Introduction à Git

Module RCPI01

Sebastien.Kramm@univ-rouen.fr

IUT R&T Rouen

2018-2019

Objectifs

- Introduction générale à la gestion de versions

UNIVERSITÉ DE ROUEN

Objectifs

- Introduction générale à la gestion de versions
- Introduction à Git : concepts & manips de base

UNIVERSITÉ DE ROUEN

Objectifs

- Introduction générale à la gestion de versions
- Introduction à Git : concepts & manips de base
- Vous permettre son utilisation pour vos propres projets

UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)

UNIVERSITÉ DE ROUEN

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.

UNIVERSITÉ DE ROUEN

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.
- Que ne faut-il **pas** versionner ?

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.
- Que ne faut-il **pas** versionner ?
  - Tout ce qui est produit par d'autres programmes (fichier compilés, générés, etc.)

UNIVERSITÉ DE ROUEN

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.
- Que ne faut-il **pas** versionner ?
  - Tout ce qui est produit par d'autres programmes (fichier compilés, générés, etc.)
  - Tout ce qui est fichier binaires "évolutifs" (fichiers bureautiques)
    Mais on peut versionner des fichiers binaires "fixes" (images)

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.
- Que ne faut-il **pas** versionner ?
  - Tout ce qui est produit par d'autres programmes (fichier compilés, générés, etc.)
  - Tout ce qui est fichier binaires "évolutifs" (fichiers bureautiques)
    Mais on peut versionner des fichiers binaires "fixes" (images)
- Pas conçu comme un outil de sauvegarde...

UNIVERSITÉ DE ROUEN

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.
- Que ne faut-il **pas** versionner ?
  - Tout ce qui est produit par d'autres programmes (fichier compilés, générés, etc.)
  - Tout ce qui est fichier binaires "évolutifs" (fichiers bureautiques)
    Mais on peut versionner des fichiers binaires "fixes" (images)
- Pas conçu comme un outil de sauvegarde...

UNIVERSITÉ DE ROUEN

Pourquoi et quand utiliser un logiciel de gestion de version ?

- Contexte : developpement de "projet", seul ou en équipe
  - code source, $\forall$ le language
  - site web (Html, Css, Javascript, ...)
  - rédaction (manuels, rapports, articles, ...)
    ⚠ : implique l'utilisation d'un langage de balisage (LaTeX, Markdown)
- Que peut-on versionner ?
  - En principe : des fichiers "texte brut" uniquement, d'origine humaine.
  - En pratique : un peu tout, mais l'intéret est surtout sur le texte.
- Que ne faut-il **pas** versionner ?
  - Tout ce qui est produit par d'autres programmes (fichier compilés, générés, etc.)
  - Tout ce qui est fichier binaires "évolutifs" (fichiers bureautiques)
    Mais on peut versionner des fichiers binaires "fixes" (images)
- Pas conçu comme un outil de sauvegarde...
  (même si ça l'est en pratique).

UNIVERSITÉ DE ROUEN

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !
- ✔ sur : rien n'est jamais perdu

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !
- ✔ sur : rien n'est jamais perdu
- ✔ léger, rapide

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !
- ✔ sur : rien n'est jamais perdu
- ✔ léger, rapide
- ✔ distribué : chaque utilisateur à tout sous la main

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !
- ✔ sur : rien n'est jamais perdu
- ✔ léger, rapide
- ✔ distribué : chaque utilisateur à tout sous la main
- ✖ Inconvénient : prise en main complexe...
  mars 2018 : ~ 100 k questions taggées "git" sur SO
  https://stackoverflow.com/questions/tagged/git

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !
- ✔ sur : rien n'est jamais perdu
- ✔ léger, rapide
- ✔ distribué : chaque utilisateur à tout sous la main
- ✖ Inconvénient : prise en main complexe...
  mars 2018 : ~ 100 k questions taggées "git" sur SO
  https://stackoverflow.com/questions/tagged/git

Pourquoi Git ? Alternatives ?

Git est :

- ✔ Puissant !
- ✔ sur : rien n'est jamais perdu
- ✔ léger, rapide
- ✔ distribué : chaque utilisateur à tout sous la main
- ✖ Inconvénient : prise en main complexe...
  mars 2018 : ~ 100 k questions taggées "git" sur SO
  https://stackoverflow.com/questions/tagged/git

Alternatives (voir WP) :

- distribué Mercurial
- centralisé : Subversion (svn)

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)
- 2018-01 : version 2.16

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)
- 2018-01 : version 2.16
- Aujourd'hui :
  → mai 2017 : "*Windows is live on Git*"

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)
- 2018-01 : version 2.16
- Aujourd'hui :
  → mai 2017 : "*Windows is live on Git*"
  - https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)
- 2018-01 : version 2.16
- Aujourd'hui :
  → mai 2017 : "*Windows is live on Git*"
  - https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/
  - https://github.com/Microsoft

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)
- 2018-01 : version 2.16
- Aujourd'hui :
  → mai 2017 : "*Windows is live on Git*"
  - https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/
  - https://github.com/Microsoft

Historique

- Développé par Linus Torvalds pour le dev. du noyau Linux.
  A l'époque insatisfait de l'outil propriétaire qu'il utilisait, a décidé d'en écrire un conforme à ses idées.
- version 1.0 : 2005
- 2014 : version 2.0 (fin de la rétro-compatibilité avec 1.X)
- 2018-01 : version 2.16
- Aujourd'hui :
  → mai 2017 : "*Windows is live on Git*"
  - https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/
  - https://github.com/Microsoft

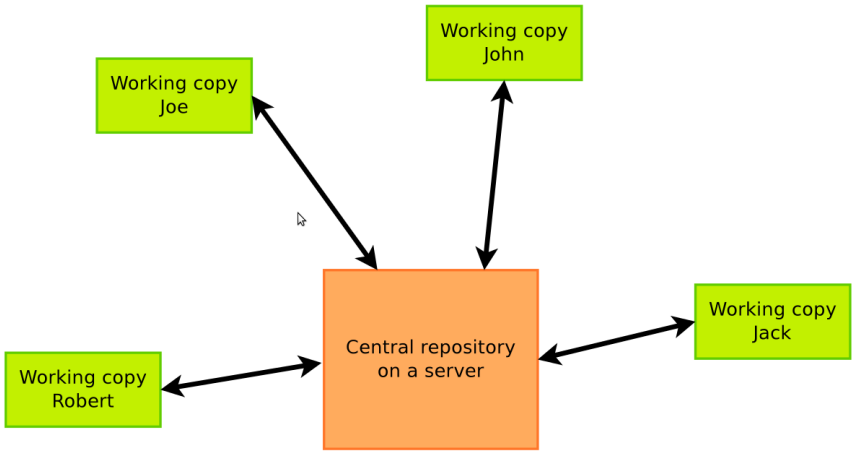
Conclusion

Git est dominant aujourd'hui, mais pas exempt de critiques

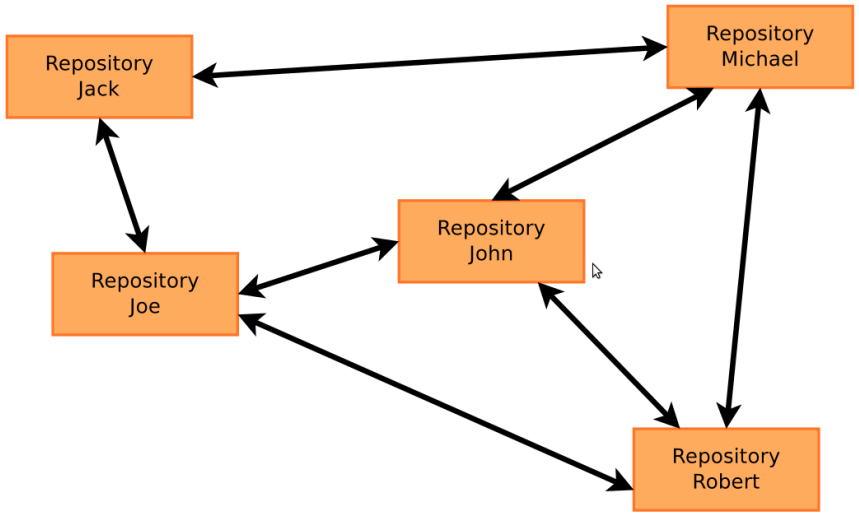
Sommaire

Modèle d'utilisation

- Centralisé :



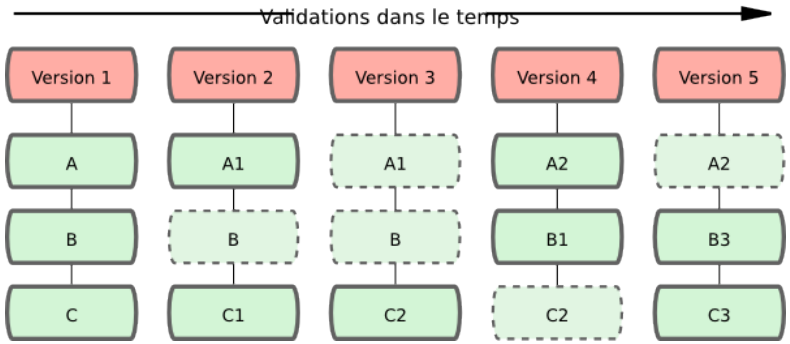
- Distribué (Git) :



- vision assez théorique : le modèle distribué est plus robuste mais a des gros inconvénients
- en pratique l'utilisation de Git se fait via un *repo* "de référence" (= centralisé) via un service type Github, ou un serveur local à l'entreprise

Introduction

- Git permet de prendre des **instantané** (*snapshot*) d'un projet, et qui sont liés entre eux : on parle de "**commit**"



- Chaque commit est identifié par un ***hash*** : clé unique.

```
commit 3b0855c8f443440038cce20c7f3a3896689ebda2
Author: skramm <sebastien.kramm@univ-rouen.fr>
Date:   Fri Mar 16 08:57:29 2018 +0100
    minor
```

- Les commits sont **locaux**.
- Lors de l'étape de synchronisation avec un autre dépot, c'est **tous** les commits qui sont transférés.

Différence entre Git et d'autres logiciels de SCM

- Git stocke des *snapshots* (totalité du fichier) à chaque commit
- D'autres SCM ne stockent que les changements entre version (*diff*)

Différence entre Git et d'autres logiciels de SCM

- Git stocke des *snapshots* (totalité du fichier) à chaque commit
- D'autres SCM ne stockent que les changements entre version (*diff*)

- Un "diff" est un morceau de texte qui montre la différence entre deux fichiers texte, ligne par ligne :
  - "-" : ligne supprimée
  - "+" : ligne ajoutée

```
index abe1cd4..04ea774 100644
--- a/liste
+++ b/liste
@@ -1,3 +1,3 @@
 fraise
-pomme
+ananas
 banane
```

Différence entre Git et d'autres logiciels de SCM

- Git stocke des *snapshots* (totalité du fichier) à chaque commit
- D'autres SCM ne stockent que les changements entre version (*diff*)

- Un "diff" est un morceau de texte qui montre la différence entre deux fichiers texte, ligne par ligne :
  - "-" : ligne supprimée
  - "+" : ligne ajoutée

```
index abe1cd4..04ea774 100644
--- a/liste
+++ b/liste
@@ -1,3 +1,3 @@
 fraise
-pomme
+ananas
 banane
```

- Un "diff" peut servir à "patcher" un logiciel.

Différence entre Git et d'autres logiciels de SCM

- Git stocke des *snapshots* (totalité du fichier) à chaque commit
- D'autres SCM ne stockent que les changements entre version (*diff*)

- Un "diff" est un morceau de texte qui montre la différence entre deux fichiers texte, ligne par ligne :
  - "-" : ligne supprimée
  - "+" : ligne ajoutée

```
index abe1cd4..04ea774 100644
--- a/liste
+++ b/liste
@@ -1,3 +1,3 @@
 fraise
-pomme
+ananas
 banane
```

- Un "diff" peut servir à "patcher" un logiciel.

Conséquences

- Un "repo" Git peut vite occuper beaucoup de place
- Les commits de Git peuvent être réorganisés à volonté

Git, c'est quoi

- Un programme unique ("git") offrant de nombreuses commandes

```
git COMMANDE (options) (arguments)
```

- La liste des (principales) commandes :

```
git
```

- Pour avoir l'aide en ligne :

```
git help COMMANDE
```

- GUI ? Oui...

Git, c'est quoi

- Un programme unique ("git") offrant de nombreuses commandes

```
git COMMANDE (options) (arguments)
```

- La liste des (principales) commandes :

```
git
```

- Pour avoir l'aide en ligne :

```
git help COMMANDE
```

- GUI ? Oui... Mais l'utilisation en ligne de commande permet de mieux comprendre ce qu'on fait.

Vocabulaire

repository	tout l'historique du projet contenu dans le répertoire .git
diff ou patch	différences entre deux versions d'un fichier
"committer"	action d'enregistrer la version actuelle d'un ensemble de fichiers dans le repository **local**
commit	résultat d'une action de commit, représenté par un hash SHA-1
branche	une "ligne" de développement
tag	un identificateur symbolique pour un commit ou une branche
Working copy	L'arborescence de fichiers en cours d'édition
Index	un objet traquant les fichiers modifiés
Blob	données binaires correspondant aux données utilisateur

Sommaire

UNIVERSITÉ DE ROUEN

Création d'un dépot local

- Création (les 2 commandes sont équivalentes)

```
mkdir monProjet
cd monProjet
git init
```

```
git init monProjet
```

⇒ Crée un dossier dans le dossier courant un dossier `monProjet` avec dedans un dossier `.git` contenant les données internes de Git.

Création d'un dépot local

- Création (les 2 commandes sont équivalentes)

```
mkdir monProjet
cd monProjet
git init
```

```
git init monProjet
```

⇒ Crée un dossier dans le dossier courant un dossier `monProjet` avec dedans un dossier `.git` contenant les données internes de Git.

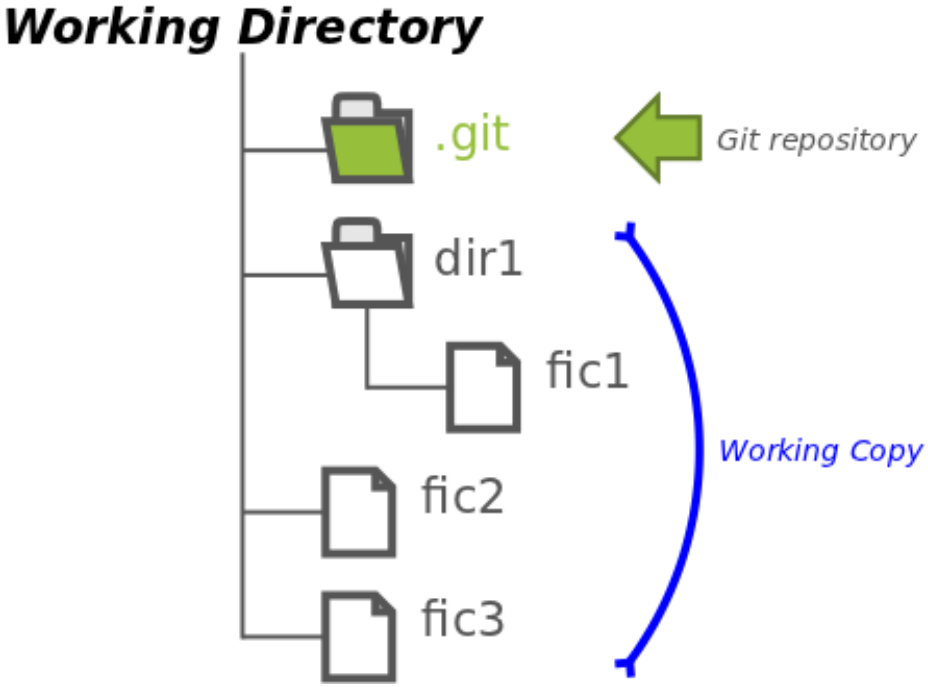
- Clonage d'un dépot distant (par ex : Github)

```
$ git clone https://github.com/USER/PROJET.git
```

  - ⚠ : va récuperer aussi tout l'historique du projet (pas forcément nécessaire)
  - Pour avoir juste la version actuelle, ajouter l'option `--depth 1`

Contenu d'un dossier versionné

- Un "dossier de travail" versionné contient deux choses :
  - Une "Working copy" des fichiers
  - Le "repository" : dossier (caché) `.git` qui contient les données internes (blobs binaires)



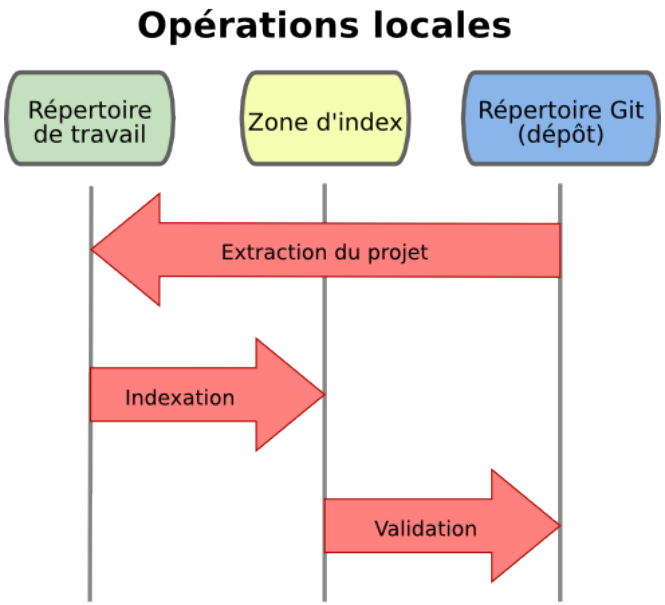
Important : les fichiers versionnés du dossier de travail sont **contrôlés** par Git.

Sommaire

Sommaire

UNIVERSITÉ DE ROUEN

Enregistrement de ses modifications

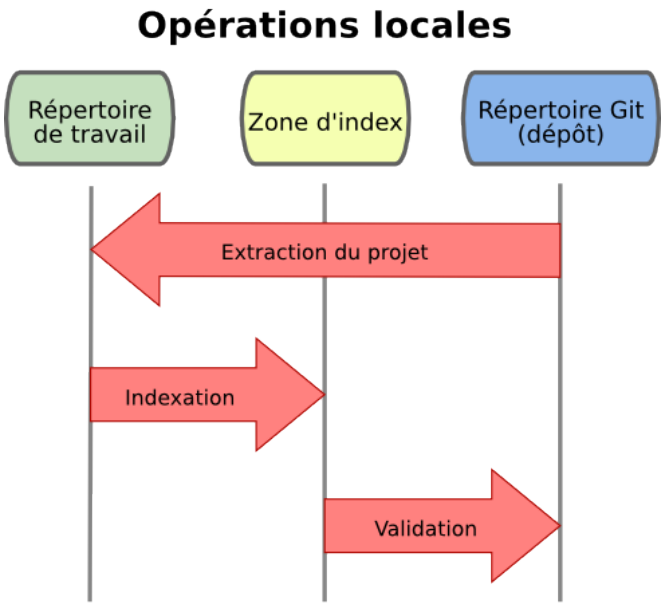


- Etat actuel de mon dossier :
  `git status`
  → liste les fichiers :
  - indexés
  - non indexés mais modifiés
  - non suivis (*untracked*)

Après une modification de fichiers dans le répertoire de travail :

1. indexation des fichiers modifiés (dans la "***staging area***")

Enregistrement de ses modifications

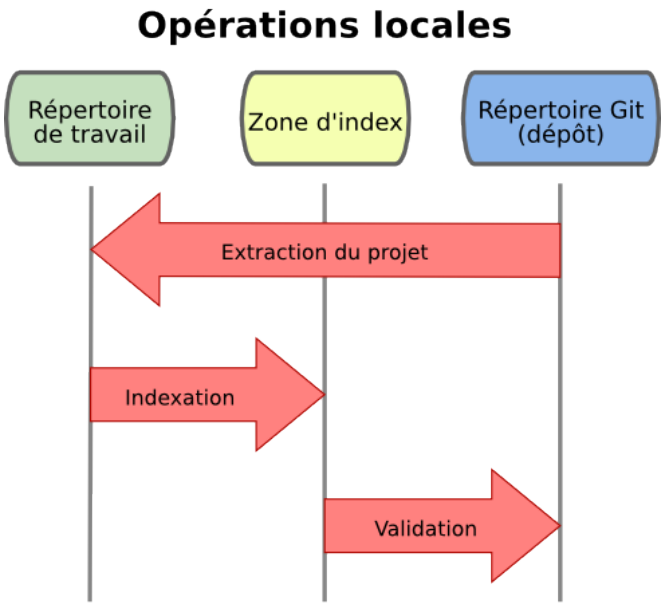


- Etat actuel de mon dossier :
  `git status`
  → liste les fichiers :
  - indexés
  - non indexés mais modifiés
  - non suivis (*untracked*)

Après une modification de fichiers dans le répertoire de travail :

1. indexation des fichiers modifiés (dans la "***staging area***")
2. basculement des fichiers indexés dans le "repo" ("commit")

Enregistrement de ses modifications

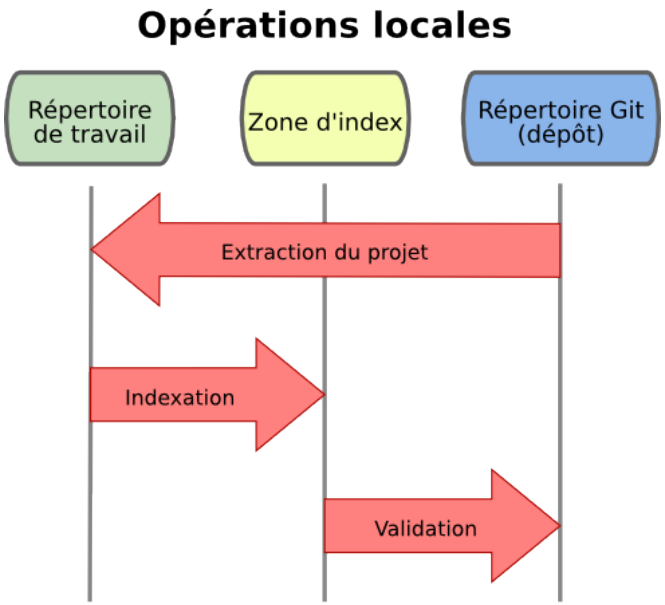


- Etat actuel de mon dossier :
  `git status`
  → liste les fichiers :
  - indexés
  - non indexés mais modifiés
  - non suivis (*untracked*)

Après une modification de fichiers dans le répertoire de travail :

1. indexation des fichiers modifiés (dans la "***staging area***")
2. basculement des fichiers indexés dans le "repo" ("commit")

Enregistrement de ses modifications



Après une modification de fichiers dans le répertoire de travail :

1. indexation des fichiers modifiés (dans la "***staging area***")
2. basculement des fichiers indexés dans le "repo" ("commit")

- Etat actuel de mon dossier :
  `git status`
  → liste les fichiers :
  - indexés
  - non indexés mais modifiés
  - non suivis (*untracked*)
- Visualiser les modifications :
  `git diff`
  → Montre les différences entre
  - version du "working copy"
  - version du dépot local

Utilisation

1. Indexer des fichiers et/ou des dossiers :

```
git add fichier1 fichier2 dossier1
```

- Attention :

- Peut être fait en plusieurs fois.
- `git status` pour voir qu'est ce qui va être enregistré.

Utilisation

1. Indexer des fichiers et/ou des dossiers :

```
git add fichier1 fichier2 dossier1
```

- Attention :
  - ajouter un dossier non vide **va ajouter tous les fichiers contenus !**

- Peut être fait en plusieurs fois.
- `git status` pour voir qu'est ce qui va être enregistré.

Utilisation

1. Indexer des fichiers et/ou des dossiers :

```
git add fichier1 fichier2 dossier1
```

- Attention :
  - ajouter un dossier non vide **va ajouter tous les fichiers contenus !**
  - ajouter un dossier vide sera ignoré.
- Peut être fait en plusieurs fois.
- `git status` pour voir qu'est ce qui va être enregistré.

Utilisation

1. Indexer des fichiers et/ou des dossiers :

```
git add fichier1 fichier2 dossier1
```

- Attention :
  - ajouter un dossier non vide **va ajouter tous les fichiers contenus !**
  - ajouter un dossier vide sera ignoré.
- Peut être fait en plusieurs fois.
- `git status` pour voir qu'est ce qui va être enregistré.

Utilisation

1. Indexer des fichiers et/ou des dossiers :

   ```
   git add fichier1 fichier2 dossier1
   ```

   - Attention :
     - ajouter un dossier non vide **va ajouter tous les fichiers contenus !**
     - ajouter un dossier vide sera ignoré.
   - Peut être fait en plusieurs fois.
   - `git status` pour voir qu'est ce qui va être enregistré.

2. Enregistrer les modifications dans le depôt local :

   ```
   git commit -m "message de commit"
   ```

Utilisation

1. Indexer des fichiers et/ou des dossiers :

   ```
   git add fichier1 fichier2 dossier1
   ```

   - Attention :
     - ajouter un dossier non vide **va ajouter tous les fichiers contenus !**
     - ajouter un dossier vide sera ignoré.
   - Peut être fait en plusieurs fois.
   - `git status` pour voir qu'est ce qui va être enregistré.

2. Enregistrer les modifications dans le depôt local :

   ```
   git commit -m "message de commit"
   ```

Utilisation

1. Indexer des fichiers et/ou des dossiers :

   ```
   git add fichier1 fichier2 dossier1
   ```

   - Attention :
     - ajouter un dossier non vide **va ajouter tous les fichiers contenus !**
     - ajouter un dossier vide sera ignoré.
   - Peut être fait en plusieurs fois.
   - `git status` pour voir qu'est ce qui va être enregistré.

2. Enregistrer les modifications dans le depôt local :

   ```
   git commit -m "message de commit"
   ```

Raccourci : pour indexer et commiter **tous** les fichiers **déjà suivis** :

```
git commit -a -m "message de commit"
```

Sommaire

Pourquoi ?

Il arrive frequemment qu'on doive :

- Corriger un bug
- Implémenter une nouvelle fonctionnalité demandée, dont on ne sait pas
  - si c'est faisable,

Pourquoi ?

Il arrive frequemment qu'on doive :

- Corriger un bug
- Implémenter une nouvelle fonctionnalité demandée, dont on ne sait pas
  - si c'est faisable,
  - si on va réussir,

Pourquoi ?

Il arrive frequemment qu'on doive :

- Corriger un bug
- Implémenter une nouvelle fonctionnalité demandée, dont on ne sait pas
  - si c'est faisable,
  - si on va réussir,
  - si elle a vraiment un intéret.

Pourquoi ?

Il arrive frequemment qu'on doive :

- Corriger un bug
- Implémenter une nouvelle fonctionnalité demandée, dont on ne sait pas
  - si c'est faisable,
  - si on va réussir,
  - si elle a vraiment un intéret.

Pourquoi ?

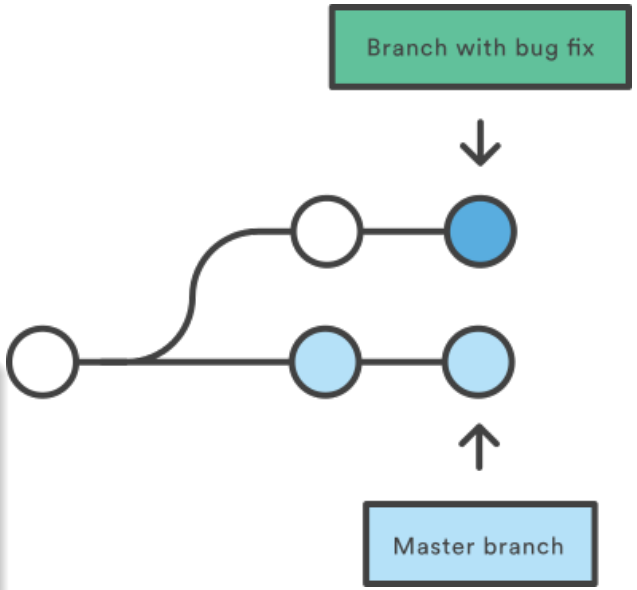
Il arrive frequemment qu'on doive :

- Corriger un bug
- Implémenter une nouvelle fonctionnalité demandée, dont on ne sait pas
  - si c'est faisable,
  - si on va réussir,
  - si elle a vraiment un intéret.

Il faut pouvoir

- modifier le code,
- mais aussi garder la version actuelle en l'état (et aussi pouvoir y opérer des corrections mineures).

⇒ Solution : créer des **branches**



En pratique

Définition

Une branche est un "séquence de commits" nommée.

- La branche par défaut s'appelle `master`

En pratique

Définition

Une branche est un "séquence de commits" nommée.

- La branche par défaut s'appelle `master`
- Créer une branche :
  `git branch mabranche`
  → Créer un pointeur `mabranche` qui pointe sur `master`

En pratique

Définition

Une branche est un "séquence de commits" nommée.

- La branche par défaut s'appelle `master`
- Créer une branche :
  `git branch mabranche`
  → Créer un pointeur `mabranche` qui pointe sur `master`
- Connaitre la branche en cours : `git status`

En pratique

Définition

Une branche est un "séquence de commits" nommée.

- La branche par défaut s'appelle `master`
- Créer une branche :
  `git branch mabranche`
  → Créer un pointeur `mabranche` qui pointe sur `master`
- Connaitre la branche en cours : `git status`
- Visualisez les branches existantes : `git branch`

En pratique

Définition

Une branche est un "séquence de commits" nommée.

- La branche par défaut s'appelle `master`
- Créer une branche :
  `git branch mabranche`
  → Créer un pointeur `mabranche` qui pointe sur `master`
- Connaitre la branche en cours : `git status`
- Visualisez les branches existantes : `git branch`
- Basculer son dossier de travail sur une branche :
  `git checkout mabranche`
  ⚠ : Il faut que les modifications actuelles aient été "committées" avant.

En pratique

Définition

Une branche est un "séquence de commits" nommée.

- La branche par défaut s'appelle `master`
- Créer une branche :
  `git branch mabranche`
  → Créer un pointeur `mabranche` qui pointe sur `master`
- Connaitre la branche en cours : `git status`
- Visualisez les branches existantes : `git branch`
- Basculer son dossier de travail sur une branche :
  `git checkout mabranche`
  ⚠ : Il faut que les modifications actuelles aient été "committées" avant.
- Merger dans la branche en cours le dernier commit de la branche XXX :
  `git merge XXX`

Commit HEAD et état courant du *working copy*

Dans tout dépot :

- Il n'y a qu'une seule branche active
- Il y a un (et un seul) commit nommé `HEAD`, qui pointe sur le dernier commit de la branche active
  ⇒ correspond à ce qui est dans le "working copy" (dernier *checkout*)

Commit HEAD et état courant du *working copy*

Dans tout dépot :

- Il n'y a qu'une seule branche active
- Il y a un (et un seul) commit nommé `HEAD`, qui pointe sur le dernier commit de la branche active
  ⇒ correspond à ce qui est dans le "working copy" (dernier *checkout*)

Rappel d'une autre version

La commande `checkout` remet le "*working directory*" dans l'état associé avec un commit

- `git checkout mabranche` : dernier commit de la branche

Commit HEAD et état courant du *working copy*

Dans tout dépot :

- Il n'y a qu'une seule branche active
- Il y a un (et un seul) commit nommé `HEAD`, qui pointe sur le dernier commit de la branche active
  ⇒ correspond à ce qui est dans le "working copy" (dernier *checkout*)

Rappel d'une autre version

La commande `checkout` remet le "*working directory*" dans l'état associé avec un commit

- `git checkout mabranche` : dernier commit de la branche
- `git checkout 7c45f65b` : rappel d'un commit particulier

(Attention, on passe alors en mode "*detached HEAD*", voir ici)

Commit HEAD et état courant du *working copy*

Dans tout dépot :

- Il n'y a qu'une seule branche active
- Il y a un (et un seul) commit nommé `HEAD`, qui pointe sur le dernier commit de la branche active
  ⇒ correspond à ce qui est dans le "working copy" (dernier *checkout*)

Rappel d'une autre version

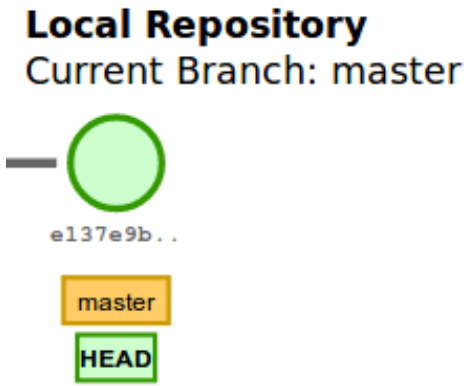
La commande `checkout` remet le "*working directory*" dans l'état associé avec un commit

- `git checkout mabranche` : dernier commit de la branche
- `git checkout 7c45f65b` : rappel d'un commit particulier

  (Attention, on passe alors en mode "*detached HEAD*", voir ici
- Attention : il faut que le WC soit "propre" : rien de modifié

Illustration : cas simple de branche

1. repo de départ

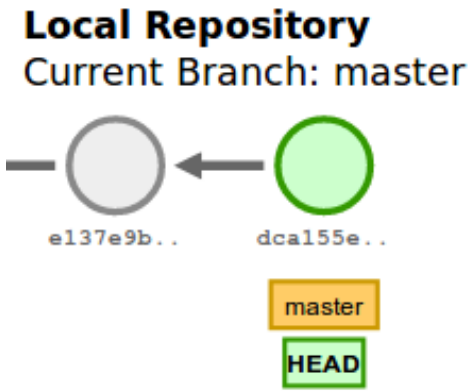


→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : cas simple de branche

1. repo de départ
2. 1 commit

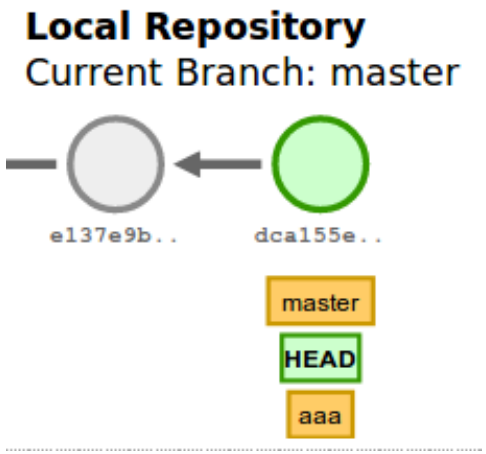


→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : cas simple de branche

1. repo de départ
2. 1 commit
3. création d'une branche aaa

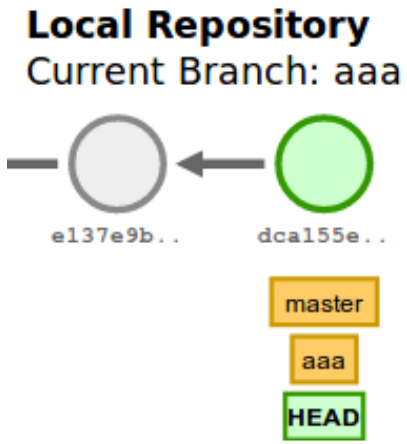


→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : cas simple de branche

1. repo de départ
2. 1 commit
3. création d'une branche aaa
4. checkout sur aaa

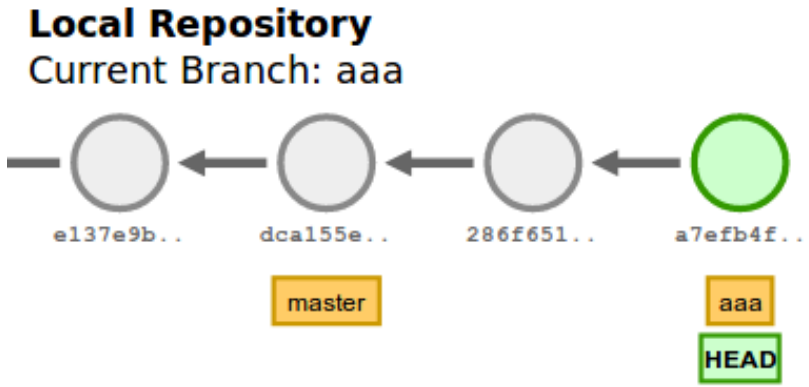


→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : cas simple de branche

1. repo de départ
2. 1 commit
3. création d'une branche aaa
4. checkout sur aaa
5. 2 commit sur aaa

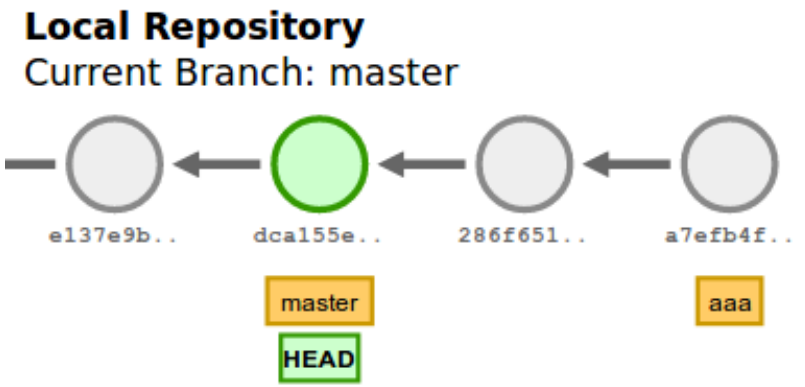


→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : cas simple de branche

1. repo de départ
2. 1 commit
3. création d'une branche aaa
4. checkout sur aaa
5. 2 commit sur aaa
6. checkout sur `master`

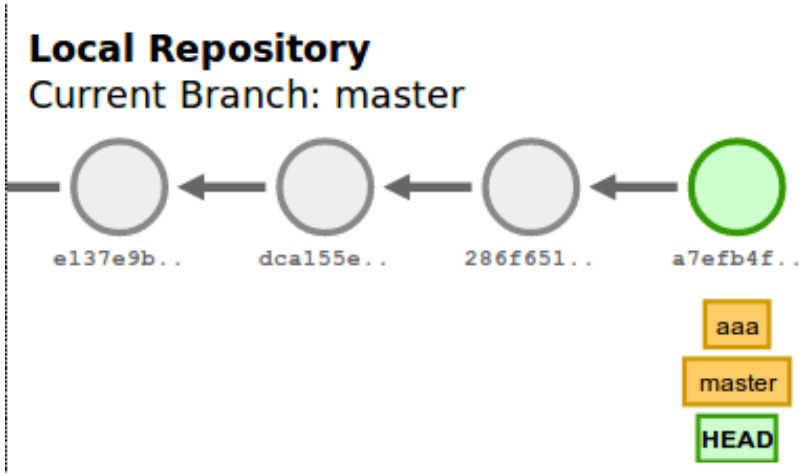


→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : cas simple de branche

1. repo de départ
2. 1 commit
3. création d'une branche aaa
4. checkout sur aaa
5. 2 commit sur aaa
6. checkout sur `master`
7. merge : fusion des modifs faites sur la branche *aaa* dans *master*



→ Pas de problème de fusion.

(Illustrations : http://onlywei.github.io/explain-git-with-d3/)

Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*

Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`

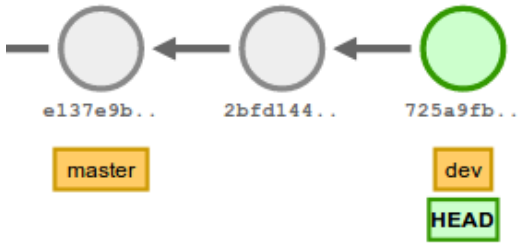


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`

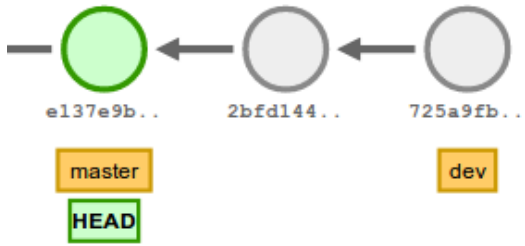


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`

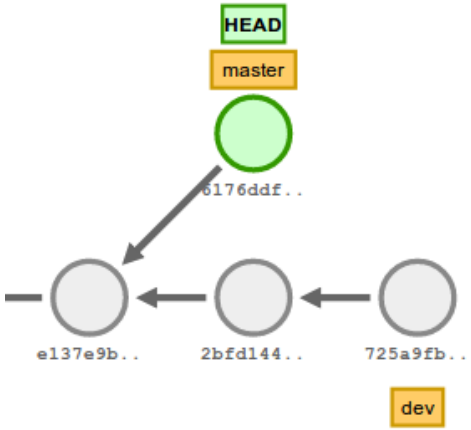


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...

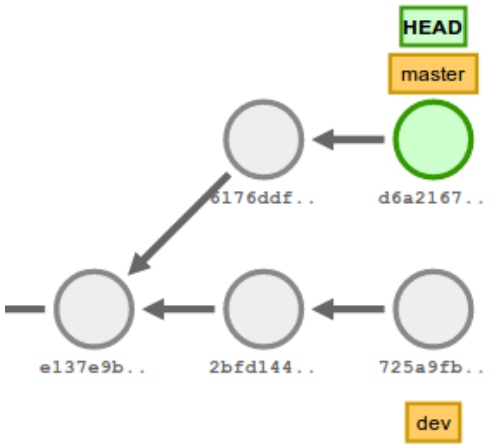


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...
6. on retourne sur `dev` continuer la feature

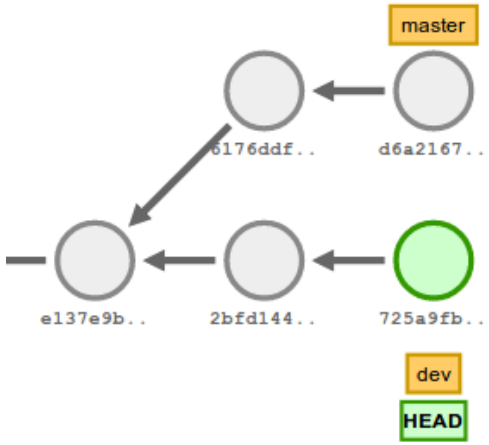


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...
6. on retourne sur `dev` continuer la feature
7. Mais on veut importer la correction de bug : merge de `master`

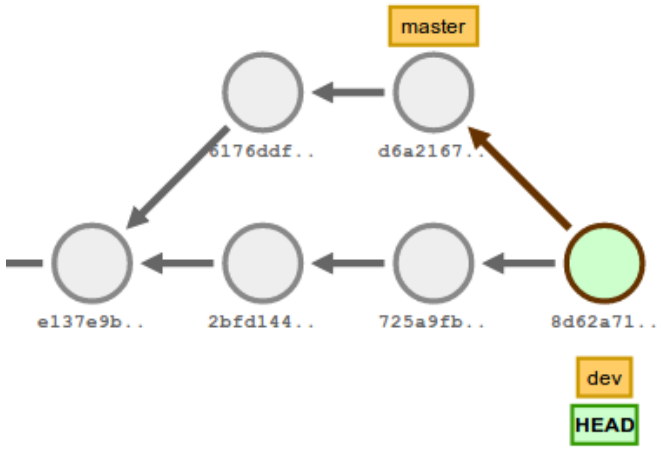


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...
6. on retourne sur `dev` continuer la feature
7. Mais on veut importer la correction de bug : merge de `master`
8. on continue le travail sur dev

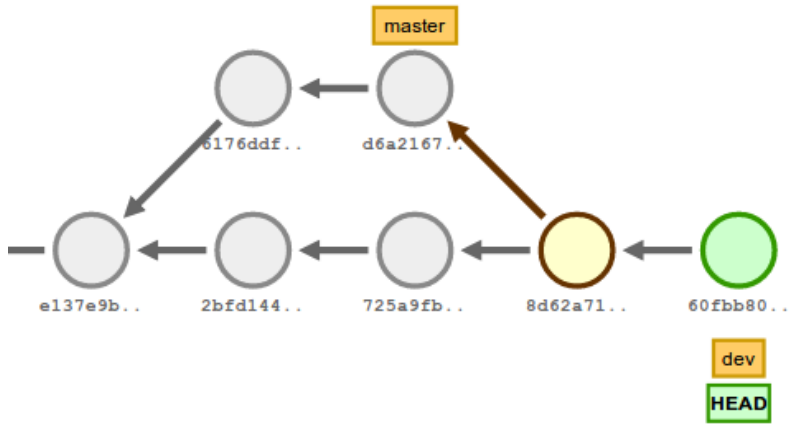


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...
6. on retourne sur `dev` continuer la feature
7. Mais on veut importer la correction de bug : merge de `master`
8. on continue le travail sur dev
9. C'est bon, fini : on repasse sur `master`

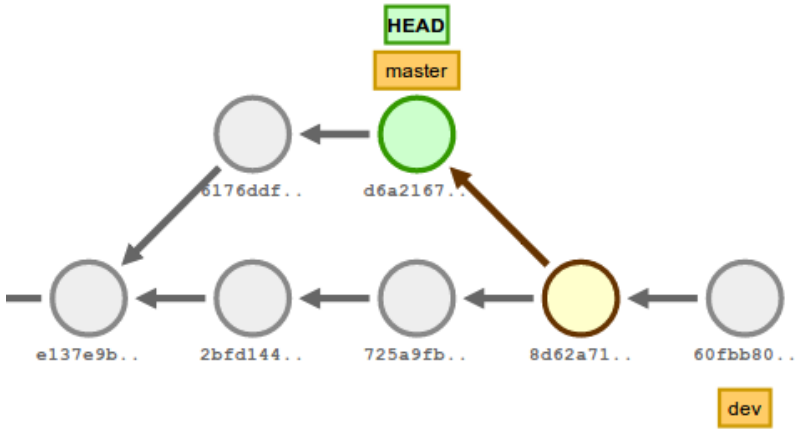


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...
6. on retourne sur `dev` continuer la feature
7. Mais on veut importer la correction de bug : merge de `master`
8. on continue le travail sur dev
9. C'est bon, fini : on repasse sur `master`
10. une dernière petite modif...

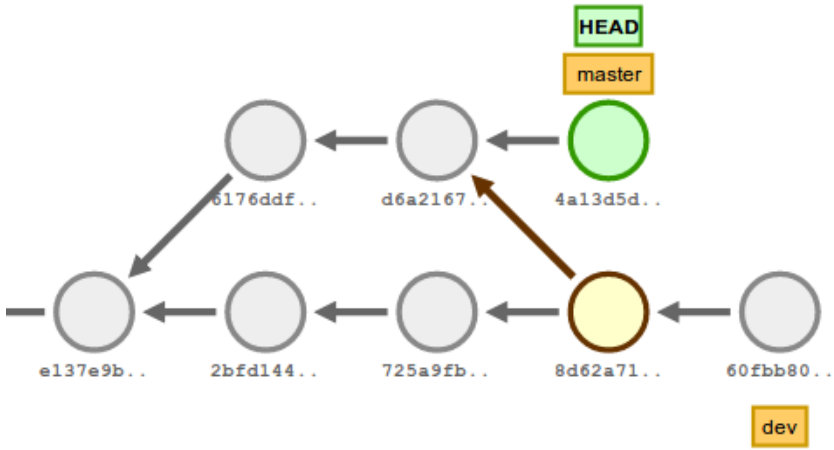
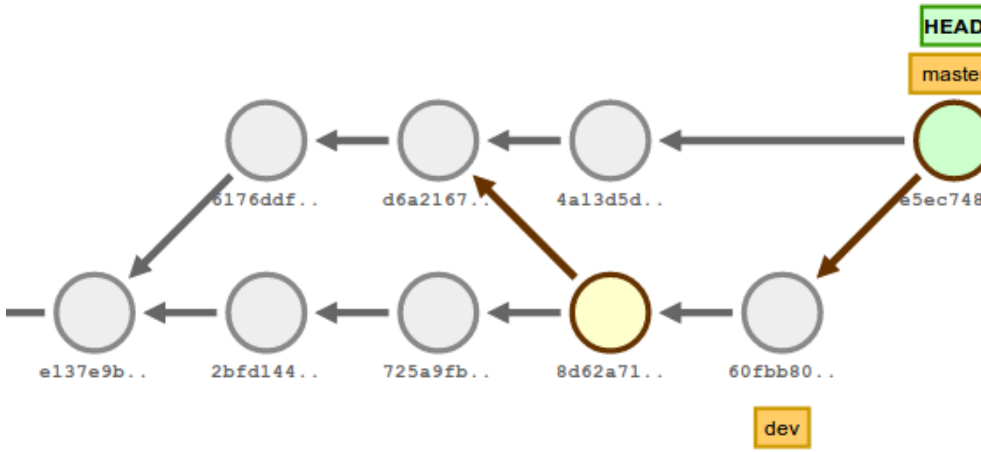


Illustration : branches divergentes

1. repo de départ, création d'une branche `dev`, et *checkout*
2. 2 commit sur `dev`
3. bug à fixer ! : retour sur `master`
4. 1 commit de correction sur `master`
5. et encore 1...
6. on retourne sur `dev` continuer la feature
7. Mais on veut importer la correction de bug : merge de `master`
8. on continue le travail sur dev
9. C'est bon, fini : on repasse sur `master`
10. une dernière petite modif...
11. ... et on merge la modif de `dev` dans `master` !



Gestion des conflits

- Il arrive que Git ne puisse pas effectuer le "merge" (par exemple, si les mêmes lignes dans les deux fichiers sont différentes).
- Git ajoute dans les fichiers concernés des **marqueurs de conflit**.
  Il faut alors :
  1. Résoudre le conflict en éditant le fichier ;
  2. Ajouter les fichiers à l'index ;
  3. Committer le resultat.
- Il existe des outils graphiques facilitant le merge (exemple : meld)

Exemple de conflit

1. Je committe dans le "master" ce fichier :

```
fraise
poire
banane
```

Exemple de conflit

1. Je committe dans le "master" ce fichier :
2. Je crée une branche "ajouts", je bascule dessus, j'édite le fichier et j'ajoute une ligne :

```
fraise
abricot
poire
banane
```

Exemple de conflit

1. Je committe dans le "master" ce fichier :
2. Je crée une branche "ajouts", je bascule dessus, j'édite le fichier et j'ajoute une ligne :
3. Je committe ce changement

```
fraise
abricot
poire
banane
```

Exemple de conflit

1. Je committe dans le "master" ce fichier :
2. Je crée une branche "ajouts", je bascule dessus, j'édite le fichier et j'ajoute une ligne :
3. Je committe ce changement
4. Je rebascule sur "master", ouvre le fichier, et retrouve :

```
fraise
poire
banane
```

Exemple de conflit

1. Je committe dans le "master" ce fichier :
2. Je crée une branche "ajouts", je bascule dessus, j'édite le fichier et j'ajoute une ligne :
3. Je committe ce changement
4. Je rebascule sur "master", ouvre le fichier, et retrouve :
5. J'ajoute une ligne et je committe ce changement :

```
fraise
ananas
poire
banane
```

Exemple de conflit

1. Je committe dans le "master" ce fichier :
2. Je crée une branche "ajouts", je bascule dessus, j'édite le fichier et j'ajoute une ligne :
3. Je committe ce changement
4. Je rebascule sur "master", ouvre le fichier, et retrouve :
5. J'ajoute une ligne et je committe ce changement :
6. Je bascule sur "ajouts", et souhaite "merger" les changements que j'ai mis dans master : `git merge master`

```
Auto-merging liste
CONFLICT (content): Merg
Automatic merge failed;
```

Exemple de conflit

1. Je committe dans le "master" ce fichier :
2. Je crée une branche "ajouts", je bascule dessus, j'édite le fichier et j'ajoute une ligne :
3. Je committe ce changement
4. Je rebascule sur "master", ouvre le fichier, et retrouve :
5. J'ajoute une ligne et je committe ce changement :
6. Je bascule sur "ajouts", et souhaite "merger" les changements que j'ai mis dans master : `git merge master`
7. En ouvrant le fichier, on voit ceci :
   Il faut éditer le fichier à la main, puis commiter

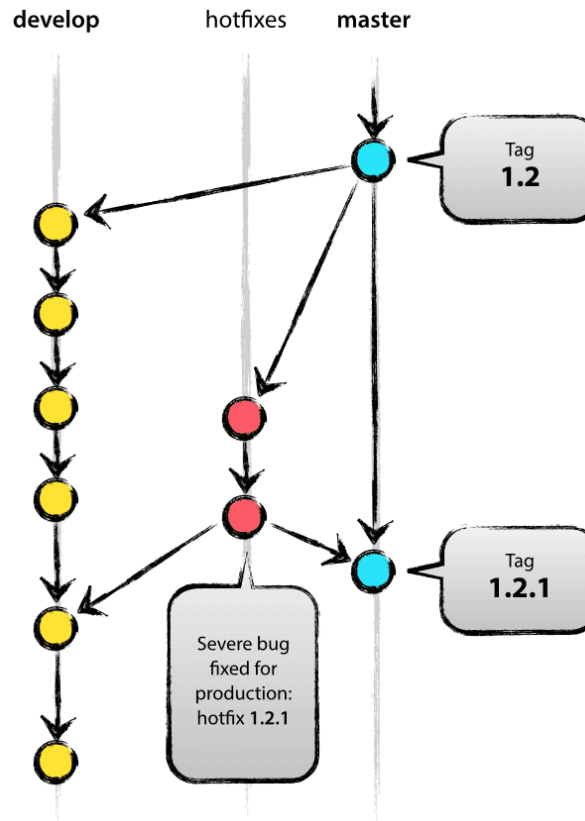
```
fraise
<<<<<<< HEAD
ananas
=======
abricot
>>>>>>> master
poire
banane
```

Sommaire

UNIVERSITÉ DE ROUEN

Tag

- A un moment donné, il peut être utile de "marquer" un des commit comme étant une étape majeure.
  Typiquement, une "release" :
  `git tag v1.0`
- Pour avoir la liste des versions taggées :
  `git tag -l` ou `git tag --list`
- Attention, comme un commit, ca reste local !
  Pour le propager, il faut le specifier lors d'un "push" :
  `git push --tags`



Sommaire

Dépots distants

- Un dépot peut en référencer d'autres
- En général, un seul dépot de référence (cas le plus courant)

Dépots distants

- Un dépot peut en référencer d'autres
- En général, un seul dépot de référence (cas le plus courant)
- Visualiser les dépôts liés : `git remote -v`

Dépots distants

- Un dépot peut en référencer d'autres
- En général, un seul dépot de référence (cas le plus courant)
- Visualiser les dépôts liés : `git remote -v`
- Récupérer les modifications faites par d'autres et les "merger" : `git pull`
  - implique la connectivité
  - ⚠ : le "merge" peut réussir ou échouer. Il faut alors éditer les conflits à la main.

Dépots distants

- Un dépot peut en référencer d'autres
- En général, un seul dépot de référence (cas le plus courant)
- Visualiser les dépôts liés : `git remote -v`
- Récupérer les modifications faites par d'autres et les "merger" : `git pull`
  - implique la connectivité
  - ⚠ : le "merge" peut réussir ou échouer. Il faut alors éditer les conflits à la main.
- Transferer les modifications (de la branche courante) aux tiers : `git push`
  - opération réseau (en général) : implique son accessibilité
  - implique l'autorisation sur le dépot distant
  - possible **uniquement** si le depot local est "synchro" avec le dépot distant (que personne d'autre n'aie poussé ses modifs)
    A défaut, il faudra d'abord faire un "pull" et "merger" avec résolution de conflits éventuels.

Branches et dépots distants

- Les commandes push et pull ne propagent / récupèrent que la branche courante.
- Pour propager une branche sur le dépot distant, il faut qu'elle existe sur le dépôt distant.
  Il faut donc la créer en le spécifiant avec une option de la commande "push" (voir `git help push`).
- Justification : une branche peut correspondre à un travail purement local, et n'a pas toujours vocation à être disponible/utilisable par les autres devs.

Sommaire

Configuration de Git

- se fait sur deux niveaux :
  - Configuration du projet : fichier `.git/config`
  - Configuration de utilisateur : fichier `~/.gitconfig`
- Commandes :
  `git config XXXXX` (config projet)
  `git config --global XXXXX` (config utilisateur)
- Pour lister tout ce qui est enregistré :
  `git config --list`

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`
2. Je veux annuler **toutes** les modifications faites au fichiers suivis depuis le dernier commit.
   ⇒ `git reset --hard`

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`
2. Je veux annuler **toutes** les modifications faites au fichiers suivis depuis le dernier commit.
   ⇒ `git reset --hard`
3. J'ai déjà "committé" le fichier FICH mais je veux le retirer du dépôt :
   ⇒ `git rm FICH` (1)
   ⚠Implique que le fichier n'a pas été encore modifié dans la *working copy* !

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`
2. Je veux annuler **toutes** les modifications faites au fichiers suivis depuis le dernier commit.
   ⇒ `git reset --hard`
3. J'ai déjà "committé" le fichier FICH mais je veux le retirer du dépôt :
   ⇒ `git rm FICH` (1)
   ⚠Implique que le fichier n'a pas été encore modifié dans la *working copy* !
4. Je veux renommer le fichier FICH en TRUC
   ⇒ `git mv FICH TRUC`
   De façon similaire, on va déplacer un fichier de la même façon : pour déplacer le fichier FICH depuis le dossier D1 vers le dossier D2 :
   ⇒ `git mv D1/FICH D2`

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`
2. Je veux annuler **toutes** les modifications faites au fichiers suivis depuis le dernier commit.
   ⇒ `git reset --hard`
3. J'ai déjà "committé" le fichier FICH mais je veux le retirer du dépôt :
   ⇒ `git rm FICH` (1)
   ⚠Implique que le fichier n'a pas été encore modifié dans la *working copy* !
4. Je veux renommer le fichier FICH en TRUC
   ⇒ `git mv FICH TRUC`
   De façon similaire, on va déplacer un fichier de la même façon : pour déplacer le fichier FICH depuis le dossier D1 vers le dossier D2 :
   ⇒ `git mv D1/FICH D2`

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`
2. Je veux annuler **toutes** les modifications faites au fichiers suivis depuis le dernier commit.
   ⇒ `git reset --hard`
3. J'ai déjà "committé" le fichier FICH mais je veux le retirer du dépôt :
   ⇒ `git rm FICH` (1)
   ⚠Implique que le fichier n'a pas été encore modifié dans la *working copy* !
4. Je veux renommer le fichier FICH en TRUC
   ⇒ `git mv FICH TRUC`
   De façon similaire, on va déplacer un fichier de la même façon : pour déplacer le fichier FICH depuis le dossier D1 vers le dossier D2 :
   ⇒ `git mv D1/FICH D2`

Attention

Les action 3 & 4 ne seront effectives que lors du prochain commit !

Oups !

1. J'ai mis le fichier FICH dans l'index (via un "git add"), mais en fait je ne veux pas le "committer" :
   ⇒ `git reset FICH`
2. Je veux annuler **toutes** les modifications faites au fichiers suivis depuis le dernier commit.
   ⇒ `git reset --hard`
3. J'ai déjà "committé" le fichier FICH mais je veux le retirer du dépôt :
   ⇒ `git rm FICH` (1)
   ⚠Implique que le fichier n'a pas été encore modifié dans la *working copy* !
4. Je veux renommer le fichier FICH en TRUC
   ⇒ `git mv FICH TRUC`
   De façon similaire, on va déplacer un fichier de la même façon : pour déplacer le fichier FICH depuis le dossier D1 vers le dossier D2 :
   ⇒ `git mv D1/FICH D2`

Attention

Les action 3 & 4 ne seront effectives que lors du prochain commit !

Quelques détails

- A chaque exécution, mon code génère des fichiers de type `.abc` dans le dossier courant. Je ne veux pas qu'ils m'encombrent, je ne veux pas les committer !

Quelques détails

- A chaque exécution, mon code génère des fichiers de type `.abc` dans le dossier courant. Je ne veux pas qu'ils m'encombrent, je ne veux pas les committer !
  Solution :
  1. ajouter dans le dossier un fichier `.gitignore`
  2. l'éditer, et y mettre la ligne `*.abc`
  3. indexer et committer ce fichier

Quelques détails

- A chaque exécution, mon code génère des fichiers de type `.abc` dans le dossier courant. Je ne veux pas qu'ils m'encombrent, je ne veux pas les committer !
  Solution :
  1. ajouter dans le dossier un fichier `.gitignore`
  2. l'éditer, et y mettre la ligne `*.abc`
  3. indexer et committer ce fichier

  ⇒ ils n'apparaitront jamais dans le `git status`

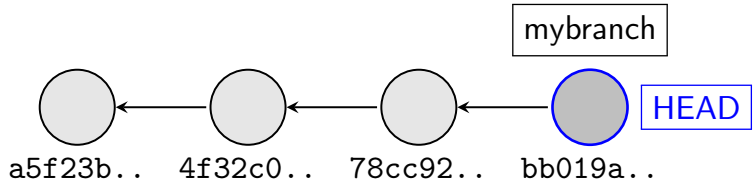
Quelques détails

- A chaque exécution, mon code génère des fichiers de type `.abc` dans le dossier courant. Je ne veux pas qu'ils m'encombrent, je ne veux pas les committer !
  Solution :
  1. ajouter dans le dossier un fichier `.gitignore`
  2. l'éditer, et y mettre la ligne `*.abc`
  3. indexer et committer ce fichier

  ⇒ ils n'apparaitront jamais dans le `git status`
- Je veux enregistrer dans le repo un dossier vide :
  - Rep. 1 : c'est impossible : Git stocke les données contenues, pas les fichiers.
    → pas de contenu=pas de stockage !
  - Rep. 2 : mettre dedans un fichier README contenant par exemple "ce dossier sert à cecicela" et le "committer".

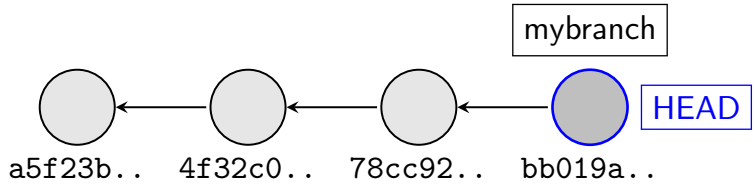
Detached HEAD

- Normalement, le WC correspond au dernier commit d'une branche (HEAD)

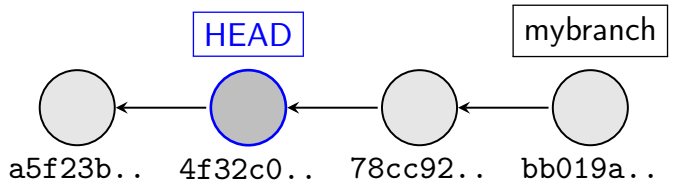


Detached HEAD

- Normalement, le WC correspond au dernier commit d'une branche (HEAD)

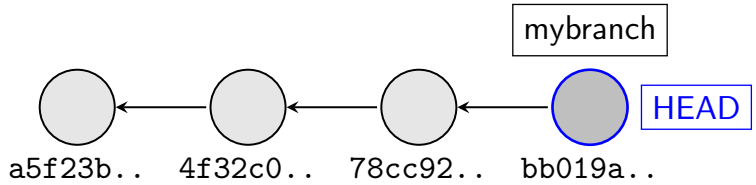


- Mais il peut arriver qu'on souhaite regarder un commit particulier du dépôt. On se retrouve dans l'état "*Detached HEAD*" :

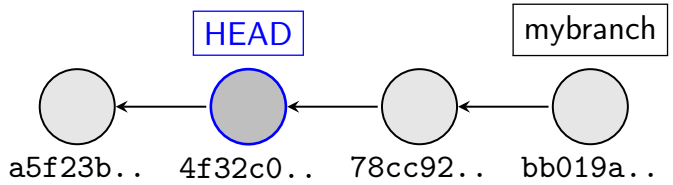


Detached HEAD

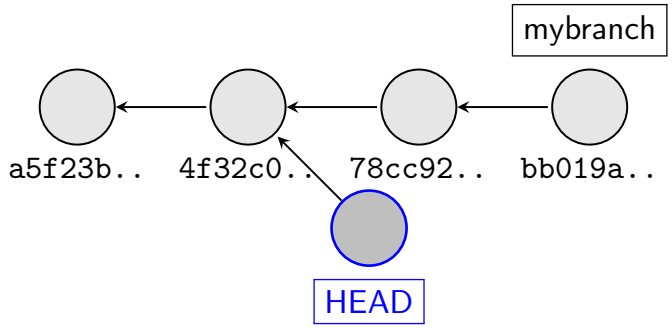
- Normalement, le WC correspond au dernier commit d'une branche (HEAD)



- Mais il peut arriver qu'on souhaite regarder un commit particulier du dépôt. On se retrouve dans l'état "*Detached HEAD*" :

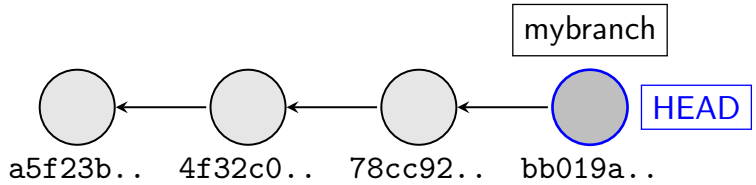


- En cas de commit, on se retrouve avec une branche sans nom :

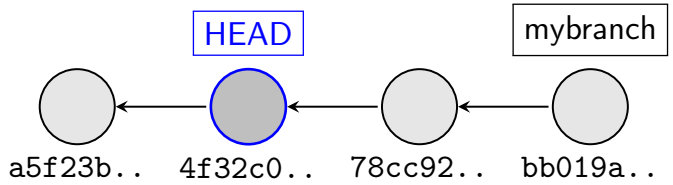


Detached HEAD

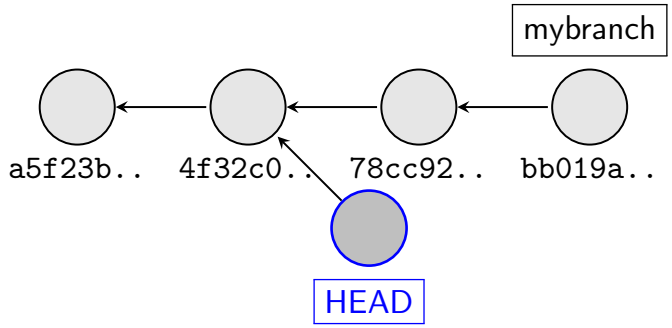
- Normalement, le WC correspond au dernier commit d'une branche (HEAD)



- Mais il peut arriver qu'on souhaite regarder un commit particulier du dépôt. On se retrouve dans l'état "*Detached HEAD*" :

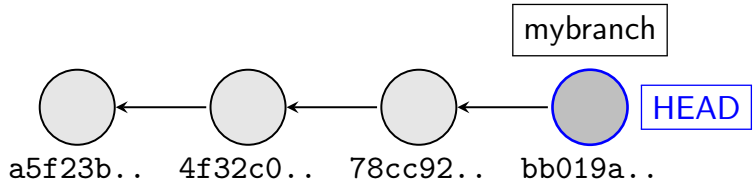


- En cas de commit, on se retrouve avec une branche sans nom :

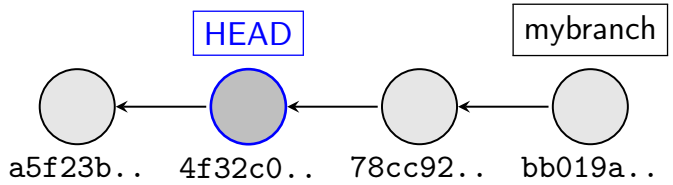


Detached HEAD

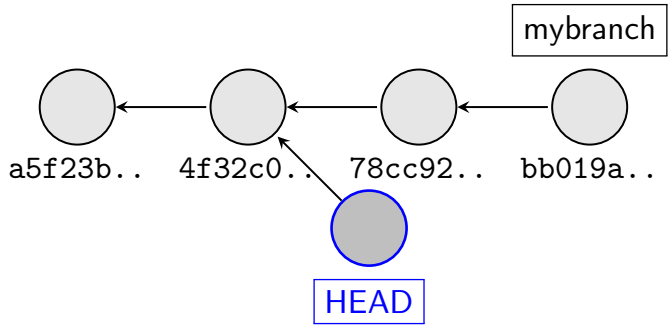
- Normalement, le WC correspond au dernier commit d'une branche (HEAD)



- Mais il peut arriver qu'on souhaite regarder un commit particulier du dépôt. On se retrouve dans l'état "*Detached HEAD*" :



- En cas de commit, on se retrouve avec une branche sans nom :



Conclusion :

Ne pas faire de commit en mode "*Detached HEAD*" !

Sommaire

FAQ

- Q : Y a-t-il des GUI pour Git ?
  R : Oui, certainement.
  Mais l'utilisation en ligne de commande permet de mieux comprendre ce qui se passe.

FAQ

- Q : Y a-t-il des GUI pour Git ?
  R : Oui, certainement.
  Mais l'utilisation en ligne de commande permet de mieux comprendre ce qui se passe.
- Q : Après avoir assimilé ce cours, quel est mon "niveau" en Git ?
  R : entre beginner et intermediate
  (voir http://gitready.com/)

FAQ

- Q : Y a-t-il des GUI pour Git ?
  R : Oui, certainement.
  Mais l'utilisation en ligne de commande permet de mieux comprendre ce qui se passe.
- Q : Après avoir assimilé ce cours, quel est mon "niveau" en Git ?
  R : entre beginner et intermediate
  (voir http://gitready.com/)
- Q : Quand faire un commit ? Quand faire un "push" ?
  R : Il faut committer assez regulièrement, typiquement plusieurs fois par jour, dès qu'on fait des "modifs significatives".
  Pour le push (propagation aux dépot connecté), il faut le faire au moins une fois par jour (en fin de journée).

FAQ

- Q : Y a-t-il des GUI pour Git ?
  R : Oui, certainement.
  Mais l'utilisation en ligne de commande permet de mieux comprendre ce qui se passe.
- Q : Après avoir assimilé ce cours, quel est mon "niveau" en Git ?
  R : entre beginner et intermediate
  (voir http://gitready.com/)
- Q : Quand faire un commit ? Quand faire un "push" ?
  R : Il faut committer assez regulièrement, typiquement plusieurs fois par jour, dès qu'on fait des "modifs significatives".
  Pour le push (propagation aux dépot connecté), il faut le faire au moins une fois par jour (en fin de journée).
- Q : Quand créer une branche ?
  R : Dès qu'on souhaite garder une "ligne principale" propre.
  A retenir : la création d'une branche a un cout nul !

Références

- En français :
  - https://git-scm.com/book/fr/v2
  - http://djibril.developpez.com/tutoriels/conception/pro-git
  - https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git
  - https://blog.squad.fr/non-classe/git-explique-avec-mes-mots.html
- En anglais :
  - http://nvie.com/posts/a-successful-git-branching-model/
  - http://gitready.com/
  - http://onlywei.github.io/explain-git-with-d3/
  - http://marklodato.github.io/visual-git-guide/index-en.html
  - https://www.atlassian.com/git/tutorials