Formation VCS

Comment organiser et partager proprement le code de vos projets

Bordeaux INP ENSEIRB MATMECA

Anthony Gautier http://silica.io

01/12/16

Sommaire

- Buts d'un VCS
- Fonctionnement dans la pratique
- Installation et utilisation de Git
- Les branches
- Points particuliers

Buts d'un VCS

« Encore un truc compliqué qui sert à rien… »

… pas tout à fait.

Buts d'un VCS

- **VCS = *Version Control System***
- Centralise tous les fichiers du code source d'un projet
- Facilite :
  - le partage
  - la traçabilité
  - l'évolutivité du code

Buts d'un VCS

- **VCS = *Version Control System***
- Centralise
  d'un proje
- Facilite :
  - le partage
  - la traçabilité
  - l'évolutivité du code

On est plusieurs à travailler sur le projet, mais c'est compliqué de se mettre d'accord pour ne pas modifier les mêmes fichiers en même temps…

Buts d'un VCS

- ***VCS = Version Control System***
- Centralise tous les fichiers du code source d'un projet
- Facilite :
  - le partage
  - la traçabilité
  - l'évolutivité du code

Quand est-ce que cette fonction a été modifiée ? Par qui ? Et pourquoi ?

Buts d'un VCS

- rol System
- chiers du code source
-
- la traçabilité
- l'évolutivité du code

On doit travailler sur plusieurs fonctionnalités en même temps, mais elles se seront pas compatibles entre elles tant qu'elles ne seront pas terminées… Comment je fais pour les tester indépendamment?

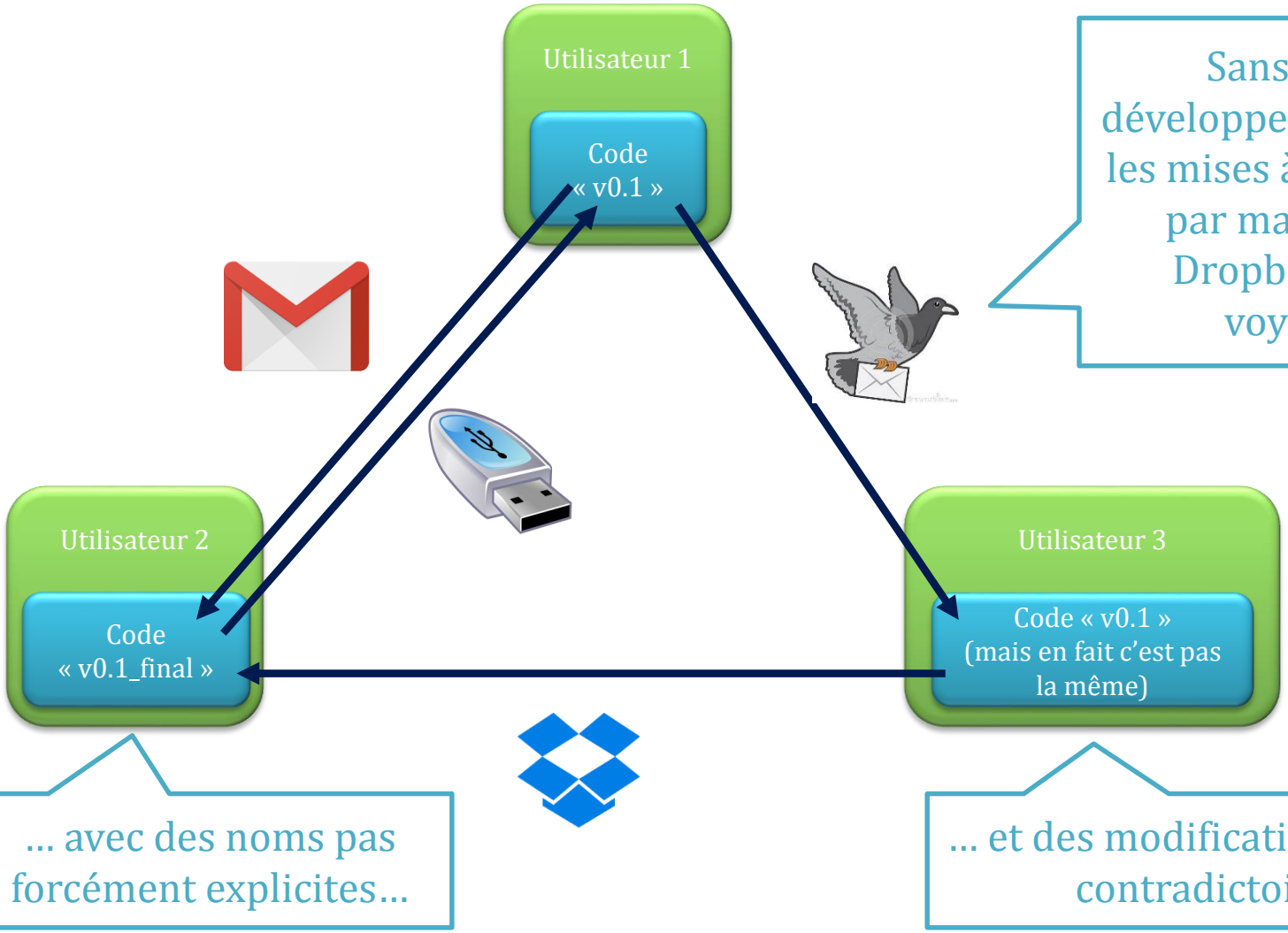
Je dois reprendre un projet écrit par quelqu'un d'autre, mais je ne sais pas où il en était ni quelle logique il a suivi dans le développement…

Buts d'un VCS

- **VCS = *Version C***
- Centralise tous
  d'un projet
- Facilite :
  - le partage
  - la traçabilité
  - l'évolutivité du code

Un VCS résout tous ces problèmes !

Partage de code, sans VCS



Partage de code, sans VCS

Utilisateur 1

Code
« v0.1 »

Sans VCS, les développeurs s'envoient les mises à jour du code par mail, clé USB, Dropbox, par-dessus l'épaule.

… où est la dernière version ?

Utilisateur 2

Code
« v0.1_final »

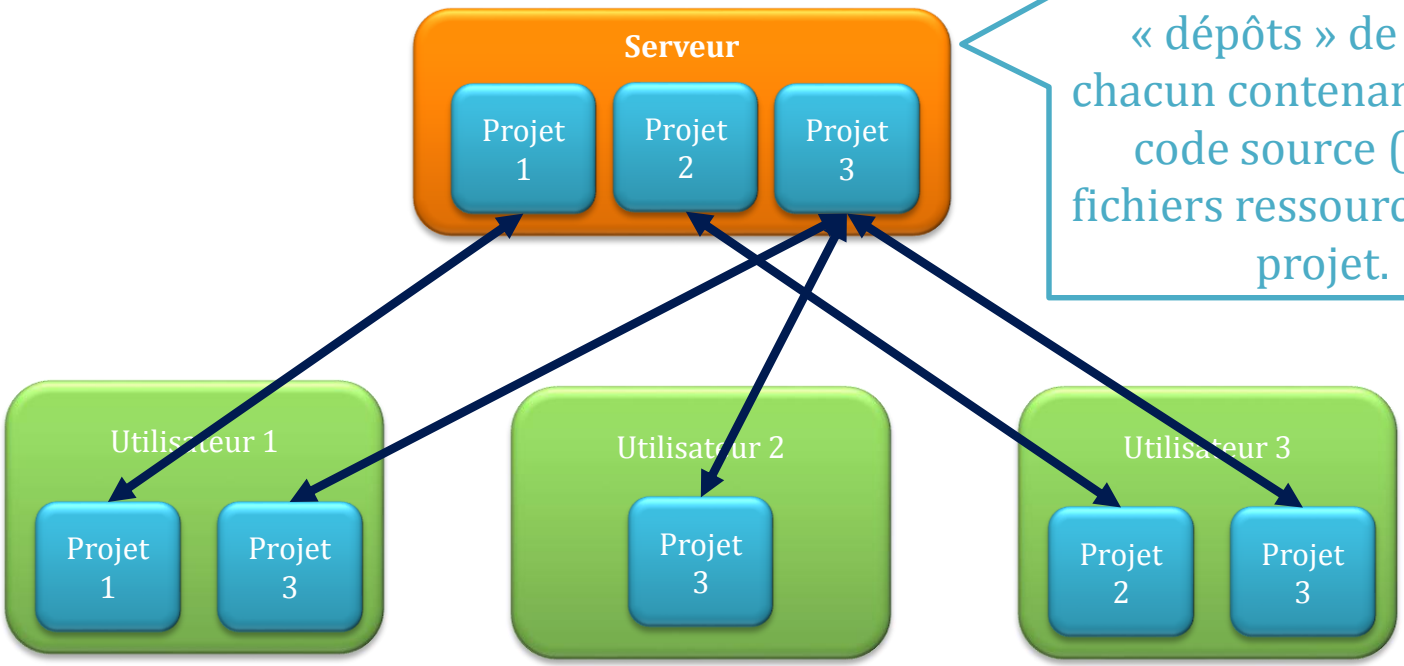
Utilisateur 3

Code « v0.1 »
(mais en fait c'est pas la même)

… avec des noms pas forcément explicites…

… et des modifications parfois contradictoires.

Partage de code, avec VCS



Un serveur héberge des « dépôts » de code, chacun contenant tout le code source (et les fichiers ressources) d'un projet.

Tous les développeurs disposent d'une copie synchronisée de ces dépôts.

En plus des fichiers de code, les dépôts contiennent notamment un historique des modifications apportées au projet, et bien d'autres choses.

Fonctionnement dans la pratique

C’est bien beau tout ça, mais concrètement, « je clique où ? »

Fonctionnement dans la pratique

Exemple : le projet **Artemis** *

* De manière générale, évitez les accents et caractères spéciaux dans les noms de projets

Fonctionnement dans la pratique

A noter :

- Les commandes présentées par la suite sont génériques et ne sont pas propres à un logiciel en particulier. Quel que soit le logiciel que vous utiliserez, le fonctionnement sera similaire. Dans la partie suivante, nous verrons plus précisément comment utiliser l'un de ces logiciels, **Git**.
- Un dépôt peut être hébergé soit à distance sur un serveur, soit localement : un serveur permet de partager plus facilement le code, alors qu'un dépôt local permet de tirer profit facilement de toutes les fonctionnalités du VCS, même pour un projet sur lequel on est le seul à travailler.

Fonctionnement dans la pratique

A noter :

- Les commandes présentées par la suite sont génériques et ne sont pas propres à un logiciel en particulier. Quel que soit le logiciel que vous utiliserez, le fonctionnement sera similaire. Dans la partie suivante, nous verrons plus précisément comment utiliser l'un de ces logiciels, **Git**.
- Un dépôt peut être hébergé soit à distance sur un serveur, soit localement : un serveur permet de partager plus facilement le code, alors qu'un dépôt local permet de tirer profit facilement de toutes les fonctionnalités du VCS, même pour un projet sur lequel on est le seul à travailler.

Fonctionnement dans la pratique

0) Au début :

Vous avez créé quelques fichiers pour votre projet sur votre disque dur, et éventuellement commencé à écrire un peu de code dedans. Vous avez accès à un *serveur*, qui peut être un autre ordinateur ailleurs ou simplement un autre dossier sur votre disque dur (*dépôt local*).

Fonctionnement dans la pratique

1) Création du projet :

init

Que le serveur soit local ou distant, il faut dans tous les cas commencer par initialiser le dépôt vide. Cette opération crée l'ensemble des paramètres dont le VCS aura besoin par la suite.

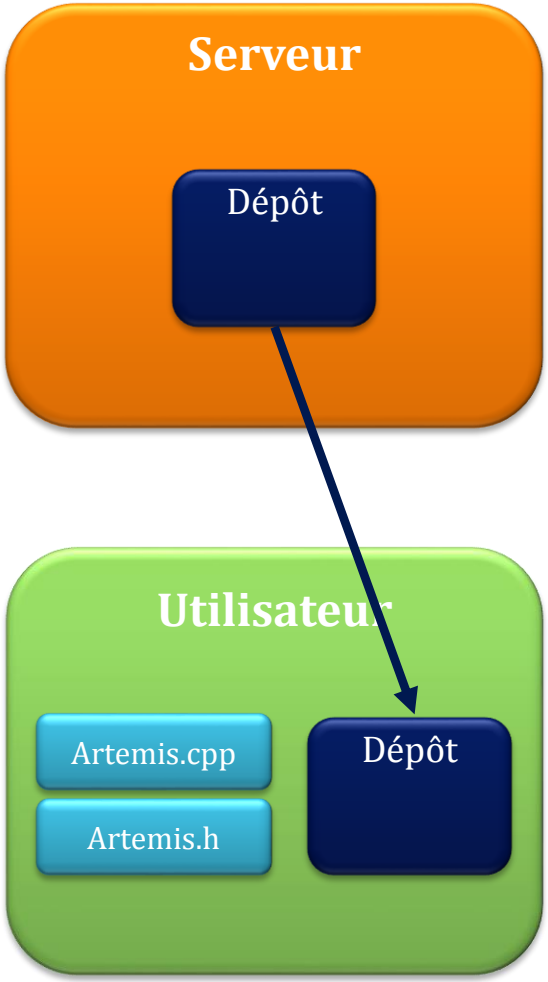
Fonctionnement dans la pratique

2) Synchronisation :

clone

Si le dépôt est distant, exécutez sur votre ordinateur la commande *clone* avec l'adresse du serveur et le chemin du dépôt, afin d'en faire une copie locale.

Un nouveau dossier vide est alors créé pour ce dépôt.

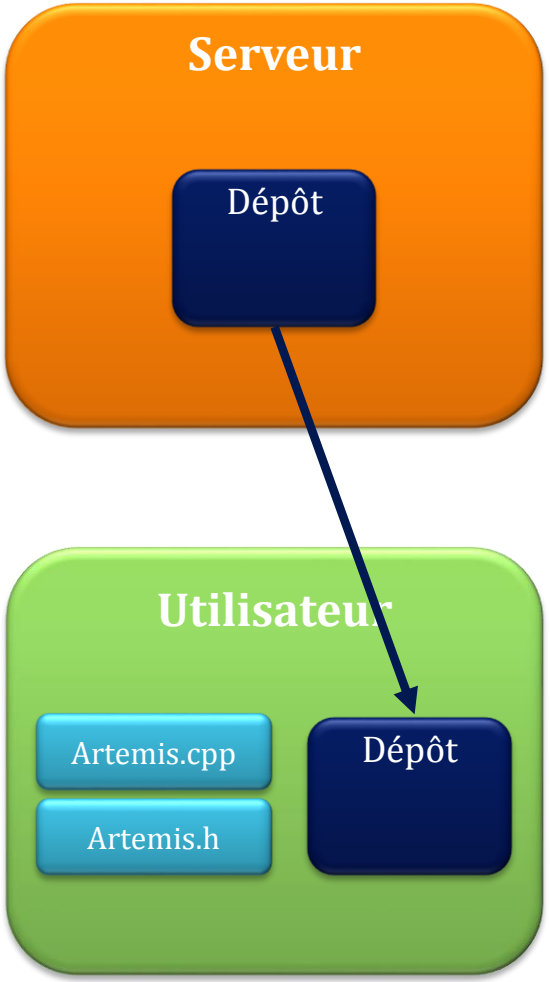


Fonctionnement dans la pratique

2) Synchronisation :

clone

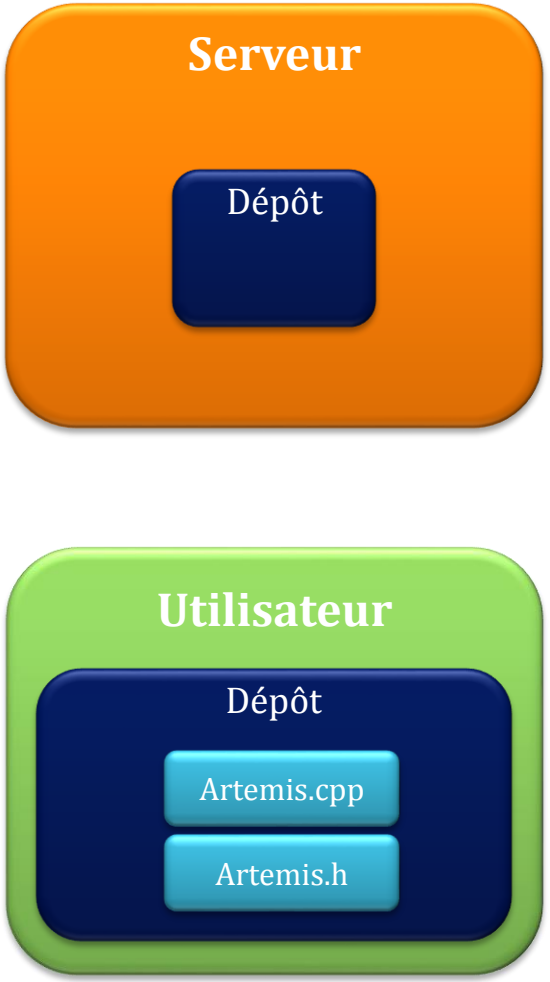
Si vous utilisez un dépôt local, pas besoin de clone, vous utiliserez directement le dépôt créé par *init*. Vous n'aurez alors pas besoin de *pull* ni *push*, exposés plus loin.



Fonctionnement dans la pratique

3) Création de fichiers :

Créez les fichiers de base de votre code (ou copiez des fichiers de code existant) dans le dossier du dépôt.



Fonctionnement dans la pratique

4) Ajout du suivi :

add

Les nouveaux fichiers ne sont pas suivis (« tracked ») par le VCS par défaut, il faut les ajouter avec la commande *add*.

Fonctionnement dans la pratique

Facultatif

5) Etat du dépôt :

status

La commande *status* vous indique quelles modifications ont été faites dans votre dépôt depuis le dernier état enregistré : fichiers ou dossiers créés, modifiés ou supprimés.

```
Status :
A Artemis.cpp
A Artemis.h
```

Fonctionnement dans la pratique

Facultatif

5) Etat du dépôt :

status

En général, les nouveaux fichiers apparaissent en vert (A), les fichiers modifiés en orange (M) et les fichiers supprimés en rouge (D).

```
Status :
A Artemis.cpp
A Artemis.h
```

Fonctionnement dans la pratique

Facultatif

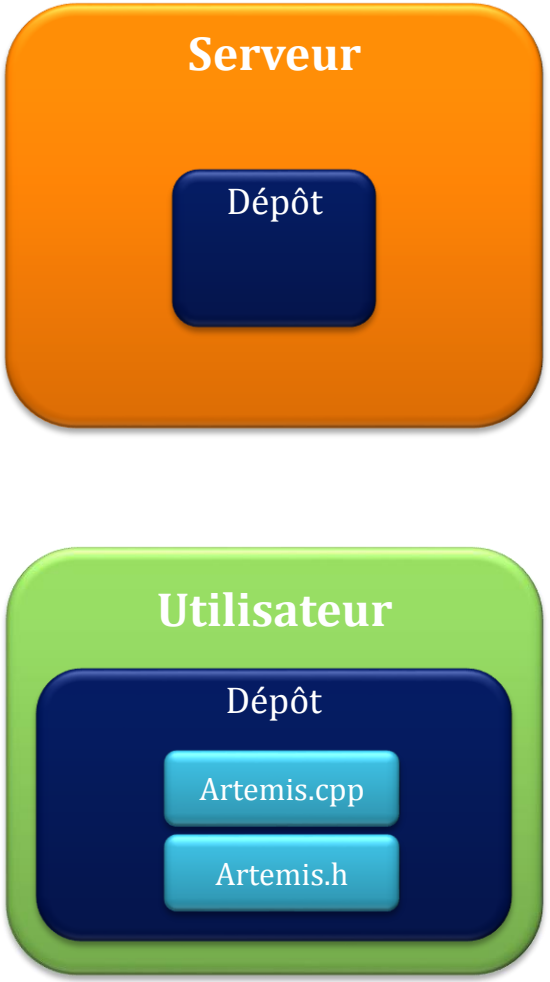
5) Etat du dépôt :

status

Si aucune modification n'a été détectée, *status* vous indiquera que « l'arbre de travail est propre ».

`Status :`

```
Working directory clean
```



Fonctionnement dans la pratique

Facultatif

6) Analyse des modifications :

diff

diff vous indique, pour chaque fichier, quelles lignes ont été modifiées.

Artemis.cpp :

```
5: - int main() {
5: + int main(int argc, char** argv) {
6: +     printf("Hello world");
```

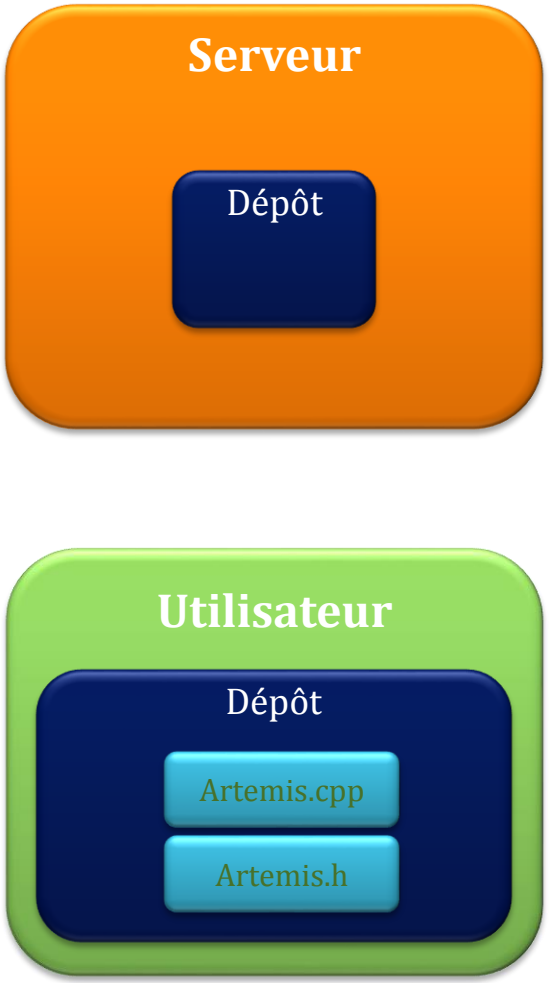
Fonctionnement dans la pratique

Important

7) Validation des modifications :

commit

Lorsque vous avez fini d'effectuer un ensemble de modifications **cohérent** (ajout d'une fonction par exemple), *commit* permet de valider ces changements et de créer une nouvelle version du projet.



Fonctionnement dans la pratique

Important

7) Validation des modifications :

commit

Cette commande vous rappelle les modifications effectuées (résultat de *status*) et vous demande un « message de commit », qui est un résumé des modifications effectuées.

Il est **obligatoire**, soyez **clair** et **précis**.

Une fois cette étape terminée, *status* vous indiquera qu'il n'y a plus de modifications non-validées.

Fonctionnement dans la pratique

Facultatif

8) Historique du projet :

log

La commande *log* vous indique la liste des modifications effectuées sur le code depuis la création du dépôt.

Pour chaque modification, la date, l'auteur et le message de commit sont indiqués.

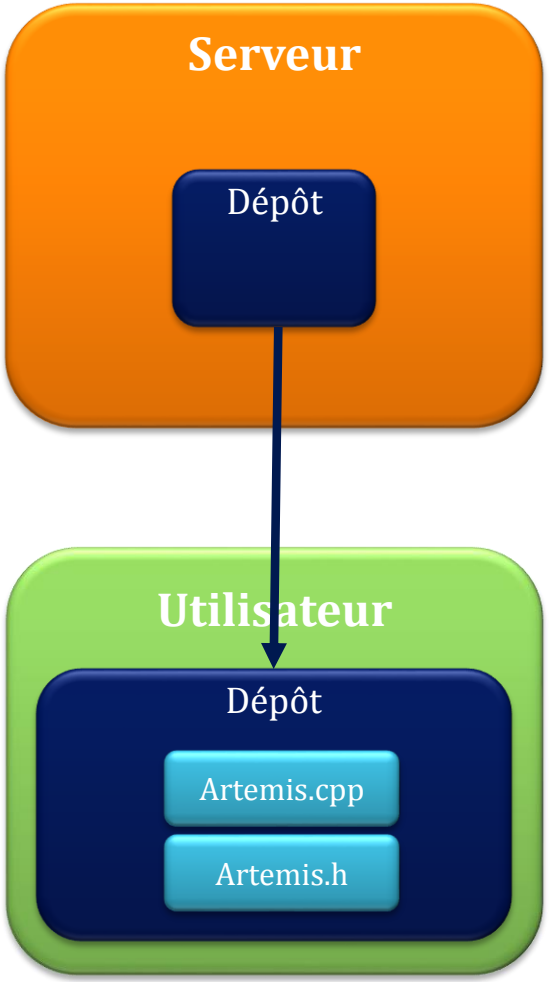
Vous pouvez également demander plus d'informations sur un commit particulier : liste des fichiers modifiés, numéros des lignes affectées, etc.

Fonctionnement dans la pratique

9) Mise à jour du dépôt :

pull

Avant d'envoyer vos modifications sur le serveur, assurez-vous que personne d'autre n'en a fait en même temps que vous, en mettant à jour votre dépôt local avec *pull*.

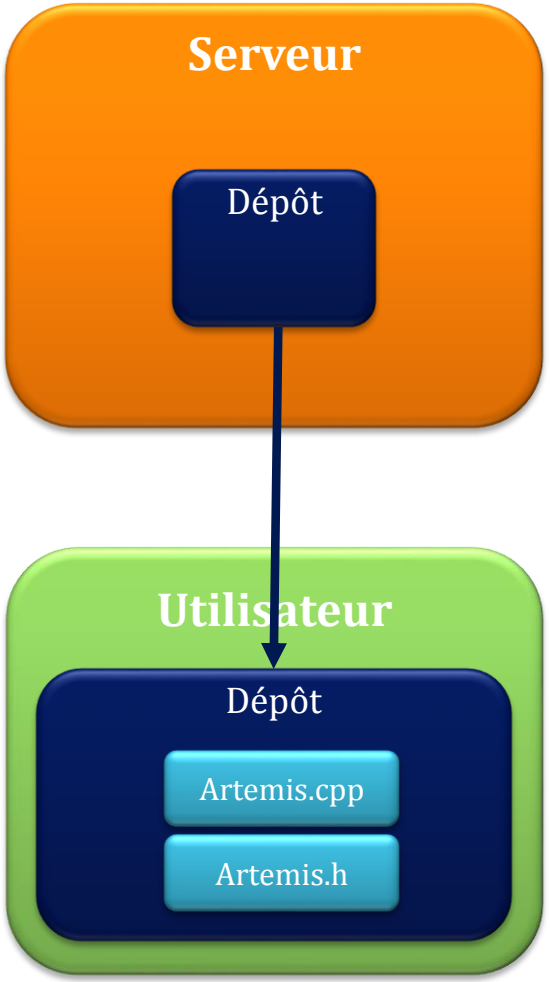


Fonctionnement dans la pratique

9) Mise à jour du dépôt :

pull

Si des modifications ont été faites sur le dépôt par d'autres utilisateurs, le VCS va tenter de les incorporer au mieux dans votre version : on appelle cette opération un *merge*.

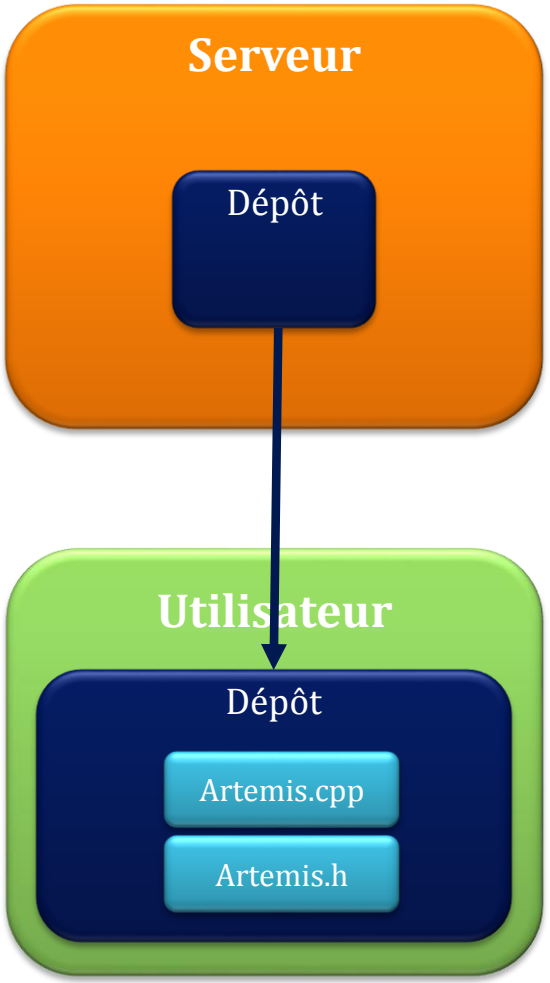


Fonctionnement dans la pratique

9) Mise à jour du dépôt :

pull

Dans le cas où une même ligne d'un même fichier a été modifiée par les deux utilisateurs, un conflit est détecté, et la procédure s'arrête. Il vous est alors demandé de résoudre manuellement le conflit en sélectionnant quelle version conserver parmi les deux existantes.



Fonctionnement dans la pratique

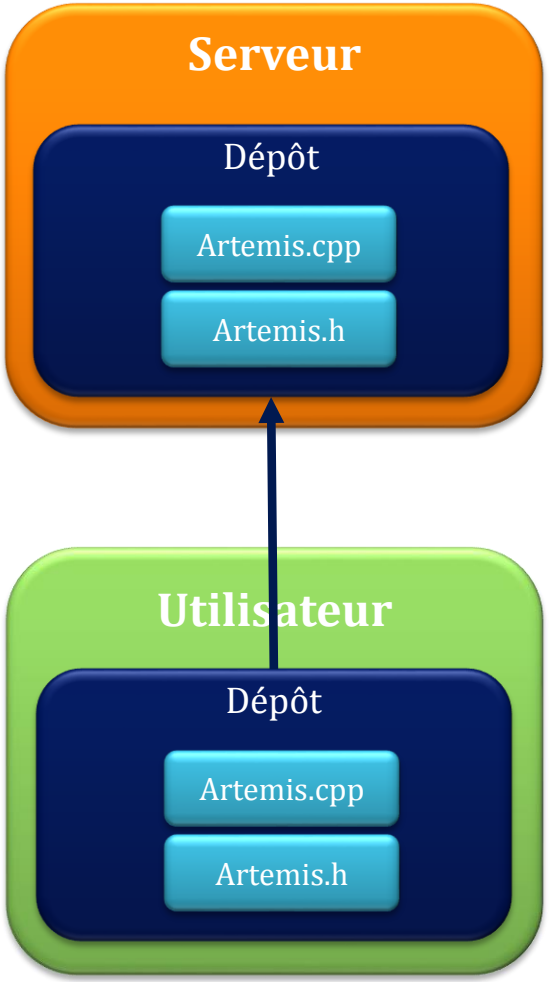
10) Envoi des modifications :

push

La commande push envoie sur le serveur tous les commits que vous avez effectué localement et qu'il ne connait pas encore.

Attention :

- **Ne faites de *push* que sur un dépôt clean**, car les modifications non-commitées ne sont pas envoyées.
- **Faites toujours un *pull* juste avant un *push***, car en cas de conflit, le *merge* ne peut pas être effectué sur le serveur.

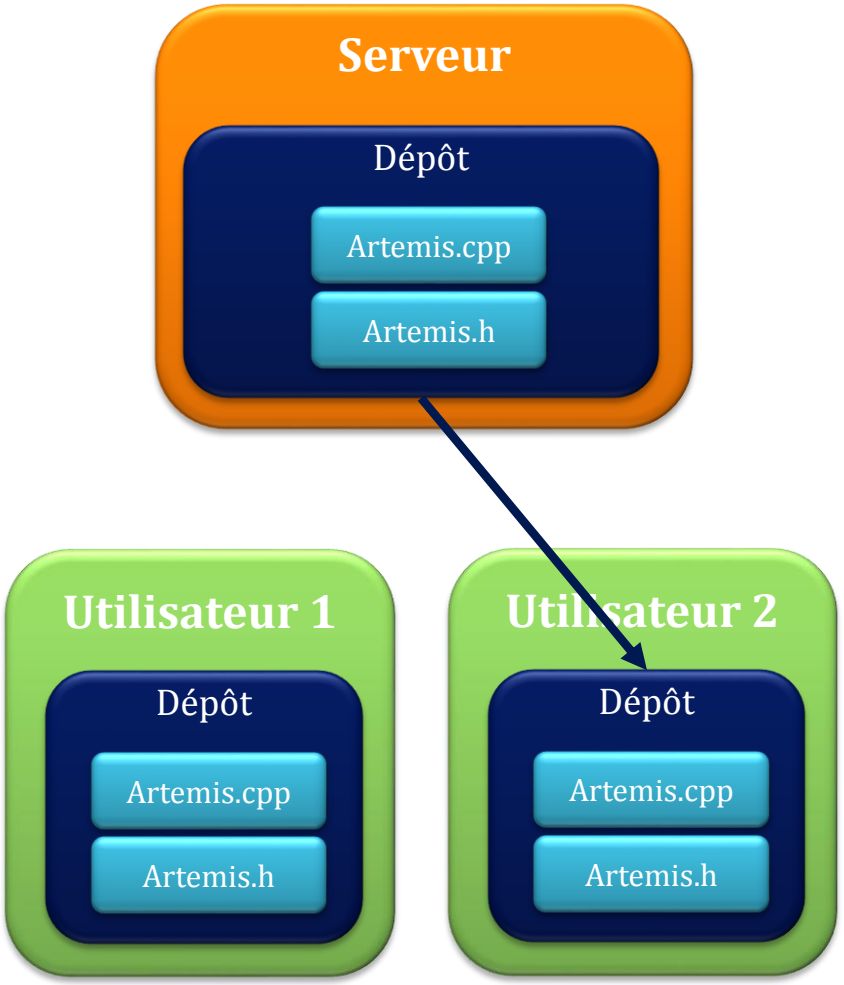


Fonctionnement dans la pratique

11) Partage du projet :

clone

Pour partager le projet, il suffit de répéter *clone* sur les ordinateurs de tous les utilisateurs, et de suivre la même procédure : *commit*, *pull*, *push*.



Fonctionnement dans la pratique

Résumé :

Initialisation du projet

- *init*
- *clone*

Session de travail

- *pull* — Mettez à jour le dépôt avant de commencer à modifier le code
- modifications…
- *status*
- *commit*

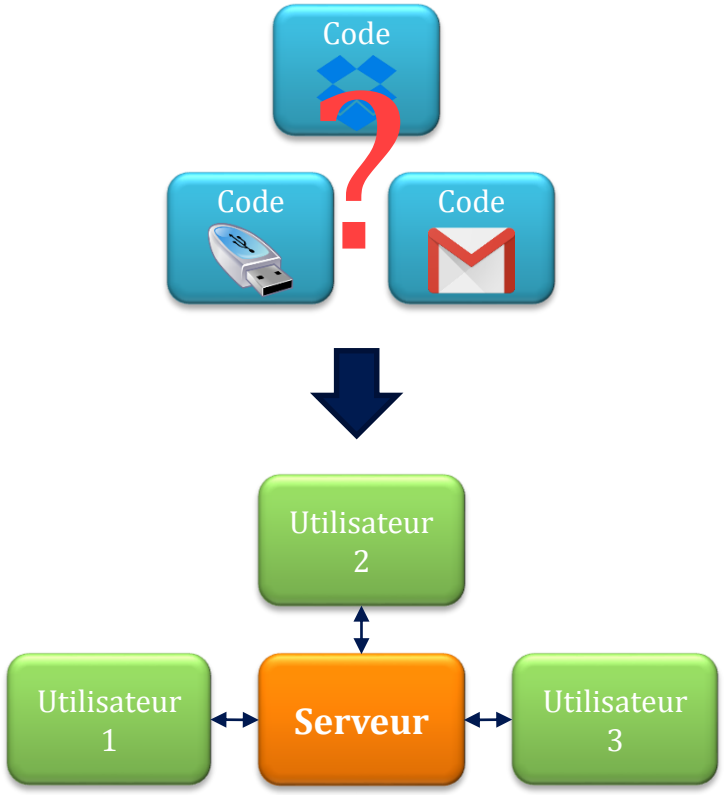
  Pour chaque nouvelle fonction ajoutée dans le code
- *pull*
- *push*

  Avant de partir, envoyez vos changements sur le serveur

Fonctionnement dans la pratique

Ce fonctionnement permet un suivi clair du projet :

1) Le dépôt représente **la** version actuelle de référence du code.



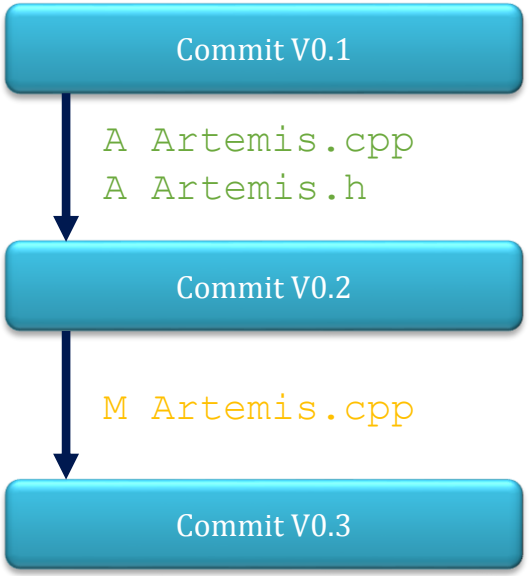
Fonctionnement dans la pratique

Pensez à faire un *push* à la fin d'une session de travail !

Fonctionnement dans la pratique

Ce fonctionnement permet un suivi clair du projet :

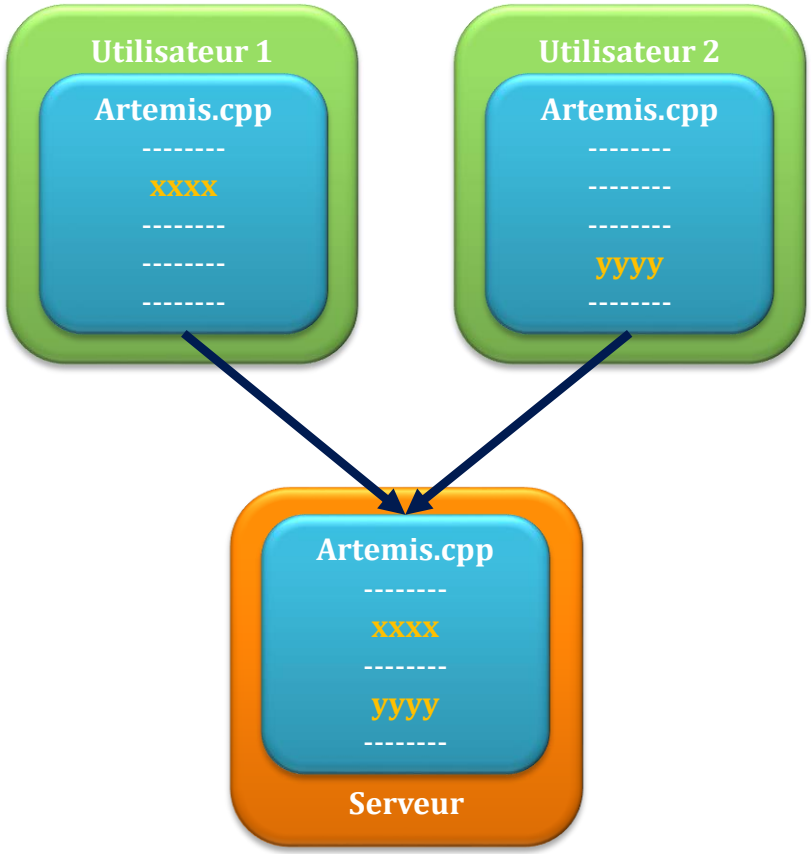
2) Vous avez un historique précis de toutes les versions successives du code.



Fonctionnement dans la pratique

Ce fonctionnement permet un suivi clair du projet :

3) Les modifications concurrentes d'un même fichier sont automatiquement gérées.



Installation et utilisation de Git

Cool, un TP!

Exemples de VCS

git

mercurial

Bazaar

SUBVERSION

Récents, fonctionnalités avancées, dans l'ensemble similaires

Plus ancien, moins puissant, utilisé pour des raisons historiques

Installation sur Windows

- Téléchargez le portage suivant :

http://git-scm.com/download/win

- Pendant l'installation, sélectionnez
  - « Windows Explorer integration »
    \ « Simple context menu »
  - « Use Git from Git Bash only »
  - « Checkout Windows-style, commit Unix-style »
- Pour lancer Git, faites clic droit sur un dossier et sélectionnez « Git Bash here »

Installation sur Linux

```
$ sudo apt-get install git
```

… c'est tout.

Exemple

```
~ $ ssh git@eirspace.fr
git@eirspace.fr:~ $ cd /home/eirspace/
git@eirspace.fr:/home/eirspace $ mkdir artemis
git@eirspace.fr:/home/eirspace $ cd artemis
git@eirspace.fr:/home/eirspace/artemis $ git init
Initialized empty Git repository in /home/eirspace/artemis/.git/
```

Sur le serveur, on crée un nouveau dossier pour le dépôt et on utilise *init* pour l'initialiser.

La procédure pour créer un dépôt local est identique, sans la commande ssh.

Exemple

```
~ $ git clone ssh://git@eirspace.fr/home/eirspace/artemis
Cloning into 'artemis'...
git@eirspace.fr's password:
warning: You appear to have cloned an empty repository.
```

On clone le nouveau dépôt

Pour un dépôt local, inutile de spécifier « ssh://*user@serveur*/ », il suffit d'indiquer le chemin du dossier du dépôt

Exemple

```
~ $ git clone ssh://git@eirspace.fr/home/eirspace/artemis
Cloning into 'artemis'...
git@eirspace.fr's password:
warning: You appear to have cloned an empty repository.

~ $ cd artemis/

~/artemis $ vim artemis.cpp
int main() {
}
```

On clone le nouveau dépôt

On créé un nouveau fichier

Exemple

```
~ $ git clone ssh://git@eirspace.fr/home/eirspace/artemis
Cloning into 'artemis'...
git@eirspace.fr's password:
warning: You appear to have cloned an empty repository.

~ $ cd artemis/

~/artemis $ vim artemis.cpp
int main() {
}

~/artemis $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#         artemis.cpp
nothing added to commit but untracked files present (use "git add" to track)
```

On clone le nouveau dépôt

On créé un nouveau fichier

La commande *status* nous indique que ce nouveau fichier est détecté mais n'est pas encore suivi

Exemple

```
~ $ git clone ssh://git@eirspace.fr/home/eirspace/artemis
Cloning into 'artemis'...
git@eirspace.fr's password:
warning: You appear to have cloned an empty repository.

~ $ cd artemis/

~/artemis $ vim artemis.cpp
int main() {
}

~/artemis $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#         artemis.cpp
nothing added to commit but untracked files present (use "git add" to track)

~/artemis $ git add artemis.cpp
```

On clone le nouveau dépôt

On créé un nouveau fichier

La commande *status* nous indique que ce nouveau fichier est détecté mais n'est pas encore suivi

On l'ajoute

Exemple

```
~/artemis $ git commit -a
> Ajout du main
[master (root-commit) 585af1d] Ajout du main
 1 file changed, 2 insertions(+)
 create mode 100644 artemis.cpp

~/artemis $ git status
# On branch master
nothing to commit (working directory clean)
```

L'option « -a » (« all ») de *commit* permet de sélectionner tous les fichiers modifiés

On spécifie « Ajout du main » comme message de commit

Exemple

```
~/artemis $ git commit -a
> Ajout du main
[master (root-commit) 585af1d] Ajout du main
 1 file changed, 2 insertions(+)
 create mode 100644 artemis.cpp

~/artemis $ git status
# On branch master
nothing to commit (working directory clean)
```

L'option « -a » (« all ») de *commit* permet de sélectionner tous les fichiers modifiés

On spécifie « Ajout du main » comme message de commit

Le pull ici est omis pour des raisons de clarté

Exemple

```
~/artemis $ git commit -a
> Ajout du main
[master (root-commit) 585af1d] Ajout du main
 1 file changed, 2 insertions(+)
 create mode 100644 artemis.cpp

~/artemis $ git status
# On branch master
nothing to commit (working directory clean)

~/artemis $ git push origin master
git@eirspace.fr's password:
Counting objects: 3, done.
Writing objects: 100% (3/3), 237 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@eirspace.fr/home/eirspace/artemis
 * [new branch]      master -> master
```

L'option « -a » (« all ») de *commit* permet de sélectionner tous les fichiers modifiés

On spécifie « Ajout du main » comme message de commit

Le pull ici est omis pour des raisons de clarté

Pour le premier commit uniquement, il faut préciser « origin master » au *push*

Exemple

```
~/artemis $ git commit -a
> Ajout du main
[master (root-commit) 585af1d] Ajout du main
 1 file changed, 2 insertions(+)
 create mode 100644 artemis.cpp

~/artemis $ git status
# On branch master
nothing to commit (working directory clean)

~/artemis $ git push origin master
git@eirspace.fr's password:
Counting objects: 3, done.
Writing objects: 100% (3/3), 237 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@eirspace.fr/home/eirspace/artemis
 * [new branch]      master -> master

~/artemis $ git push
git@eirspace.fr's password:
Everything up-to-date

~/artemis $ git pull
git@eirspace.fr's password:
Already up-to-date.
```

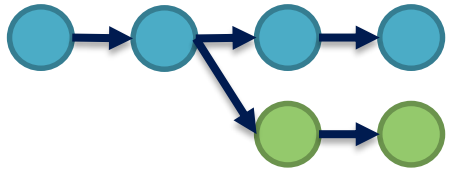
L'option « -a » (« all ») de *commit* permet de sélectionner tous les fichiers modifiés

On spécifie « Ajout du main » comme message de commit

Le pull ici est omis pour des raisons de clarté

Pour le premier commit uniquement, il faut préciser « origin master » au *push*

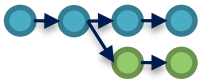
Le dépôt local est bien synchronisé avec le serveur



Les branches

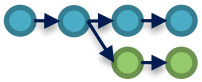
Toute la puissance d'un VCS moderne

Les branches



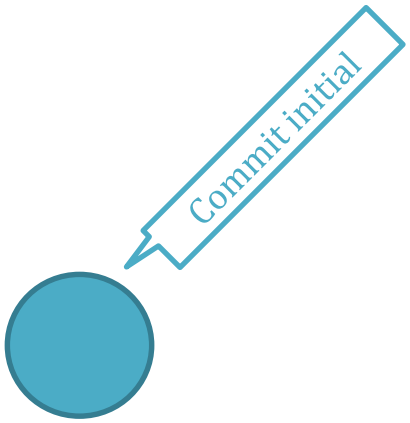
- Souvent, lorsqu'un projet devient complexe, son développement n'est pas linéaire :
  - Vous travaillez sur plusieurs fonctionnalités indépendantes en même temps
  - Alors que vous êtes au milieu d'importants changements dans le code, on vous demande de corriger un bug urgent dans la version du code en production (mais la version actuelle est instable)
- Les VCS récents proposent un système pour faciliter ces opérations : les **branches**

Les branches



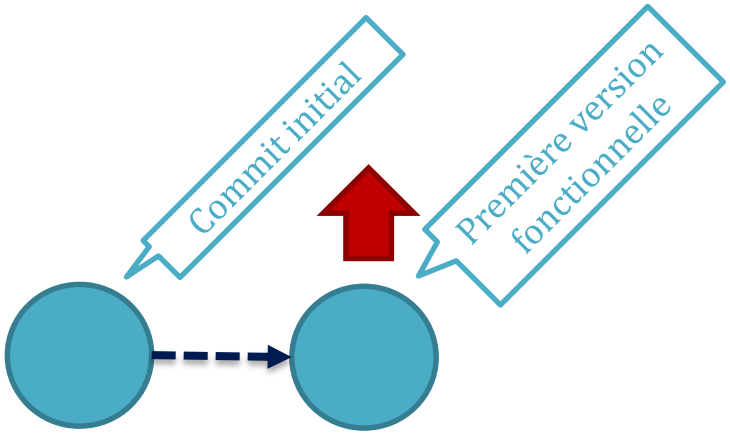
- Une branche est un **ensemble de commits successifs**
- Les branches sont **indépendantes**
- Vous pouvez très facilement passer d'une branche à l'autre. Le VCS sauvegardera le code de la branche courante et le remplacera par le code de l'autre branche

Les branches – exemple



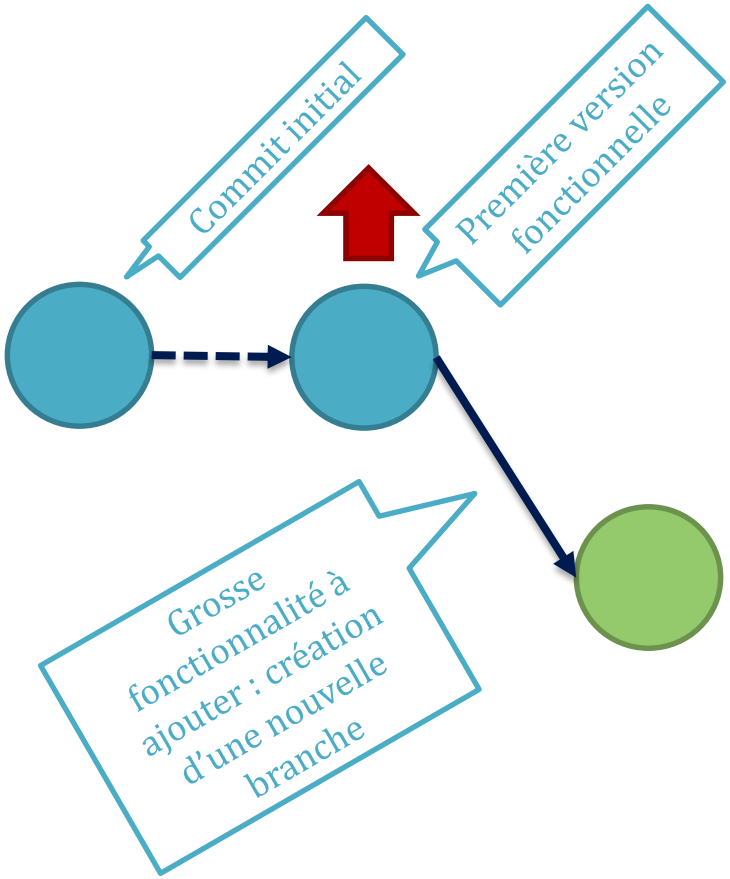
Par défaut, le VCS crée la branche *master* pour votre code, c'est celle sur laquelle vous travaillez immédiatement après la création du dépôt.

Les branches – exemple



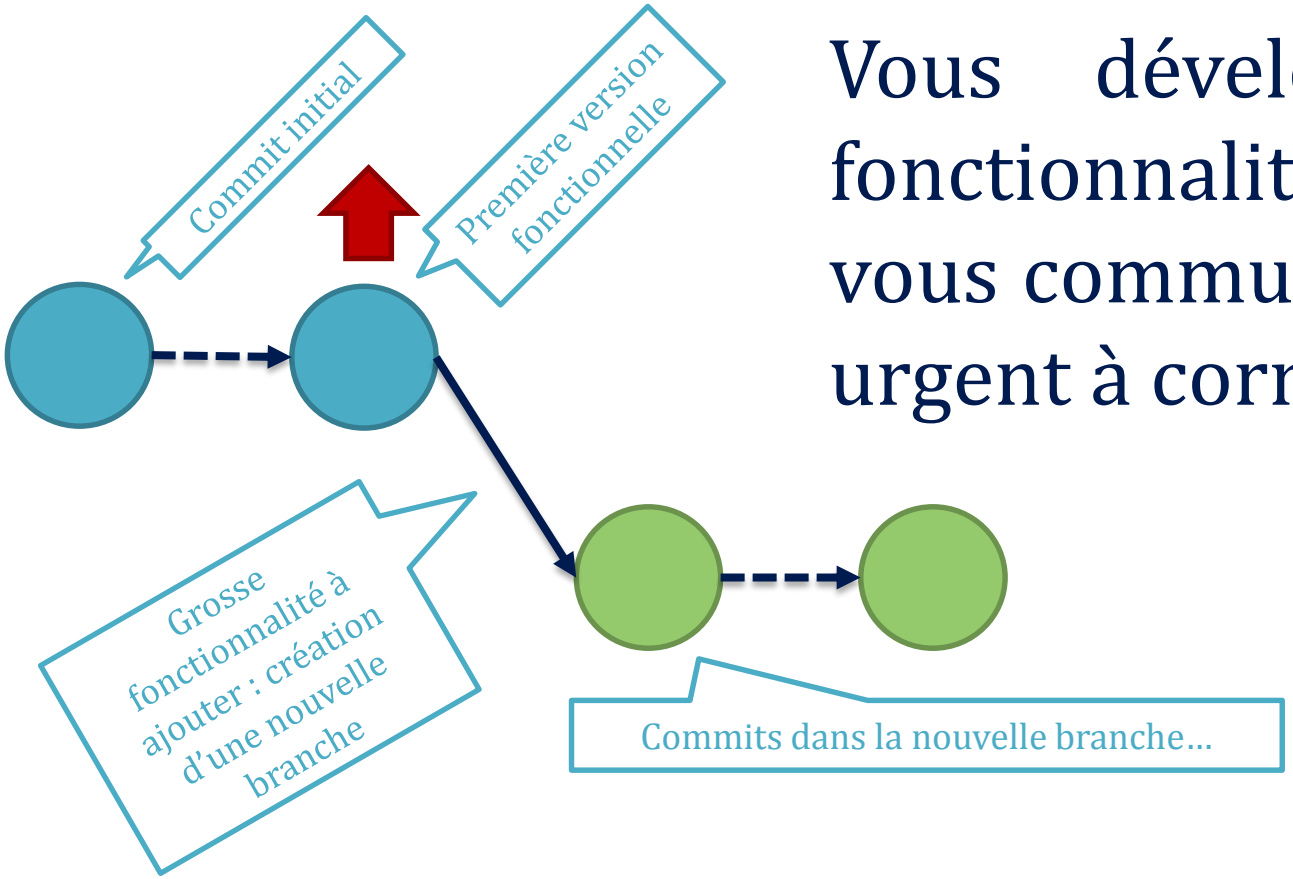
Vous faites des commits et avez une première version fonctionnelle, que vous publiez.

Les branches – exemple



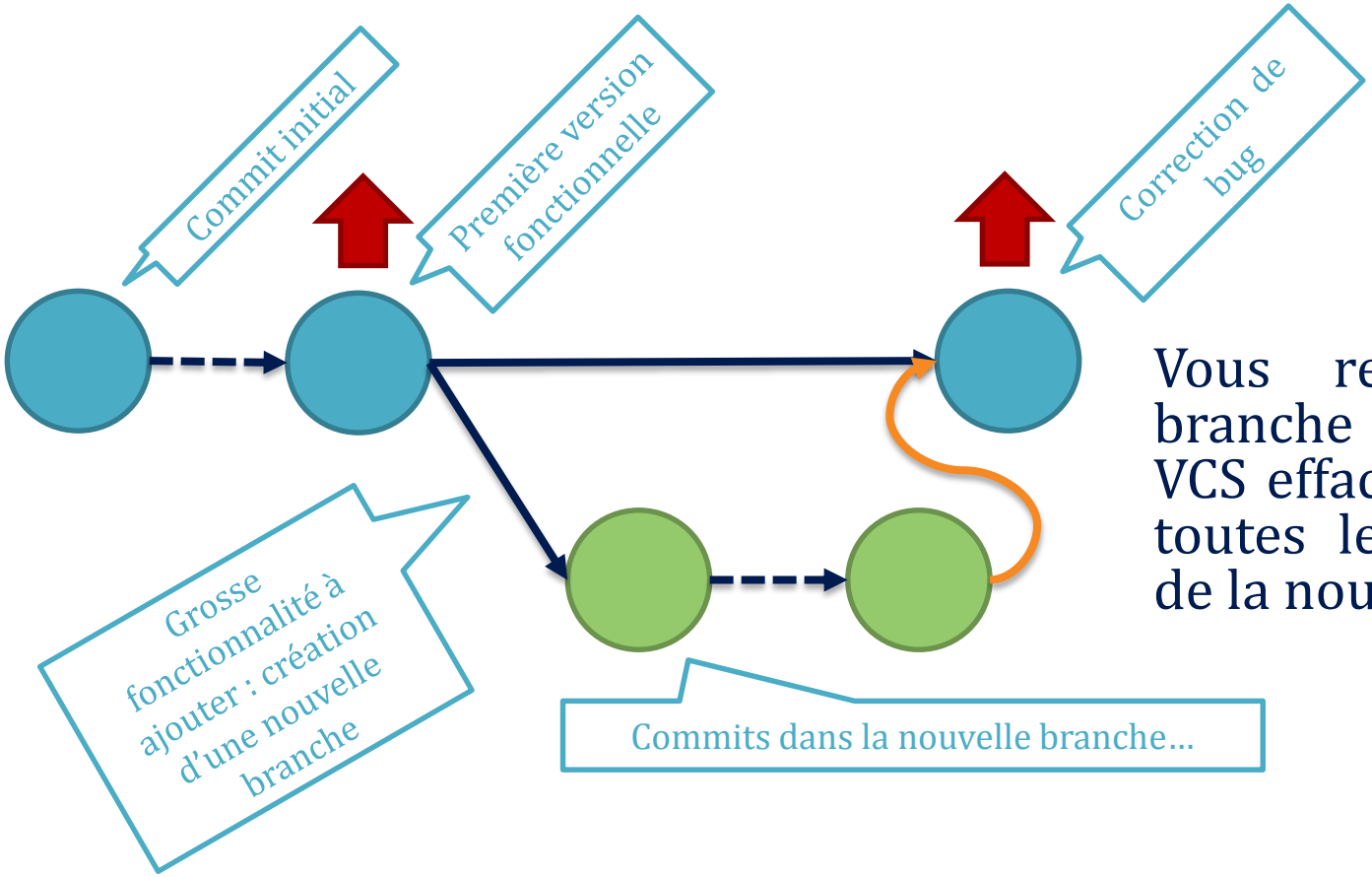
L'ajout d'une grosse fonctionnalité est prévue : vous créez une nouvelle branche dédiée à celle-ci.

Les branches – exemple



Vous développez cette fonctionnalité… quand on vous communique un bug urgent à corriger.

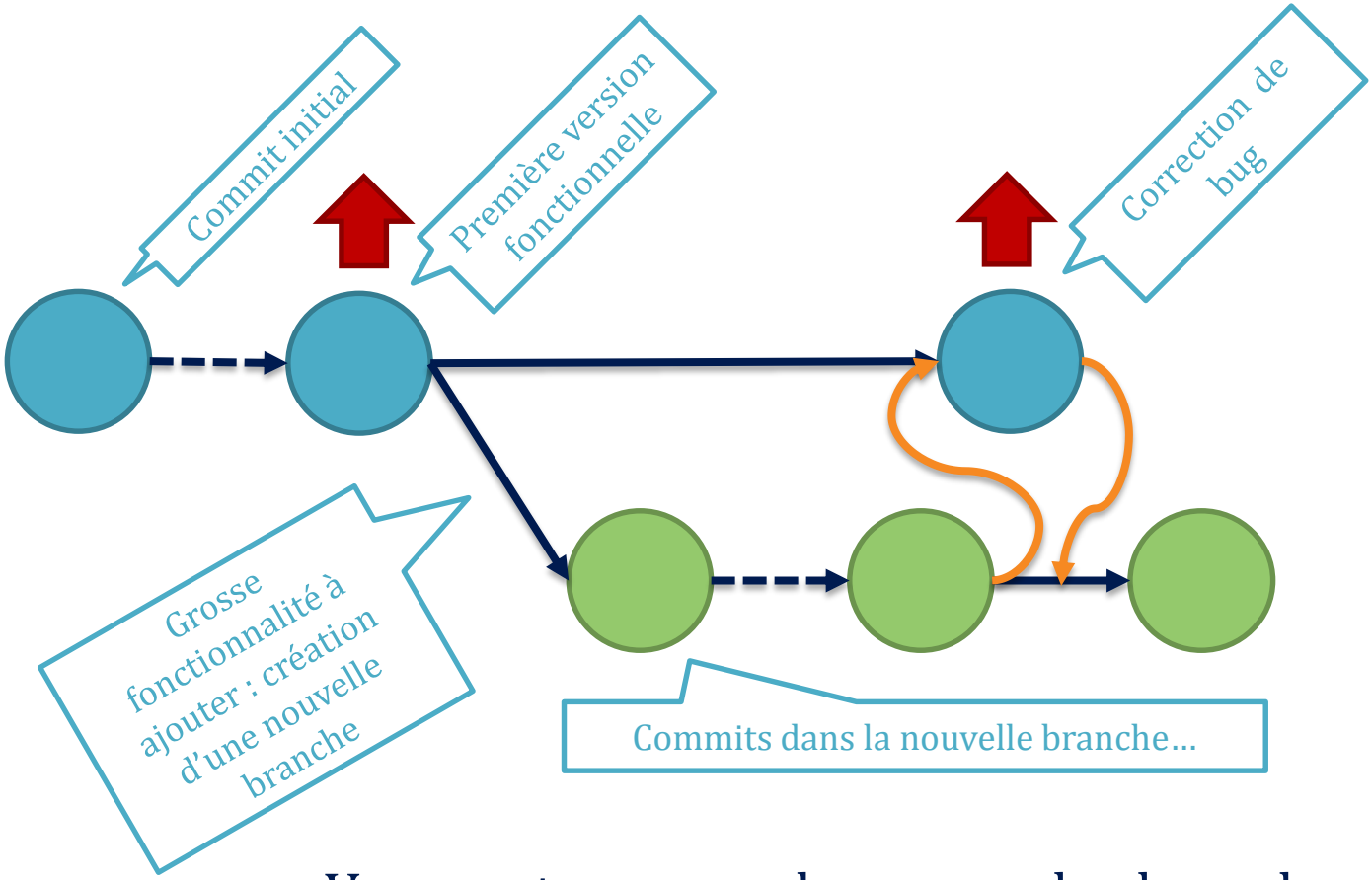
Les branches – exemple



Vous revenez sur la branche principale. Le VCS efface de votre code toutes les modifications de la nouvelle branche.

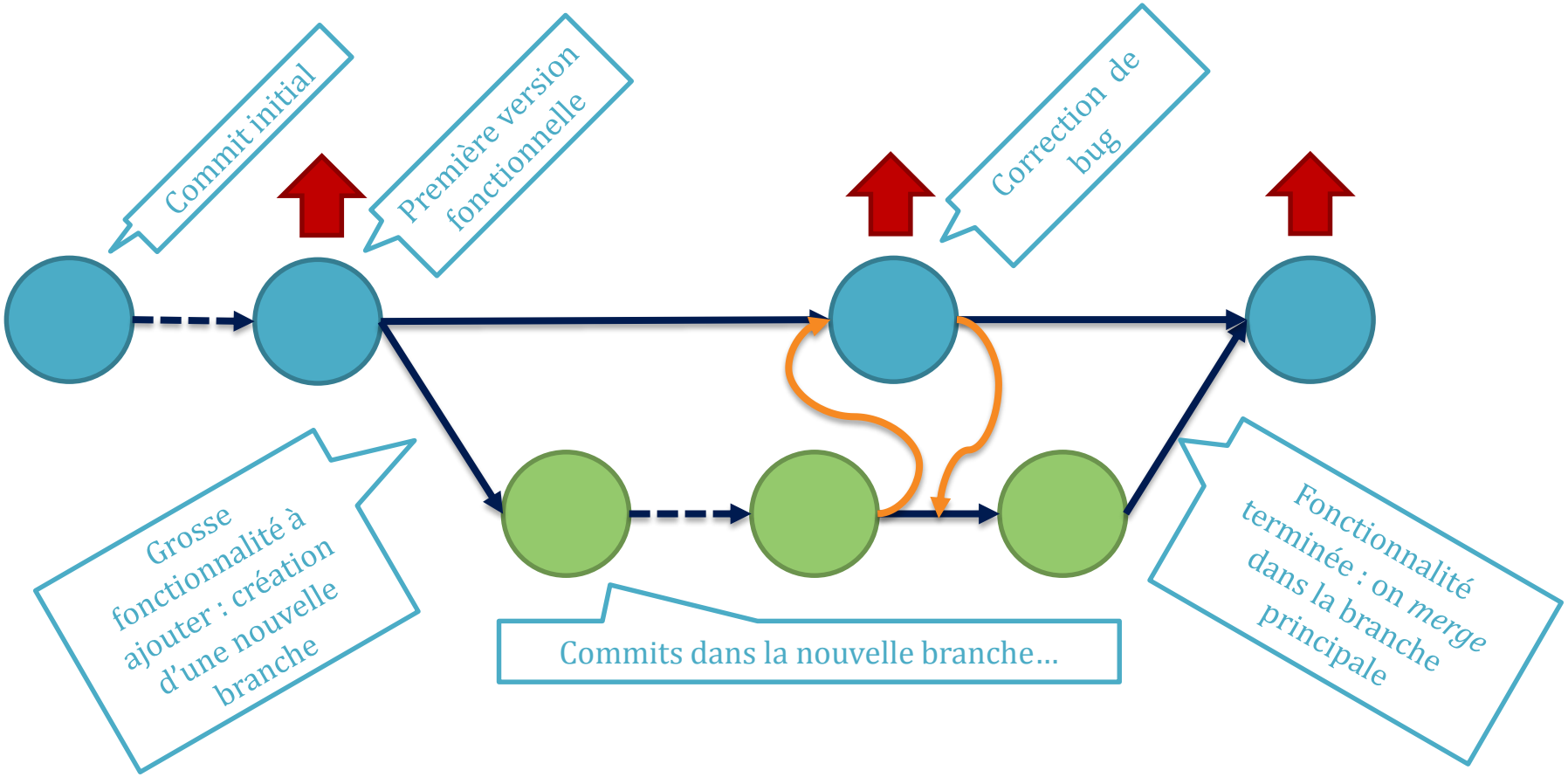
Vous pouvez alors corriger le bug et publier le correctif.

Les branches – exemple



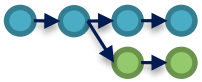
Vous retournez alors sur la branche de travail et continuez de développer la nouvelle fonctionnalité.

Les branches – exemple



Quand la fonctionnalité est terminée, vous l'ajoutez à la version en production en *mergeant* la branche de travail dans la branche principale et vous publiez le résultat.

Les branches



- Vous pouvez :
  - créer autant de branches que vous voulez
  - *merger* n'importe quelle branche dans n'importe quelle autre
  - supprimer une branche sans la *merger* si vous abandonnez cette fonctionnalité
  - afficher un organigramme de l'organisation de vos branches
- Attention : **toujours s'assurer que le dépôt est clean avant de changer de branche**! Sinon, les modifications en cours seront transmises dans la nouvelle branche.

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}

~/artemis $ git branch
* master
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}

~/artemis $ git branch
* master

~/artemis $ git branch -v
* master feefad7 Initial commit
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

L'option « -v » active l'affichage de la révision courante sur cette branche (c'est-à-dire le dernier commit effectué).

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

```
~/artemis $ git branch
* master
```

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

```
~/artemis $ git branch -v
* master feefad7 Initial commit
```

L'option « -v » active l'affichage de la révision courante sur cette branche (c'est-à-dire le dernier commit effectué).

```
~/artemis $ git branch user_accounts
```

Pour créer une branche, il suffit de donner son nom à *branch*.

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

```
~/artemis $ git branch
* master
```

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

```
~/artemis $ git branch -v
* master feefad7 Initial commit
```

L'option « -v » active l'affichage de la révision courante sur cette branche (c'est-à-dire le dernier commit effectué).

```
~/artemis $ git branch user_accounts
```

Pour créer une branche, il suffit de donner son nom à *branch*.

```
~/artemis $ git branch
* master
  user_accounts
```

La nouvelle branche est créée, mais n'est pas encore active. La branche active est indiquée par une étoile.

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

```
~/artemis $ git branch
* master
```

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

```
~/artemis $ git branch -v
* master feefad7 Initial commit
```

L'option « -v » active l'affichage de la révision courante sur cette branche (c'est-à-dire le dernier commit effectué).

```
~/artemis $ git branch user_accounts
```

Pour créer une branche, il suffit de donner son nom à *branch*.

```
~/artemis $ git branch
* master
  user_accounts
```

La nouvelle branche est créée, mais n'est pas encore active. La branche active est indiquée par une étoile.

```
~/artemis $ git checkout user_accounts
Switched to branch 'user_accounts'
```

La commande pour changer de branche s'appelle « checkout ».

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

```
~/artemis $ git branch
* master
```

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

```
~/artemis $ git branch -v
* master feefad7 Initial commit
```

L'option « -v » active l'affichage de la révision courante sur cette branche (c'est-à-dire le dernier commit effectué).

```
~/artemis $ git branch user_accounts
```

Pour créer une branche, il suffit de donner son nom à *branch*.

```
~/artemis $ git branch
* master
  user_accounts
```

La nouvelle branche est créée, mais n'est pas encore active. La branche active est indiquée par une étoile.

```
~/artemis $ git checkout user_accounts
Switched to branch 'user_accounts'
```

La commande pour changer de branche s'appelle « checkout ».

```
~/artemis $ git branch -v
  master        feefad7 Initial commit
* user_accounts feefad7 Initial commit
```

L'étoile montre qu'on a bien changé de branche. On voit que la nouvelle branche part du même commit que la branche mère.

Les branches avec Git

```
~/artemis $ cat artemis.cpp
int main() {
}
```

On part de notre code, auquel on veut ajouter une gestion des utilisateurs.

```
~/artemis $ git branch
* master
```

La commande Git pour la gestion des branches s'appelle « branch ». Par défaut, elle affiche les branches existantes.

```
~/artemis $ git branch -v
* master feefad7 Initial commit
```

L'option « -v » active l'affichage de la révision courante sur cette branche (c'est-à-dire le dernier commit effectué).

```
~/artemis $ git branch user_accounts
```

Pour créer une branche, il suffit de donner son nom à *branch*.

```
~/artemis $ git branch
* master
  user_accounts
```

La nouvelle branche est créée, mais n'est pas encore active. La branche active est indiquée par une étoile.

```
~/artemis $ git checkout user_accounts
Switched to branch 'user_accounts'
```

La commande pour changer de branche s'appelle « checkout ».

```
~/artemis $ git branch -v
  master        feefad7 Initial commit
* user_accounts feefad7 Initial commit
```

L'étoile montre qu'on a bien changé de branche. On voit que la nouvelle branche part du même commit que la branche mère.

```
~/artemis $ vim artemis.cpp
int main() {
}
int handleUserAccounts() { /* code ... */ }
```

On fait les modifications sur cette nouvelle branche…

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts ba25c33] Added user accounts
 1 file changed, 1 insertion(+)
```

On commit ces modifications. Git rappelle qu'on est sur la nouvelle branche *user_accounts*.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts ba25c33] Added user accounts
 1 file changed, 1 insertion(+)
```

On commit ces modifications. Git rappelle qu'on est sur la nouvelle branche *user_accounts*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

Un bug a été trouvé dans la version en production et doit être corrigé. On retourne donc sur la branche *master*, sur laquelle est basée cette version.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts ba25c33] Added user accounts
 1 file changed, 1 insertion(+)
```

On commit ces modifications. Git rappelle qu'on est sur la nouvelle branche *user_accounts*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

Un bug a été trouvé dans la version en production et doit être corrigé. On retourne donc sur la branche *master*, sur laquelle est basée cette version.

```
~/artemis $ vim artemis.cpp
int main() {
    // Correction du bug...
}
```

Le code (pour l'instant non-fonctionnel) de gestion des utilisateurs a bien disparu. On peut corriger le bug, le commiter et publier cette nouvelle version en production.

```
~/artemis $ git commit -a
[master c8d9d77] Fixed urgent bug
 1 file changed, 1 insertion(+)
```

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts ba25c33] Added user accounts
 1 file changed, 1 insertion(+)
```

On commit ces modifications. Git rappelle qu'on est sur la nouvelle branche *user_accounts*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

Un bug a été trouvé dans la version en production et doit être corrigé. On retourne donc sur la branche *master*, sur laquelle est basée cette version.

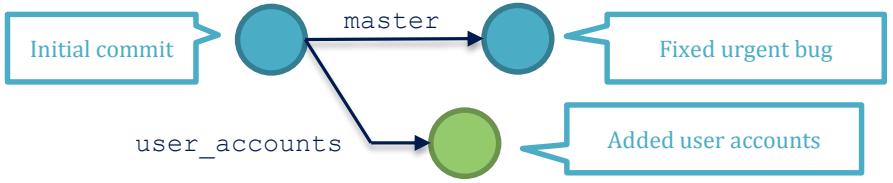
```
~/artemis $ vim artemis.cpp
int main() {
    // Correction du bug...
}
```

Le code (pour l'instant non-fonctionnel) de gestion des utilisateurs a bien disparu. On peut corriger le bug, le commiter et publier cette nouvelle version en production.

```
~/artemis $ git commit -a
[master c8d9d77] Fixed urgent bug
 1 file changed, 1 insertion(+)
```

```
~/artemis $ git branch -v
* master        c8d9d77 Fixed urgent bug
  user_accounts ba25c33 Added user accounts
```

On remarque que les deux branches sont bien sur des révisions différentes et indépendantes :



Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts ba25c33] Added user accounts
 1 file changed, 1 insertion(+)
```

On commit ces modifications. Git rappelle qu'on est sur la nouvelle branche *user_accounts*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

Un bug a été trouvé dans la version en production et doit être corrigé. On retourne donc sur la branche *master*, sur laquelle est basée cette version.

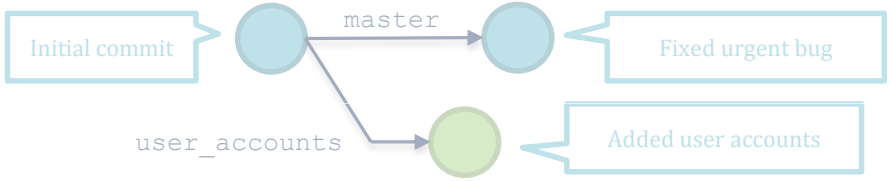
```
~/artemis $ vim artemis.cpp
int main() {
    // Correction du bug...
}
```

Le code (pour l'instant non-fonctionnel) de gestion des utilisateurs a bien disparu. On peut corriger le bug, le commiter et publier cette nouvelle version en production.

```
~/artemis $ git commit -a
[master c8d9d77] Fixed urgent bug
 1 file changed, 1 insertion(+)
```

```
~/artemis $ git branch -v
* master        c8d9d77 Fixed urgent bug
  user_accounts ba25c33 Added user accounts
```

On remarque que les deux branches sont bien sur des révisions différentes et indépendantes :




```
~/artemis $ git checkout user_accounts
Switched to branch 'user_accounts'
```

```
~/artemis $ vim artemis.cpp
int main() {
}
int handleUserAccountsImproved() { /* code ... */ }
```

On repasse sur la branche *user_accounts* pour continuer le code de gestion des utilisateurs. La correction du bug n'existe pas sur cette branche.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts b304efc] Improved user accounts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On commit ces améliorations.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts b304efc] Improved user accounts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On commit ces améliorations.

Après de nombreux tests, la nouvelle fonctionnalité de compte utilisateur est jugée stable et peut maintenant être publiée en production. Il faut donc merger *user_accounts* dans *master*.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts b304efc] Improved user accounts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On commit ces améliorations.

Après de nombreux tests, la nouvelle fonctionnalité de compte utilisateur est jugée stable et peut maintenant être publiée en production. Il faut donc merger *user_accounts* dans *master*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

C'est la branche *master* qui va être modifiée, il faut donc d'abord retourner dessus.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts b304efc] Improved user accounts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On commit ces améliorations.

Après de nombreux tests, la nouvelle fonctionnalité de compte utilisateur est jugée stable et peut maintenant être publiée en production. Il faut donc merger *user_accounts* dans *master*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

C'est la branche *master* qui va être modifiée, il faut donc d'abord retourner dessus.

```
~/artemis $ git merge user_accounts
Auto-merging artemis.cpp
Merge made by the 'recursive' strategy
 artemis.cpp | 1 +
 1 file changed, 1 insertion(+)
```

La commande *merge* est simple à utiliser : on indique simplement la branche qui contient les modifications voulues. Dans la mesure du possible, Git s'occupe d'incorporer automatiquement les modifications.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts b304efc] Improved user accounts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On commit ces améliorations.

Après de nombreux tests, la nouvelle fonctionnalité de compte utilisateur est jugée stable et peut maintenant être publiée en production. Il faut donc merger *user_accounts* dans *master*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

C'est la branche *master* qui va être modifiée, il faut donc d'abord retourner dessus.

```
~/artemis $ git merge user_accounts
Auto-merging artemis.cpp
Merge made by the 'recursive' strategy
 artemis.cpp | 1 +
 1 file changed, 1 insertion(+)
```

La commande *merge* est simple à utiliser : on indique simplement la branche qui contient les modifications voulues. Dans la mesure du possible, Git s'occupe d'incorporer automatiquement les modifications.

```
~/artemis $ cat artemis.cpp
int main() {
    // Correction du bug...
}
int handleUserAccountsImproved() { /* code ... */ }
```

On remarque que le code contient maintenant bien les modifications des deux branches : la correction du bug et la gestion des utilisateurs.

Les branches avec Git

```
~/artemis $ git commit -a
[user_accounts b304efc] Improved user accounts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

On commit ces améliorations.

Après de nombreux tests, la nouvelle fonctionnalité de compte utilisateur est jugée stable et peut maintenant être publiée en production. Il faut donc merger *user_accounts* dans *master*.

```
~/artemis $ git checkout master
Switched to branch 'master'
```

C'est la branche *master* qui va être modifiée, il faut donc d'abord retourner dessus.

```
~/artemis $ git merge user_accounts
Auto-merging artemis.cpp
Merge made by the 'recursive' strategy
 artemis.cpp | 1 +
 1 file changed, 1 insertion(+)
```

La commande *merge* est simple à utiliser : on indique simplement la branche qui contient les modifications voulues. Dans la mesure du possible, Git s'occupe d'incorporer automatiquement les modifications.

```
~/artemis $ cat artemis.cpp
int main() {
    // Correction du bug...
}
int handleUserAccountsImproved() { /* code ... */ }
```

On remarque que le code contient maintenant bien les modifications des deux branches : la correction du bug et la gestion des utilisateurs.

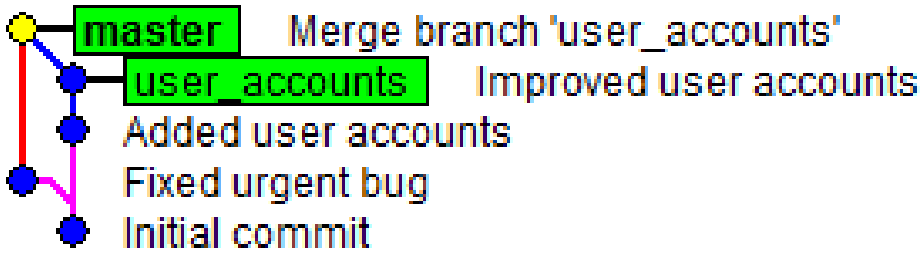
```
~/artemis $ git branch -d user_accounts
Deleted branch user_accounts (was b304efc)
```

Si aucune autre modification n'est prévue prochainement sur cette branche, on peut la supprimer.

Les branches avec Git

L'utilitaire *gitk* est pratique pour afficher le log de votre dépôt de manière plus graphique, en particulier pour les branches.

Dans l'exemple précédent, cela donne l'arbre suivant :



Points particuliers

Trucs et astuces pour gagner du temps

Créer un dépot local

- Même lorsque vous travaillez seul sur un projet, utiliser un VCS est une bonne idée pour :
  - conserver l'historique des modifications du code et ainsi faciliter son évolutivité
  - utiliser des branches
  - faire des backups (en ajoutant un serveur et en utilisant *push*)
- Pour cela, faites simplement un *init* sur votre dossier de code, suivi des *commit* habituels. Pas besoin de *clone*.

Créer un dépot local

Cependant, utiliser un **dépôt distant** sur un serveur même lorsqu'on est le seul développeur a plusieurs **avantages** :

- **Ce dépôt sert de backup** : même si vous perdez votre ordinateur, votre code est encore sur le serveur. Un simple *push* met la backup à jour.
- Si vous avez besoin d'aide ou que votre projet prend de l'ampleur, il devient très simple de partager votre code.

Utiliser git stash

- Il est parfois nécessaire de rapidement mettre de côté les modifications en cours par rapport au dernier commit pour revenir à un dépôt clean.
  - par exemple, pour corriger un bug sur *master* rapidement, sans devoir faire un commit incomplet sur la branche de développement

Utiliser git stash

- Pour cela, Git dispose de la commande *stash*, qui enregistre dans un autre endroit les modifications en cours et replace le code dans son état au dernier commit.
- Ensuite, il suffit de faire *stash apply* pour récupérer les modifications.

Utiliser git stash

```
~/example $ git checkout –b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

Utiliser git stash

```
~/example $ git checkout -b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

```
~/example $ vim example.cpp
int main() {
    // Une modif...
}
```

On fait une modification qu'on ne souhaite pas commiter pour le moment, mais on doit retourner sur *master* pour corriger un bug…

Utiliser git stash

```
~/example $ git checkout -b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

```
~/example $ vim example.cpp
int main() {
    // Une modif...
}
```

On fait une modification qu'on ne souhaite pas commiter pour le moment, mais on doit retourner sur *master* pour corriger un bug…

```
~/example $ git stash
Saved working directory and index state WIP on test
```

On fait donc un *stash*

Utiliser git stash

```
~/example $ git checkout -b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

```
~/example $ vim example.cpp
int main() {
    // Une modif...
}
```

On fait une modification qu'on ne souhaite pas commiter pour le moment, mais on doit retourner sur *master* pour corriger un bug…

```
~/example $ git stash
Saved working directory and index state WIP on test
```

On fait donc un *stash*

```
~/example $ cat example.cpp
int main() {
}
```

Les changements ont bien disparu, le dépôt est clean

Utiliser git stash

```
~/example $ git checkout -b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

```
~/example $ vim example.cpp
int main() {
    // Une modif...
}
```

On fait une modification qu'on ne souhaite pas commiter pour le moment, mais on doit retourner sur *master* pour corriger un bug…

```
~/example $ git stash
Saved working directory and index state WIP on test
```

On fait donc un *stash*

```
~/example $ cat example.cpp
int main() {
}
```

Les changements ont bien disparu, le dépôt est clean

```
~/example $ git checkout master
Switched to branch 'master'
```

On peut maintenant revenir sur la branche master et corriger le bug

Utiliser git stash

```
~/example $ git checkout -b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

```
~/example $ vim example.cpp
int main() {
    // Une modif...
}
```

On fait une modification qu'on ne souhaite pas commiter pour le moment, mais on doit retourner sur *master* pour corriger un bug…

```
~/example $ git stash
Saved working directory and index state WIP on test
```

On fait donc un *stash*

```
~/example $ cat example.cpp
int main() {
}
```

Les changements ont bien disparu, le dépôt est clean

```
~/example $ git checkout master
Switched to branch 'master'
```

On peut maintenant revenir sur la branche master et corriger le bug

```
~/example $ git checkout test
```

L'opération effectuée, on revient sur test

Utiliser git stash

```
~/example $ git checkout -b test
```

« git checkout –b test » est un raccourcis pour « git branch test » / « git checkout test »

```
~/example $ vim example.cpp
int main() {
    // Une modif...
}
```

On fait une modification qu'on ne souhaite pas commiter pour le moment, mais on doit retourner sur *master* pour corriger un bug…

```
~/example $ git stash
Saved working directory and index state WIP on test
```

On fait donc un *stash*

```
~/example $ cat example.cpp
int main() {
}
```

Les changements ont bien disparu, le dépôt est clean

```
~/example $ git checkout master
Switched to branch 'master'
```

On peut maintenant revenir sur la branche master et corriger le bug

```
~/example $ git checkout test
```

L'opération effectuée, on revient sur test

```
~/example $ git stash apply

~/example $ cat example.cpp
int main() {
    // Une modif...
}
```

On utilise *stash apply* pour retrouver nos modifications initiales

Le fichier .gitignore

- Certains fichiers ne doivent **jamais** être ajoutés dans le dépôt :
  - fichiers de compilations (propres à votre système)
  - fichiers binaires (compilés)
  - fichiers de configuration propres à votre IDE
  - …
- Dans le cas de Git, pour que *status* et *commit* n'en tiennent pas compte, il faut créer un fichier nommé « .gitignore »* à la racine du dépôt, et indiquer ce qui doit être ignoré

* ne pas oublier le point au début du nom du fichier

Le fichier .gitignore

- Chaque ligne contient un élément à ignorer.
- Exemple de fichier .gitignore :

```
*.o
build/
artemis
artemis.sublime-project
artemis.sublime-workspace
.gitignore
```

Tous les fichiers temporaires de compilation

Un répertoire quelconque

Le binaire compilé du projet

Les fichiers de configuration de votre IDE

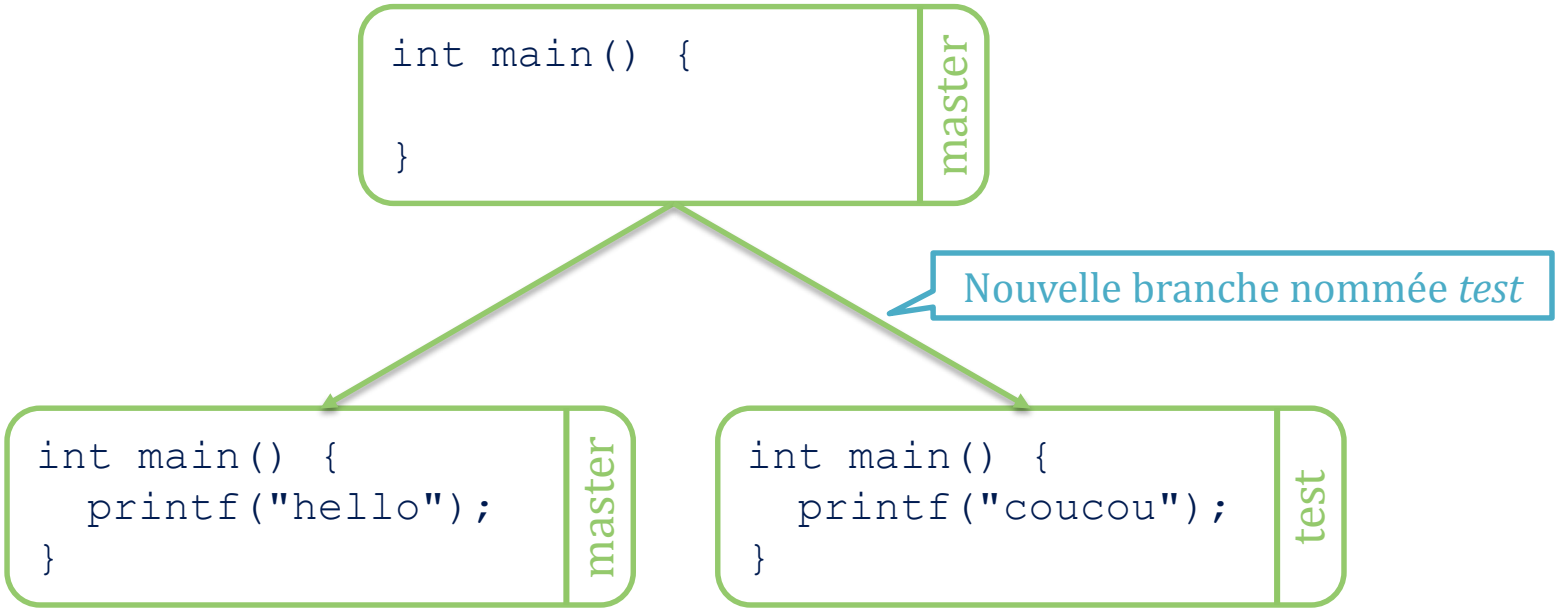
Ce fichier .gitignore dont le contenu vous est propre

Résoudre un conflit

- Lorsqu'un conflit apparait pendant un merge (par exemple, suite à un *pull*), le VCS va :
  - Suspendre l'opération en cours
  - Modifier le code pour faire apparaître le conflit
  - Se bloquer tant que le conflit n'est pas résolu
- Le développeur doit corriger manuellement tous les fichiers concernés par le conflit, puis faire un commit pour valider sa résolution

Résoudre un conflit avec Git

On considère les deux branches suivantes :



Tenter de merger *test* dans *master* va provoquer un conflit, car la même ligne de code a été modifiée dans les deux branches.

Résoudre un conflit avec Git

```
~/example $ git merge test
Auto-merging example.cpp
CONFLICT (content): Merge conflict in example.cpp
Automatic merge failed; fix conflicts and then commit the result
```

Comme prévu, le merge échoue

Résoudre un conflit avec Git

```
~/example $ git merge test
Auto-merging example.cpp
CONFLICT (content): Merge conflict in example.cpp
Automatic merge failed; fix conflicts and then commit the result
```

Comme prévu, le merge échoue

```
~/example $ cat example.cpp
int main() {
<<<<<<< HEAD
    printf("coucou");
=======
    printf("hello");
>>>>>>> test
}
```

Git a ajouté des marqueurs dans le code pour aider à la résolution du conflit. Les deux versions sont affichées l'une après l'autre, séparées par « ======= ».
HEAD désigne la version actuelle (c'est-à-dire *master*), et *test* est la branche que l'on essaye de merger.

Résoudre un conflit avec Git

```
~/example $ git merge test
Auto-merging example.cpp
CONFLICT (content): Merge conflict in example.cpp
Automatic merge failed; fix conflicts and then commit the result
```

Comme prévu, le merge échoue.

```
~/example $ cat example.cpp
int main() {
<<<<<<< HEAD
    printf("coucou");
=======
    printf("hello");
>>>>>>> test
}
```

Git a ajouté des marqueurs dans le code pour aider à la résolution du conflit. Les deux versions sont affichées l'une après l'autre, séparées par « ======= ».
HEAD désigne la version actuelle (c'est-à-dire *master*), et *test* est la branche que l'on essaye de merger.

```
~/example $ vim example.cpp
int main() {
    printf("coucou/hello");
}
```

On réécrit une version cohérente du code et on supprime les marqueurs

Résoudre un conflit avec Git

```
~/example $ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:   example.cpp

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

status donne de précieuses informations sur la situation

Résoudre un conflit avec Git

```
~/example $ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:   example.cpp

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

status donne de précieuses informations sur la situation

```
~/example $ git add example.cpp
```

On marque le fichier comme résolu

Résoudre un conflit avec Git

```
~/example $ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:   example.cpp

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

status donne de précieuses informations sur la situation

```
~/example $ git add example.cpp
```

On marque le fichier comme résolu

```
~/example $ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be commited:
        modified:   example.cpp
```

A nouveau, *status* nous indique la marche à suivre

Résoudre un conflit avec Git

```
~/example $ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:   example.cpp

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

status donne de précieuses informations sur la situation

```
~/example $ git add example.cpp
```

On marque le fichier comme résolu

```
~/example $ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be commited:
        modified:   example.cpp
```

A nouveau, *status* nous indique la marche à suivre

```
~/example $ git commit
[master 91bca8f] Merge branch 'test'
```

Il ne reste plus qu'à commiter pour terminer la résolution du conflit et le merge

Merci de votre attention !

Source : http://xkcd.com/1597/