le fait que leurs identifiants sont différents.

Exemple

Finalisation

- La conséquence de cette opération est que des conflits peuvent éventuellement arriver lors du rebasage, conflits qu'il vous faudra résoudre.
  - Remarquez aussi que la référence wip s'est déplacée. Les anciens commits ne sont plus attachés à une référence. Le ramasse-miettes finira par les effacer automatiquement.
- On se retrouve donc maintenant dans le cas d'un merge fast-forward. Il n'y a plus qu'à fusionner wip dans master :
  - **git checkout main**
  - **git merge wip**

Checkout puis merge



Mise en pratique

Nous allons maintenant utiliser git rebase dans plusieurs cas réels.

Pour commencer, nous allons faire un peu de ménage dans notre dépôt.

1. En étant placé sur la branche main, annulez-le commit de fusion
    1. git checkout main,
    2. git reset --hard HEAD^ (1)
2. En étant placé sur la branche 'wip', on supprime le commit qui est source de conflit
    1. git checkout wip
    2. git reset --hard HEAD^ (2)
3. Retournez sur la branche main.

(1)

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git branch
* main
  wip

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git reset --hard HEAD^
HEAD is now at 64b7c31 Ajout majuscule procedure ecrire

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg --decorate --graph
* 64b7c31 (HEAD -> main) Ajout majuscule procedure ecrire
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

(2)

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git checkout wip
Switched to branch 'wip'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git reset --hard HEAD^
HEAD is now at a0786d1 Ajout procédure exrireXFois

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git lg --decorate --graph
* a0786d1 (HEAD -> wip) Ajout procédure exrireXFois
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

Rebaser sans conflit

Remarque utile pour la suite

- Vous pouvez à tout moment annuler un rebasage en cours en utilisant git rebase --abort.
- Vous reviendrez alors dans l'état où vous étiez avant d'exécuter la commande git rebase.

- Avant de faire la fusion de la branche wip sur la branche main, nous allons donc déplacer la base de wip au bout de la branche main. Placez-vous dans wip et effectuez le rebasage
  - Git checkout wip
  - Git rebase main (3)
- Git lg montre que nous avons maintenant un arbre linéaire de commits (et que l'identifiant du commit de la branche 'wip' est différent) (4)
- Nous pouvons retourner sur main et faire un "merge fast-forward en tapant :
  - Git checkout main
  - Git merge wip (5)

Captures résultats

3

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git checkout wip
Already on 'wip'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git rebase main
Successfully rebased and updated refs/heads/wip.
```

4

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git lg --decorate --graph
* a0db5af (HEAD -> wip) Ajout procédure exrireXFois
* 64b7c31 (main) Ajout majuscule procedure ecrire
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

5

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git checkout main
Switched to branch 'main'
Your branch and 'origin/main' have diverged,
and have 1 and 8 different commits each, respectively.
  (use "git pull" if you want to integrate the remote branch with yours)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git merge wip
Updating 64b7c31..a0db5af
Fast-forward
 hello.py | 5 +++++
 1 file changed, 5 insertions(+)

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (main)
$ git lg --decorate --graph
* a0db5af (HEAD -> main, wip) Ajout procédure exrireXFois
* 64b7c31 Ajout majuscule procedure ecrire
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

Rebaser avec conflit

Pour provoquer un conflit, nous allons maintenant modifier la même ligne du README.md dans les 2 branches.

1. Modifiez la première ligne du README.md dans main puis commitez.
2. Modifiez (avec un autre contenu) la première ligne du README.md dans wip puis commitez.
3. Afficher votre historique (6)
4. Tenter une rebasage de wip sur main (7)

Captures résultats

6

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git lg --decorate --graph --all
* 154c7fd (HEAD -> wip) rebase modification README 2
| * b0f0204 (main) rebase modification README 1
|/
* a0db5af Ajout procédure exrireXFois
* 64b7c31 Ajout majuscule procedure ecrire
| *   a85b157 (origin/main) Merge branch 'main' of https://github.com/neojero/tutoGit
| |\
| | * 0587325 Collaboration action 1
| * | 1d765d3 Collaboration action 2
| |/
| * 29f161b AMEND README + Revert "Refonte total"
| * ff7a8b5 Refonte total
| * e085932 suppression du fichier2
| * 1336ab4 ajout du fichier2 aprés renommage
| * cbc3942 Ajout de fichier1
|/
* ac598ca Utilisation procédure ecrire
* 538eb85 Ajout procédure ecrire
* ef6b67a Ajout du programme hello
* 0d4f59b Ajout de README
```

7

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git checkout wip
Already on 'wip'

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git rebase main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: could not apply 154c7fd... rebase modification README 2
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
hint: Disable this message with "git config advice.mergeConflict false"
Could not apply 154c7fd... rebase modification README 2
```

Rebaser avec conflit

Un conflit a donc été détecté dans README.md.

On est dans un cas similaire à une fusion avec conflit, que vous devez régler de la même façon

- Editer le fichier README.md qui contient un marquage de conflit.
- Conservez ce que vous voulez.
  - ***Remarque*** *: si vous choisissez de garder la version de main, en fait vous décidez d'oublier purement et simplement la modification qu'apporte le commit de wip que vous essayez d'appliquer.*
- git status vous rappelle ce que vous avez à faire
- utilisez git add pour indexer la résolution du conflit
- un nouveau git status vous indique comment poursuivre le rebasage :
  - git rebase --continue
  - ***Remarque*** *: Si, lors de la résolution du conflit, vous choisissez de garder la version de main, il n'y a de fait rien à ajouter à l'historique courant et Git vous propose de sauter le patch (c'est à dire de ne pas appliquer un commit qui n'apporte rien) avec un git rebase --skip.*
  - *Si vous changez d'avis, annulez le rebase avec git rebase --abort.*
- Constatez que votre arbre de commit est linéaire
- Vous pouvez retourner dans main pour exécuter un 'merge fast-forward' :
  - git merge wip

Les Etiquettes

tag

Etiquettes

Faire des branches pour protéger sa branche principale est une bonne pratique.

Cela permet d'avoir en permanence une version de référence (dont les tests passent) de votre projet.

- Cette version, considérée comme suffisamment stable, peut avoir fait l'objet d'une livraison.
- Il est souvent utile de marquer toutes les versions stables d'un projet au cours du temps en leur donnant un numéro : '1.3.25', 'v0.4', 'v0.8RC2'...
- Git permet de mettre une étiquette sur un commit grâce à la commande git tag.
  - Il existe 2 types d'étiquettes : les étiquettes légères et les étiquettes annotées.

Etiquettes légères

Pour étiqueter le commit courant,

- vous pouvez simplement utiliser :
  - Git tag vx.x
- Pour étiqueter une commit après coup :
  - Git tag vx.x <id_commit>
  - Cela crée une référence constante sur le commit donné. Elle est constante car elle ne peut pas être déplacée par git reset.
  - Vous pouvez néanmoins, en cas d'erreur, forcer la re-création de l'étiquette à l'aide de git tag -f <tag_name> <new_id_commit>
    - A ne pas utiliser si l'étiquette a été poussée sur un dépôt partagé

Etiquettes annotés

Une étiquette annotée est un objet de commit, c'est-à-dire une donnée contenant une date, un auteur et un commentaire.

- Création :
  - git tag -a v3.1 <id_commit>
- Affichage du contenu :
  - git show <tag_name>

```
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git tag -a v1.0 790c9dc
hint: Waiting for your editor to close the file...

jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/tutoGit/tpGit (wip)
$ git show v1.0
tag v1.0
Tagger: neojero <jboebion@gmail.com>
Date:   Thu Sep 12 14:59:24 2024 +0200

commit taggé

commit 790c9dc91be79a0d576940826fe95a8767041f1d (HEAD -> wip, tag: v1.0, main)
Author: neojero <jboebion@gmail.com>
Date:   Thu Sep 12 10:57:42 2024 +0200

    rebase-conflit modification README 2

diff --git a/README.md b/README.md
index 9c12f37..54aea1b 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-# modification rebase 1
\ No newline at end of file
+# modification rebase 2
```

Utilisation

Une étiquette étant une référence, vous l'utiliserez principalement comme un alias vers un commit particulier.

- git checkout <tag_name> pour vous déplacer sur le commit en question
- git branch <tag_name> pour créer une branche à partir de ce commit.

- Il est conseillé de n'utiliser les étiquettes légères que comme marques pages personnelles, et de ne pas les pousser sur un dépôt partagé.
  - Une fois que vous n'avez plus besoin de ce marque page, vous pouvez le supprimer à l'aide de git tag -d <tag_name>.
- Au contraire, les étiquettes annotées sont-elles destinées à être partagées, par le fait qu'elles contiennent des informations utiles (date, auteur, commentaire).
  - Pour lister les étiquettes de votre projet, il vous suffit d'exécuter simplement git tag sans paramètre.
- La commande git log --oneline --decorate vous permet d'afficher les étiquettes associées à un commit.
- Une étiquette n'est pas automatiquement transmise sur le dépôt distant lorsque le commit associé est poussé. Il faut le faire manuellement à l'aide de git push :
  - git push origin v1.0
  - Il y a cependant une option permettant de transmettre les étiquettes en même temps que les commits : git push --follow-tags.
  - Cette option a l'avantage de ne pousser que les étiquettes annotées qui référencent des commits existants sur le dépôt.
  - Elle peut être activée par défaut par git config push.followTags true. Une fois ceci fait, plus besoin de spécifier l'option --follow-tags.
- Les étiquettes sont automatiquement récupérées lors des git fetch

Merci d'avoir suivi ce TP !

Jérôme BOEBION
Concepteur Développeur d'Applications
Formateur
Jerome.boebion@afpa.fr