Laboratoire GEN

GIT

HES-SO – Chr. Greppin/Eric Lefrançois/Miguel Santamaria

Février 2018

1 Introduction

A l'occasion de votre collaboration dans le cadre de votre projet de groupe, vous allez certainement être confronté à certains problèmes:

- Qui a modifié le fichier X ? Il ne fonctionne plus, il comporte désormais des bugs !
- Bob, peux-tu m'aider à travailler sur le fichier Y pendant que je travaille sur le fichier X ?
- Qui a modifié ce fichier ? Ces changements sont inutiles !
- Qui a ajouté ces nouveaux fichiers au projet ? A quoi servent-ils ?
- Qui a résolu le bug alpha ? Et comment ?
- Attention à ne pas modifier le fichier X car je risque d'écraser tes modifications !

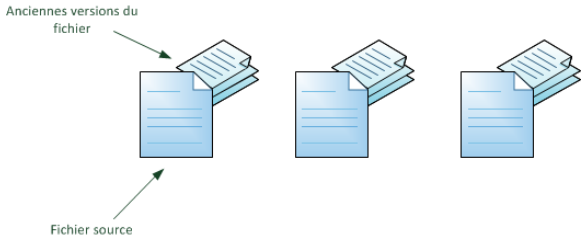
Pour éviter ces problèmes, ou pour y répondre.., vous aurez besoin d'utiliser un logiciel de gestion de versions (SCM pour Source Code Management). Un tel outil est indispensable lorsque l'on travaille simultanément sur un même projet et donc sur les mêmes fichiers. Même s'il vous arrive de travailler seul sur un projet, un logiciel de gestion de versions peut vous offrir de nombreux avantages comme la conservation de l'historique de vos modifications.

Il existe plusieurs systèmes de gestion de versions : CVS (Current Versions System), SVN (Subversion), Mercurial, etc.

Dans le cadre de ce laboratoire, nous utiliserons **GIT** qui est un logiciel open source gratuit.

1.1 Qu'est qu'un SCM ?

Les logiciels de gestion suivent l'évolution des fichiers sources et gardent les anciennes versions ou modifications de chacun d'eux.



Mais ils n'offrent pas uniquement cette fonctionnalité, sinon il ne s'agirait que d'outils de backup. Ils proposent entre autres plusieurs fonctionnalités très utiles pour le développement de vos projets :

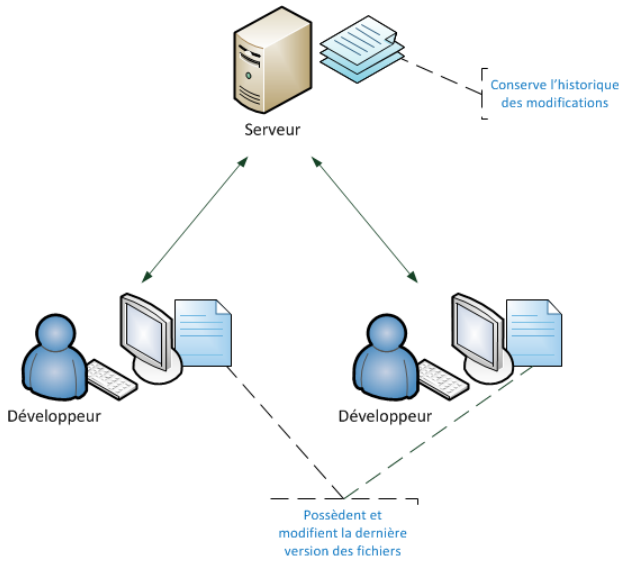
- Si deux personnes travaillent sur le même fichier simultanément, les SCM sont capables d'assembler les modifications et d'éviter que le travail d'une des deux personnes ne soit écrasé.
- Les SCM gardent toutes les modifications se rattachant à chacun des fichiers en enregistrant le motif de la modification. On est donc capable de retrouver qui a écrit chaque ligne de code de chaque fichier et pourquoi.

Les SCM sont donc principalement utilisés pour **suivre l'évolution du code source** afin de retenir les modifications et de revenir en arrière si besoin, et de **travailler à plusieurs** sans risquer de perdre des changements effectués sur des mêmes fichiers.

1.2 DEUX TYPES DE SCM

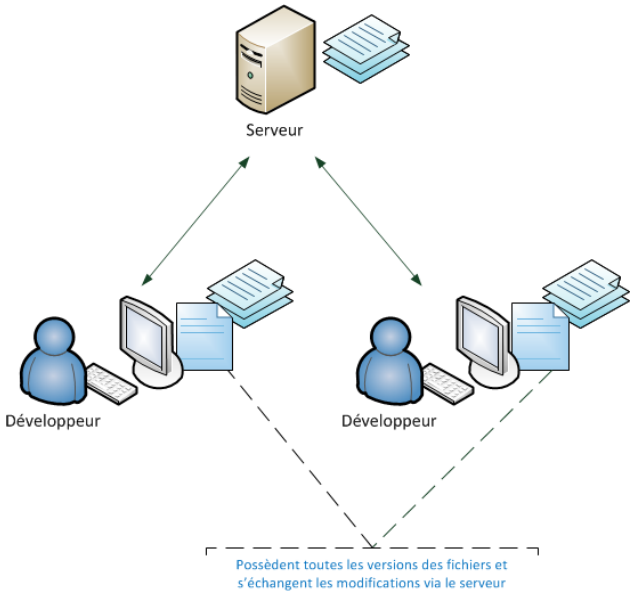
Il existe deux types principaux de logiciels de gestion de versions :

1. Les logiciels **centralisés**, dotés d'un serveur qui conserve les anciennes versions de fichiers et auquel on doit se connecter pour charger et/ou envoyer les modifications effectuées.



SCM centralisé

2. Les logiciels **distribués** fonctionnant sans serveur, où chaque personne travaillant sur le projet possède en local l'historique de l'évolution de chacun des fichiers. La plupart des SCM distribués utilisent néanmoins un serveur comme **point de rencontre**. Si tel est le cas, le serveur dispose également d'un historique des modifications afin de permettre l'échange des informations entre les personnes collaborant sur le projet.



SCM Distribué

⚠ Git fait partie de la deuxième catégorie de logiciel, à savoir les systèmes de gestion de version non centralisés.

1.3 GIT ET LES AUTRES

Dans ce laboratoire, vous allez apprendre à manipuler Git, mais vous devez savoir qu'il existe d'autres logiciels libres du même type que vous pouvez utiliser pour votre projet de groupe. Chacun possédant ses forces et ses faiblesses.

Outil	Description	Type
SVN (Subversion)	Probablement l'outil le plus connu et le plus utilisé. Simple d'utilisation et bien intégré à Windows à l'aide du programme possédant une interface graphique : Tortoise SVN	Centralisé
CVS	Un des plus anciens SCM. Malgré le fait qu'il soit encore utilisé, il est préférable d'utiliser SVN qui corrige certains défauts comme notamment son incapacité à suivre des fichiers renommés.	Centralisé
Mercurial	Plus récent, comparable à Git	Distribué
Bazaar	Un outil complet et récent qui se focalise sur sa facilité d'utilisation et flexibilité	Distribué
Git	Le SCM que nous allons utiliser. Son atout majeur est sa rapidité et la gestion des branches.	Distribué

En résumé, avant de choisir votre SCM, retenez que :

- CVS est le plus ancien ; il est recommandé de ne plus l'utiliser car moins puissant et plus vraiment mis à jour.
- SVN est le plus connu et le plus utilisé mais de nombreux projets choisissent de passer à des outils plus récents.
- Les trois autres, Mercurial, Bazaar et Git se valent. Récents, puissants et distribués, chacun a ses avantages et défauts.

1.4 FORCES DE GIT ?

Les caractéristiques essentielles de Git..

Distribué

Git, comme tout SCM distribué, permet de cloner l'intégralité du dépôt.

Cela signifie, même si GIT s'appuie sur un serveur central, que chaque utilisateur possède en local une sauvegarde complète.

Rapidité

Git est rapide, voire très rapide. L'origine de cette vitesse vient du fait que les opérations se font localement et que Git a été conçu à l'origine pour soutenir l'élaboration du code source du noyau Linux. Il devait donc gérer

efficacement un grand nombre de dépôts dès le premier jour. Une autre raison est que Git est écrit en C et que, dès le départ, les développeurs à l'origine de Git ont été très soucieux de sa rapidité.

Léger

La grande force de Git est également sa taille extrêmement réduite. En général, un répertoire Git sera à peine plus lourd qu'un checkout SVN alors que vous avez accès à toutes les modifications et logs inaccessibles avec SVN.

GitHub

Bon nombre de développeurs choisissent Git en raison de l'existence de GitHub. GitHub est en quelque sorte un réseau social pour code source. Sur le site, vous pouvez trouver d'autres développeurs, des projets qui sont similaires aux vôtres et auxquels vous pouvez facilement contribuer.

Dans le cadre de ce laboratoire, nous n'utiliserons pas GitHub pour héberger nos projets tests, mais **BitBucket**. En effet, pour des raisons historiques, BitBucket a offert dès le départ la gratuité aux milieux académiques.

Toutefois, depuis quelque temps déjà, GitHub permet, via l'école, d'avoir pendant deux années un repository gratuit et non public. Un programme éducatif.

Donc, libre à vous d'utiliser ou non GitHub pour votre projet de groupe.

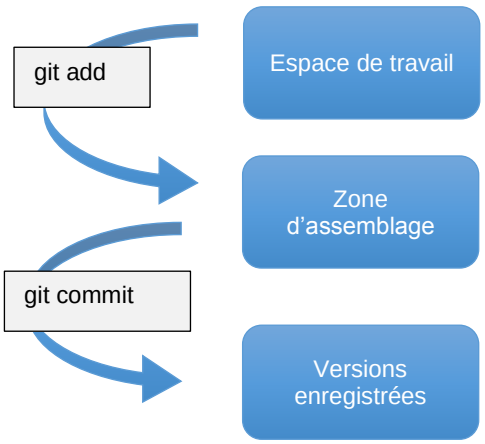
Staging area – Zone d'assemblage

Supposons que votre applicatif soit géré par GIT.

Son code source sera rassemblé dans un dossier que l'on appellera par la suite « **Espace de travail** ».

Cet espace de travail contiendra un fichier caché, le fichier `.git`, que l'on appelle le **dépôt local** («**Local repository**»). Ce dépôt local qui contiendra entre autres deux éléments essentiels :

- Une **zone d'assemblage** («**Staging area**»)
- La liste des versions enregistrées



Cette configuration fonctionne de la manière suivante :

- Dès qu'une modification est apportée à un fichier de votre « **Espace de travail** », et que la nouvelle version de ce fichier mérite selon vous d'être sauvegardée en l'état, opérez un « **`git add`** » du fichier en question.
- Un **snapshot** du fichier est alors opéré (**photo** de son état actuel) puis enregistré dans la **zone d'assemblage**.
- Quand vous le désirez, maintenant ou plus tard, vous pouvez ordonner une sauvegarde (`commit`) de la zone d'assemblage.

  Vous créez ainsi une **nouvelle version** de votre logiciel qui porte un numéro : le numéro du commit.

   *Le nouveau commit enregistré dans le dépôt local contient directement – sous forme compressée - le **snapshot** des fichiers des fichiers qui ont été commités.*

  Avec GIT, contrairement à un SVN classique, une nouvelle version n'est pas décrite par une liste de modifications à apporter à la version précédente mais tout simplement par un snapshot. Cette manière d'opérer est très efficace!

La sauvegarde dans le dépôt local

Comme dit précédemment, cette sauvegarde est opérée au moyen d'une commande « **`git commit`** ». Cette commande n'a donc pas besoin d'avoir à lister tous les fichiers que l'on désire sauvegarder, elle prend automatiquement le contenu de la zone d'assemblage.

 *Notez que la sauvegarde opérée sur le dépôt **ne contient pas l'état actuel** de tel ou tel fichier, mais l'état qui le caractérisait au moment de son « assemblage » (`git add`). Cela vous permet de sauvegarder l'état intermédiaire d'un fichier que vous êtes en train de modifier, et de continuer à le modifer...*

Texte descriptif

Au moment du commit – *qui a pour effet de créer une nouvelle version* - , vous avez la possibilité de saisir un texte descriptif.

Une séquence typique :

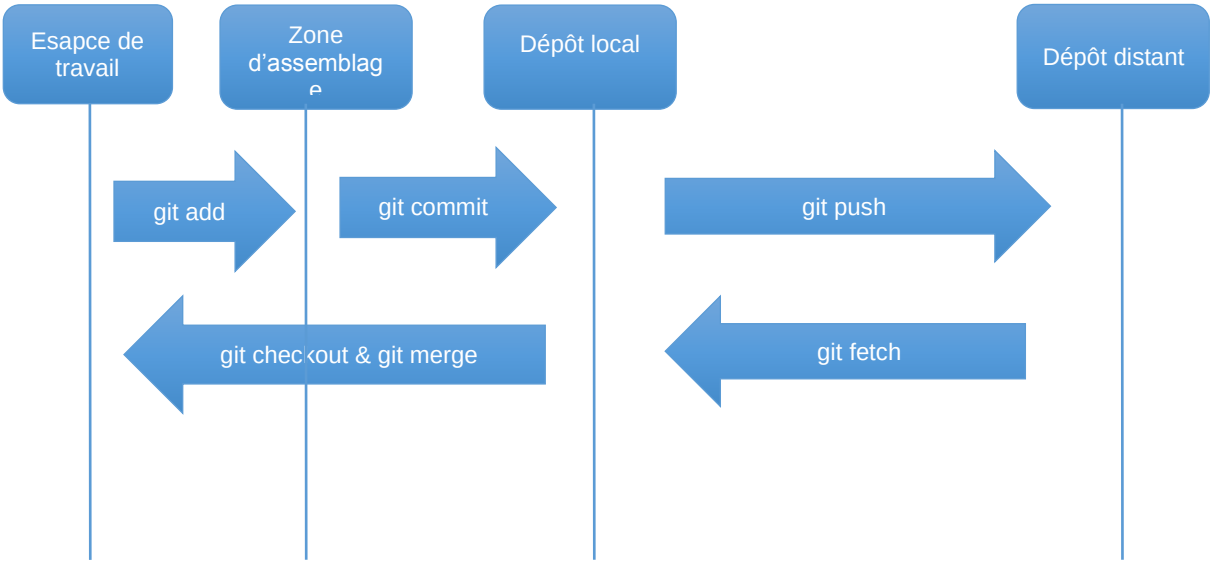
```
$git add *                    Pour ajouter tous les fichiers dans la zone d'assemblage
ou :
$git add nomDesFichiers

$git commit -m "Blabla..(raison de la modification)"
« commiter », avec l'option « -m » permettant de spécifier directement un texte enregistrant la raison du commit.
```

Tout en local

En dehors des 3 commandes **`fetch`**, **`pull`** et **`push`**, prévues pour communiquer avec le serveur distant, la plupart des commandes mises à disposition ne communiquent qu'avec votre disque dur, en local.

Hormis le fait que chaque opération soit effectuée de manière plus rapide que d'habitude, l'avantage d'un tel système est qu'il vous permet de travailler quand vous n'êtes pas connecté à internet. Cela peut paraitre anodin, mais pouvoir travailler à tout moment à 100% est un énorme avantage.

Vous pouvez récupérer toutes les données actuelles depuis le serveur distant avec un **`fetch`**, contrôler les dernières versions mises à disposition, puis opérer les fusions que vous désirez, en gérant les conflits éventuels.



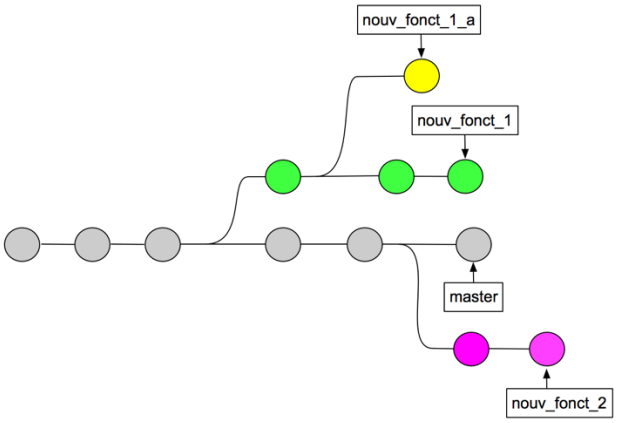
Gestion des branches

Le principal atout de Git est son **modèle de gestion des branches !** 

Git travaille avec une branche de départ, la **branche dite « Master »**, créée par défaut, puis vous permet de gérer de multiples branches locales qui peuvent être totalement indépendantes les unes des autres et dont la création, la fusion et la suppression ne prennent que quelques secondes.

La figure ci-dessous présente l'arbre des branches.

- La branche `master` avec 6 commits en l'état.
- Une branche `nouv_fonct_1` a été créée à partir du 3ème commit du `master` pour développer une nouvelle grosse fonctionnalité. Ce développement se déroule en parallèle avec l'évolution de la branche master. Cette branche possède déjà 3 commits. Le jour où cette nouvelle fonctionnalité sera intégrée au master (opération merge), des conflits potentiels devront être gérés.
- Une branche `nouv_fonct_1_a` a été créée pour développer – *à titre d'essai* - une sous fonctionnalité participant à la fonctionnalité `nouv_fonct_1`. Cette branche présente en l'état un seul commit. Si l'essai est concluant, cette branche sera intégrée à la branche `nouv_fonct_1`, en présentant éventuellement des conflits.
- Une branche `nouv_fonct_2`, qui a été créée pour développer une deuxième grosse fonctionnalité. Cette branche possède déjà 2 commits.



Pour quelles raisons créer de nouvelles branches ?

- Avoir une première branche contenant le code en production et une autre pour faire des tests que vous fusionnerez plus tard.
- Créer une branche pour toutes les nouvelles fonctionnalités que vous créez, vous permettant ainsi d'aller et revenir facilement entres-elles et effacer la branche une fois la fonctionnalité incluse dans la branche principale.
- Créer une branche pour vos expérimentations, et, si vous réalisez que cela ne fonctionne pas, il vous suffira de la supprimer pour abandonner le travail, sans que personne ne s'en soit aperçu.
- Créer une branche pour essayer une nouvelle idée, committer quelques fois, revenir à l'endroit où vous avez créé cette branche, appliquer un « patch » (voir plus loin la notion de « patch »), retourner à l'endroit où vous expérimentez, le fusionner en local avec votre branche principale…

L'important est de réaliser que lorsque vous poussez (push) les modifications sur un dépôt distant, vous n'êtes pas obligé de pousser toutes vos branches. Vous pouvez vous limiter au partage de quelques branches seulement. Cela permet aux développeurs d'expérimenter des nouvelles idées sans se soucier du quand et du comment les partager avec les autres.

Git a le mérite de rendre très simple tout ce processus et peut changer la manière de travailler d'un développeur.

2 Le travail en local

Vous trouverez ci-dessous la liste des fonctions essentielles de GIT, à saisir dans une fenêtre de commande.

En préliminaire..

Installez GIT : voir l'annexe « Installation de Git » à la fin du document

Créer le dossier de travail

Créez un dossier `EssaiGit` (votre dossier de travail « Working Directory »)

Contenant le fichier `Hello.java`

```
public class Hello {
	public static void main(String … args) {
		System.out.println ("Coucou");
	}
}
```

Créer un dépôt local (« Local repository »)

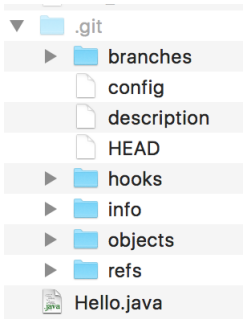
Le dépôt local (« local repository ») sera créé dans le dossier de travail

```
$ cd votre/path/EssaiGit
```

Dans ce dossier, initialisez alors un repository local Git avec la commande :

```
$ git init
```

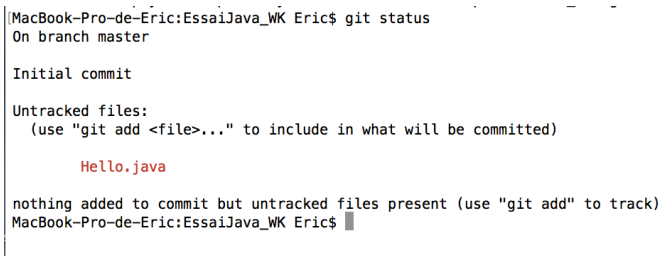
Cela a pour effet de créer un dossier `.git` dans votre dossier de travail.



Examiner l'état de votre dépôt local : `git status`

```
$ git status
```

La commande `git status` affiche la liste des fichiers que vous avez modifiés depuis que la dernière version a été sauvegardée.



L'image ci-dessus indique que le fichier `Hello.java` a été créé sans avoir encore été pris en compte dans le dépôt local de GIT. Ce fichier est « **Untracked** ».

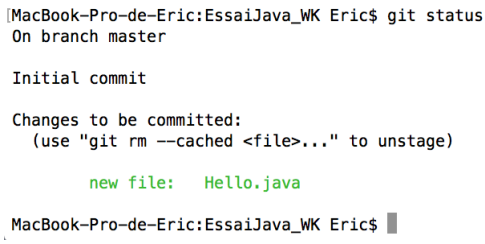
La zone d'assemblage est vide, non mentionnée...

Ajouter un fichier à la zone d'assemblage: `git add`

Pour ajouter le fichier Hello.java, utiliser la commande `git add` :

```
$ git add Hello.java
```

Un nouveau `git status` indique alors que le fichier `Hello.java` sera sauvegardé au prochain `commit`.



Ce qui a été ajouté dans la zone d'assemblage est un **snapshot** de l''état de votre fichier.

Pour mettre en évidence ce **snapshot**,, avant de passer à la suite, modifiez maintenant le texte « `Coucou` » par « `Hello` »

Créer votre première version : `commit`

Commiter avec l'option « **-m** » qui vous permet de spécifier un texte descriptif.

```
$ git commit -m "Premier commit"
```

```
MacBook-Pro-de-Eric:Essai Eric$ git commit -m "Premier commit"
[master (root-commit) dd3ab8e] Premier commit
 1 file changed, 1 insertion(+)
 create mode 100644 Hello.java
MacBook-Pro-de-Eric:Essai Eric$
```

Votre première version a été enregistrée dans le dépôt local !

Contrôlez l'historique des versions : `git log`

Opérez un `git log` pour obtenir l'historique actuel de vos différentes versions :

```
$ git log
```

Vous obtenez :

```
MacBook-Pro-de-Eric:Essai Eric$ git log
commit dd3ab8e6d85be2753291e963f17109a1b91e5f0f
Author: ELS <eric.lefrancois@hispeed.ch>
Date:   Tue Feb 7 10:36:04 2017 +0100

    Premier commit
MacBook-Pro-de-Eric:Essai Eric$
```

Vous signalant que vous avez opérer un commit en date du 7 Février et que ce dernier porte le numéro :

dd3ab8e6d85be2753291e963f17109a1b91e5f0f

Ce numéro correspond à un numéro de version, une version vers laquelle il sera toujours possible de revenir dans le futur !

Contrôler l'état actuel des modifications : `git status`

Un nouveau `git status` vous signale:

```
MacBook-Pro-de-Eric:Essai Eric$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Hello.java

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-de-Eric:Essai Eric$
```

Le dernier commit a effacé le contenu de la zone d'assemblage.

Or.. souvenez-vous, le commit que vous avez effectué correspondait à la version

`System.out.println("Coucou");` ⇨ Snapshot enregistré dans le `git add`

Etat actuel de votre fichier : `System.out.println("Hello");`

Ainsi, le `git status` vous signale que le fichier `Hello.java` a été modifié depuis le dernier commit.

Il vous signale par ailleurs que cette modification n'a pas été enregistrée dans la zone d'assemblage.

 Un commit a pour effet d'effacer la zone d'assemblage

Revenir à la dernière version du fichier : `git checkout nom_fichier`

 Nous allons maintenant revenir à la version de `Hello.java` telle qu'elle avait été sauvegardée lors du dernier commit, celle qui affiche le texte « `Coucou` » :

```
$ git checkout Hello.java
```

En observant votre éditeur, vous remarquerez alors que vous êtes revenu à l'état que vous aviez sauvegardé.

Magique ! 

 Un git status pour contrôler..

```
$ git status
```

```
MacBook-Pro-de-Eric:Essai Eric$ git status
On branch master
nothing to commit, working tree clean
MacBook-Pro-de-Eric:Essai Eric$
```

Tout est dans l'ordre.. Votre dernière sauvegarde correspond à l'état actuel de votre espace de travail 

 Un raccourci pour commiter l'ensemble du dossier de travail

Dans la majorité des cas, le développeur souhaite commiter tout un ensemble de modifications opérées sur de nombreux fichiers. Un `git add` fichier par fichier serait deviendrait alors très laborieux..

 Pour placer tout le contenu d'un espace de travail dans la zone d'assemblage, utiliser la commande `git add -A`, l'option **`-A`** signifiant «`all`».

Puis commiter avec l'option « **`-m`** » qui vous permet de spécifier un texte descriptif.

```
$ git add -A
$ git commit -m "Premier commit"
```

Un raccourci pour les deux commandes ci-dessus :

```
$ git commit -a -m "Premier commit"
```

Attention toutefois : Ce raccourci ne prend pas en compte les **nouveaux fichiers qui ont été créés entre temps**. Il est nécessaire de « tracker » ces derniers (de « suivre » ces derniers) au moyen d'un `git add` préalable, ou alors d'utiliser la commande `git commit -a -m` .. avec une option **`-i`** supplémentaire qui vous permet d'ajouter des nouveaux fichiers, dont il faut spécifier le nom.

Continuer l'investigation : Modifier le code…

Modifiez votre code..

Le fichier `Hello.java` devient :

```
public class Hello {
	public static void main(String[] args) {
		System.out.println (new Message().getText());
	}
}
```

Un nouveau fichier `Message.java` contient le code suivant :

```
class Message {
	public String getText() {
		return "Re-Coucou";
	}
}
```

Un `git status` vous affiche dorénavant:

```
MacBook-Pro-de-Eric:EssaiJava_WK Eric$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

	modified:   Hello.java

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

	Message.java

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-de-Eric:EssaiJava_WK Eric$
```

Le fichier `Message.java` a été créé, le fichier `Hello.java` a été modifié.

L'affichage du `git status` vous signale :

1/ Que les modifications apportées dans `Hello.java` n'ont pas été enregistrées dans la zone d'assemblage !

2/ Que le nouveau fichier `Message.java` est untracked : pas encore pris en compte par GIT

Vous pouvez également voir les changements que vous avez effectués dans votre dossier de travail par rapport à la dernière version enregistrée sur votre dépôt local à l'aide de la commande `git diff`:

```
$ git diff
```

```
MacBook-Pro-de-Eric:EssaiJava_WK Eric$ git diff
diff --git a/Hello.java b/Hello.java
index 8dc6516..87a8f38 100644
--- a/Hello.java
+++ b/Hello.java
@@ -1,5 +1,5 @@
 public class Hello {
-       public static void main(String[] args) {
-              System.out.println ("Coucou");
-       }
+    public static void main(String[] args) {
+        System.out.println (new Message().getText());
+    }
 }
MacBook-Pro-de-Eric:EssaiJava_WK Eric$
```

L'image ci-dessus indique qu'une ligne du fichier `Hello.java` a été remplacée par une nouvelle.

Vous pouvez voir les changements opérés sur un fichier spécifique en indiquant son chemin d'accès :

```
 $ git diff path_fichier
```

Enregistrez votre deuxième version, avec un enregistrement préalable dans la zone d'assemblage

```
$ git add -A
```

Vous pouvez contrôler que le nouveau fichier Message.java est maintenant pris en compte par GIT :

```
$ git status
```

```
MacBook-Pro-de-Eric:Essai Eric$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Hello.java
        new file:   Message.java

MacBook-Pro-de-Eric:Essai Eric$
```

Opérer le commit

```
$ git commit -m "Deuxième commit"
```

Modifiez encore une fois le fichier `Hello.java` :

```
public class Hello {
       public static void main(String[] args) {
              Message m = new Message();
              System.out.println (m.getText());
       }
}
```

Un `git status` vous affichera maintenant :

```
MacBook-Pro-de-Eric:EssaiJava_WK Eric$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Hello.java

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-de-Eric:EssaiJava_WK Eric$
```

Si vous opérez un commit en l'état, il n'aura aucun effet ! En effet, la zone d'assemblage n'a pas été mise à jour.

Faites-le cependant!!

```
$ git commit -m "Troisième commit"
```

Vous pouvez constater avec un `git diff` que rien a changé.

Revenir à la dernière version du fichier: `git checkout nomFichier`

Vous voulez maintenant revenir à la version de `Hello.java` telle qu'elle avait été sauvegardée lors du dernier commit. Ce faisant, vous annulez votre dernière modification (qui n'était qu'un essai juste pour essayer..).

```
$ git checkout Hello.java
```

En éditant Hello.java, vous pouvez contrôler que vous êtes revenu à l'état que vous aviez sauvegardé.

Pour revenir à l'ancienne version (dernier commit) de tous vos fichiers

```
$ git checkout *.*
```

Afficher la liste des commits effectués en l'état: `git log`

Pour afficher le log des commit effectués :

```
$ git log
```

Vous pouvez utiliser l'option **–p** pour afficher le détail des logs.

```
MacBook-Pro-de-Eric:Essai Eric$ git log
commit 9eac8446fb4e17343aaa0ac27969e86f61803264
Author: ELS <eric.lefrancois@hispeed.ch>
Date:   Tue Feb 7 12:20:06 2017 +0100

    Deuxième commit

commit dd3ab8e6d85be2753291e963f17109a1b91e5f0f
Author: ELS <eric.lefrancois@hispeed.ch>
Date:   Tue Feb 7 10:36:04 2017 +0100

    Premier commit
MacBook-Pro-de-Eric:Essai Eric$
```

Modifier le texte descriptif d'un commit : `-amend`

Vous pouvez très facilement modifier votre dernier message de commit à l'aide de la commande suivante :

```
$ git commit --amend -m "C'est vraiment les deuxième commit"
```

Opérer un « clear » de la zone d'assemblage : `git reset HEAD`

Pour remettre de l'ordre dans votre zone d'assemblage…

```
$ git reset HEAD
```

Expérimentez

- Etat préliminaire : Vous venez de faire un commit
- Modifier `Hello.java` : Ajout d'un nouveau commentaire
- `git add Hello.java` : Un snapshot de `Hello.java` est enregistré dans la zone d'assemblage
- `git status` pour contrôler
- `git reset HEAD` : Vidage de la zone d'assemblage
- `git status` pour contrôler

Annuler un commit : `git reset HEAD^`

Pour annuler le dernier commit, utiliser la commande :

```
$ git reset HEAD^
```

*Cette opération annule les sauvegardes effectuées au niveau du **dépôt local**, mais cela n'affecte en rien l'état actuel des fichiers de votre **dossier de travail**!!*

Pour indiquer à quel commit vous souhaitez revenir, la notation suivante est utilisée :

- `HEAD` : dernier commit

  Aucun commit n'est supprimé !

  ⇨ Cela a simplement pour effet d'opérer un clear de la zone d'assemblage, comme nous l'avons vu précédemment.

- `HEAD^` : avant-dernier commit
  - La zone d'assemblage actuelle est effacée.
  - Le dernier commit est alors supprimé.

  ⇨ On revient ainsi à l'avant-dernier commit

- `HEAD^^` : avant-avant-dernier commit
  - La zone d'assemblage actuelle est effacée.
  - Les deux derniers commits sont supprimés.

  ⇨ On revient ainsi à l'avant-avant-dernier commit

- `HEAD~2` : avant-avant dernier commit (notation équivalente)

- **`36830fa21`** : indique un numéro de commit précis (seul les premiers caractères sont suffisant tant que l'identifiant utilisé du commit est unique)
  - La zone d'assemblage actuelle est effacée.
  - Tous les commits effectués jusqu'au **`36830fa21`** (mais non comris) sont supprimés.

  ⇨ On revient ainsi au commit **`36830fa21`**

Au niveau du dernier commit, si vous désirez uniquement éliminer la sauvegarde d'un fichier en particulier, utilisez la commande suivante :

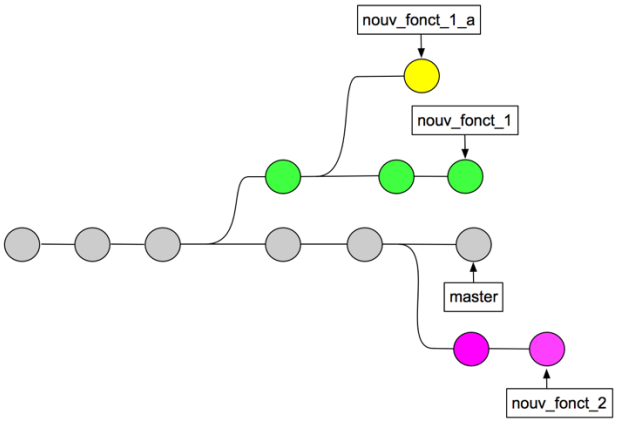
```
$ git reset HEAD^ -- PathFichier
```

Ignorer des fichiers ou des répertoires

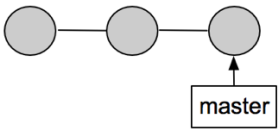
Vous pouvez demander à git d'ignorer certains fichiers, – *pratique quand on travaille avec l'option* **`–A`** -, en créant un fichier nommé **`.gitignore`** dans la racine de votre répertoire de travail. Ce fichier contiendra ce type d'information:

```
# Les ligne commençant par '#' sont des commentaires.
# Ignorer tous les fichiers nommés foo.txt
foo.txt
# Ignorer tous les fichiers du répertoire bin
bin
# Ignorer tous les fichiers html
*.html
# à l'exception de foo.html
!foo.html
```

2.1 LES BRANCHES



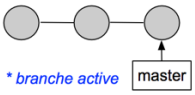
De base, votre projet ne comporte qu'une seule branche, la branche **master**.



Pour afficher la branche sur laquelle on travaille (branche « active » ou branche « courante ») : `git branch`.

```
$ git branch
```

→ * master

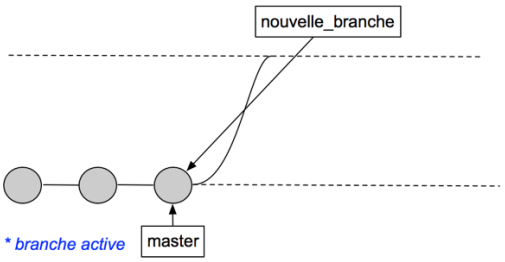


Il se peut, lors du développement de votre projet, que vous soyez amené à effectuer une modification importante de votre code, entrainant de multiples changements. Le mieux est alors de créer une nouvelle branche secondaire, sur laquelle vous pourrez continuer à utiliser toutes les fonctionnalités de Git. En cas de succès, le résultat de cette nouvelle branche devra être fusionné avec la branche principale. En cas d'échec, le retour en arrière sera possible.

Il est bien entendu prévu de pouvoir travailler sur plusieurs branches en parallèle (notamment s'il s'agit d'un travail de groupe).

Pour créer une nouvelle branche

```
$ git branch nouvelle_branche
```

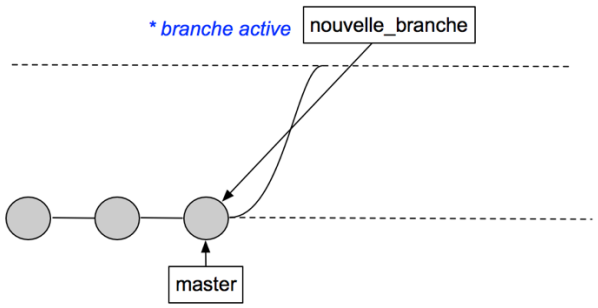


Pour l'instant, vous avez simplement créé la branche, mais elle n'est pas encore active ! Vous pouvez le vérifier en tapant à nouveau la commande `git branch`.

Pour « aller » sur une branche

```
$ git checkout nouvelle_branche
```

Pour basculer sur `nouvelle_branche` qui devient la branche courante.



Affichez l'historique des commits de `nouvelle_branche`

```
$ git log
```

→ **Vous pourrez constater que cette nouvelle branche s'appuie sur tous les commits de la branche master dont elle est issue** !

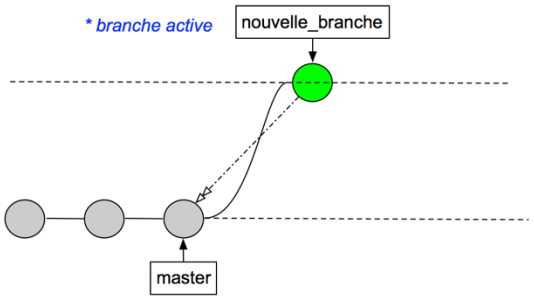

Vous pouvez directement créer et aller sur la branche en une seule commande :

```
$ git checkout -b nouvelle_branche
```

Un premier commit sur la nouvelle branche

Faites une modification dans une des fichiers et commitez cette modification (→ 1er commit sur la nouvelle branche)

```
$ git commit -a -m "Nouvelle_branche: commit 1"
```



Affichez l'historique des commits de `nouvelle_branche`

```
$ git log
```

→ **Vous pourrez constater que la nouvelle branche s'appuie sur son premier commit (en vert sur la figure), qui lui-même s'appuie sur tous les commits de la branche master dont la nouvelle branche est issue** !

Pour fusionner

Lorsque le travail sur la branche secondaire est terminé et que vous êtes satisfait de votre travail, il vous faut fusionner le contenu de la branche `nouvelle_branche` avec la branche `master`. Pour ce faire, activez la branche principale puis intégrez ensuite le travail que vous avez fait.

```
$ git checkout master
$ git merge nouvelle_branche
```

Supprimer une branche secondaire

```
git branch -d nom_branche
```

Pour effacer une branche.

Mais ne fonctionne que si la branche a été déjà fusionnée !

```
git branch -D nom_branche
```

Pour effacer **une branche en force**, même si la branche n'a pas encore été fusionnée

Expérimentez

1. Créer une branche secondaire
2. Vous placer dans cette branche secondaire
3. Modifier le fichier `Hello.java` (ajout d'un commentaire)
4. Commiter dans la branche secondaire

Puis, en laissant le fichier `Hello.java` ouvert (mais sans y toucher afin de ne pas créer de conflit) pour visualiser les changements

5. Revenir à la branche principale, consulter `Hello.java` et contrôler que le commentaire n'existe plus

Vous pouvez visualiser l'état des différentes branches avec la commande :

```
git branch -v
```

6. Revenir à la branche secondaire, consulter `Hello.java` et contrôler que le commentaire est réapparu
7. Fusionner la branche secondaire
8. Supprimer la branche secondaire

Visualisez l'état des différentes branches

```
git branch -v
```

2.2 MERGE & GESTION DES CONFLITS

GIT est très fort pour ce qui est de la fusion.

Toutefois, si **deux mêmes parties du même fichier** ont été modifiées simultanément, un conflit sera signalé au moment de la fusion.

Expérimentation : Modifier parallèlement le fichier `Hello.java` sur les branches master et `nouvelle_branche`

1. Opérer un commit de la branche `master`
2. Revenir sur la branche `nouvelle_branche` et modifier le fichier `Hello.java` en rajoutant un commentaire
3. Commiter le changement sur la branche `nouvelle_branche`
4. Revenir sur la branche `master` et modifier le fichier `Hello.java` en rajoutant un commentaire
5. Commiter le changement sur la branche `master`
6. En restant sur la branche `master`, demandez à fusionner la branche `nouvelle_branche`

```
Automatic merge failed; fix conflicts and then commit the result.
```

Cela signifie en clair que GIT n'a pas créé de commit ! 

Le processus a été « mis en pause » jusqu'à ce que le conflit soit résolu.

Pour voir quels sont les fichiers mis en cause, faites un **`git status`** :

```
MacBook-Pro-de-Eric:EssaiJava Eric$ 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:   Hello.java

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-de-Eric:EssaiJava Eric$
```

Tout ce qui n'a pas été résolu est affiché. Notamment, on peut voir dans l'image ci-dessus que le fichier `Hello.java` comporte un conflit.

Git conserve des markers dans tous les fichiers qui comportent des conflits.

 Ouvrez le fichier Hello.java

Il suffit alors de résoudre tranquillement la chose en éditant chacun des fichiers présentant un problème.. 

Après avoir résolu les conflits :

1. Exécuter la commande **`git add`** pour chacun des fichiers qui se sont trouvés en conflit et que vous avez modifiés.
2. Terminer par un commit.

 Il est possible d'utiliser un outil graphique qui peut aider à résoudre les conflits, l'outil `mergetool`, mais qui doit au préalable avoir être configuré. Consulter le cas échéant la documentation GIT.

```
$ git mergetool
```

2.3 LES « PATCHS »

Un **patch** est un fichier texte qui rassemble des changements opérés par rapport à un code source.

Un patch est créé avec la commande `git format-patch`. Le patch incluera non seulement la différence entre les fichiers, mais aussi toutes les méta-informations relatives aux changements : la date des commit, les messages correspondants et le nom du « commiter ».

Par exemple

```
git format-patch master --stdout > nom_fichier.patch
```

Cela va avoir pour effet de créer un fichier "`nom_fichier`" qui contiendra tous les changements opérés sur la **branche actuelle** par rapport à la branche master.

Le fichier patch peut alors être envoyé à une personne qui l'utilisera pour appliquer le patch à son dépôt local, la méta-information sera alors préservée.

Pour appliquer le patch, par exemple à la branche master sur laquelle on est retourné :

```
git apply nom_fichier.patch
```

Une alternative au patch consiste à utiliser la commande `git diff`. Toutefois, le fichier `diff` généré ne contient pas la méta-information.

2.4 RESUME DES FONCTIONS GIT

Terminologie « Untracked »

Untracked

Liste des fichiers situés dans l'espace de travail qui ne sont pas encore prise en compte par GIT.

Terminologie « HEAD»

HEAD

HEAD est un pointeur qui référence le commit à partir duquel se rapportent les modifications que l'on opère dans l'espace de travail.

Normalement, ce pointeur désigne le commit le plus récent qui a été effectué sur la branche courante.

Si on change de branche, HEAD est mis à jour automatiquement de manière à référencer le dernier commit de la nouvelle branche courante.

Pointeurs sur les anciens commits

- `HEAD^` : avant-dernier commit
- `HEAD^^` : avant-avant-dernier commit
- `HEAD~2` : avant-avant dernier commit (notation équivalente)
- **`36830fa21`** : indique un numéro de commit précis (seul les premiers caractères sont suffisant tant que l'identifiant utilisé du commit est unique)

Terminologie « DETACHED HEAD»

Si on se retrouve dans une situation « détachée » (« HEAD detached from commit xyz), cela signifie que le pointeur HEAD pointe sur un commit qui n'est pas le commit le plus récent de la branche courante.

C'est par exemple le cas si on effectue la manœuvre qui consiste à revenir sur une ancienne version.
Voir *Annexe : Revenir à une ancienne version* en page *48.*

Si on se trouve « par erreur » dans cette situation, cela peut être très dangereux car tout ce que l'on modifie dans l'espace de travail risque d'être perdu le jour on l'on revient sur le commit le plus récent.

Pour revenir à la situation normale, il suffit de demander à revenir sur la branche avec un `git checkout` classique.

Par exemple, pour rattacher le HEAD alors que l'on se trouve sur la branche master :

```
git checkout master
```

Démarrer

```
$ git init
```

Créer un dépôt local dans le répertoire courant.

Etat des modifications & analyse

```
$ git status
```

Obtenir l'état des modifications entre le dossier de travail et le dépôt local (état au dernier commit) .

```
git diff
```

Lister ligne par ligne les différences entre les fichiers du dossier de travail et l'état des fichiers dans le dépôt local (état au dernier commit).

```
git diff path_fichier
```

Idem, mais en limitant le `diff` à l'analyse d'un seul fichier.

```
git diff <commit> <commit> path_fichier
```

Idem, en limitant le `diff` à l'analyse d'un seul fichier et en comparant la différence entre deux commits.

Exemple : `git diff HEAD^ HEAD path_fichier` Différence entre maintenant (`HEAD`) et dernier commit (`HEAD^`)

Sauvegarde en local

```
$ git add fichier1 fichier2 fichier3
..
$ git commit -m "Blabla"
```

Sauvegarde en deux temps.

1. Spécifier la liste des fichiers ou des zones de fichiers à sauvegarder en faisant des ajouts (`add`) dans la zone d'assemblage (staging area)
2. Sauvegarde de la zone d'assemblage (commit)

Pour placer tout le contenu d'un espace de travail dans la zone d'assemblage, utiliser la commande `git add -A`, l'option `-A` signifiant «`all`».

```
$ git add -A
$ git commit -m "Blabla"
```

Il est possible d'opérer des sauvegardes en court-circuitant la zone d'assemblage..

```
$ git commit -a -m "Blabla"
```

Pour sauvegarder tous les fichiers qui étaient listés lors du `git status` dans les colonnes « `Changes to be commited` » et « `Changed but not updated` »

 Attention toutefois : Ce raccourci ne prend pas en compte les **nouveaux fichiers qui ont été créés entre temps**. Il est nécessaire de « tracker » ces derniers au moyen d'un `git add` préalable, ou alors d'utiliser la commande `git commit -a -m` .. avec une option **`-i`** supplémentaire qui vous permet d'ajouter des nouveaux fichiers, dont il faut spécifier le nom.

```
$ git commit fichier1 fichier2 fichier3
```

➪ Sauvegarder en indiquant la liste précise des fichiers à sauvegarder

Gestion des commits (annulation, historique, etc..)

```
$git log
```

➪ Pour obtenir l'historique des commits.

Quelques options intéressantes :

```
$git log -10
```

➪ Historique des commits en limitant l'affichage aux 10 derniers commits

```
$git log --stat
```

➪ Historique des commits avec liste des fichiers modifiés pour chacun des commits et en affichant pour chaque fichier le nombre de lignes modifiées

```
$git commit --amend
```

➪ Pour modifier le dernier message commit.

```
$ git checkout Hello.java
```

➪ Pour restaurer la dernière version enregistrée de `Hello.java`

```
$git reset HEAD^
```

➪ Pour annuler un commit

➪ Si vous désirez indiquer à quel commit vous souhaitez revenir, la notation suivante est utilisée :

- `HEAD` : dernier commit

  Aucun commit n'est supprimé !

  ⇨ Cela a simplement pour effet d'opérer un clear de la zone d'assemblage, comme nous l'avons vu précédemment.

- `HEAD^` : avant-dernier commit
  - La zone d'assemblage actuelle est effacée.
  - Le dernier commit est alors supprimé.

  ⇨ On revient ainsi à l'avant-dernier commit

- `HEAD^^` : avant-avant-dernier commit
  - La zone d'assemblage actuelle est effacée.
  - Les deux derniers commits sont supprimés.

  ⇨ On revient ainsi à l'avant-avant-dernier commit

- `HEAD~2` : avant-avant dernier commit (notation équivalente)
- **`36830fa21`** : indique un numéro de commit précis (seul les premiers caractères sont suffisant tant que l'identifiant utilisé du commit est unique)
    - La zone d'assemblage actuelle est effacée.
    - Tous les commits effectués jusqu'au **`36830fa21`** (mais non comris) sont supprimés.

    ⇨ On revient ainsi au commit **`36830fa21`**

Au niveau du dernier commit, si vous désirez uniquement éliminer la sauvegarde d'un fichier en particulier, utilisez la commande suivante :

```
$ git reset HEAD^ -- PathFichier
```

```
$git reset HEAD -- Hello.java
```

Pour retirer un fichier de la sauvegarde.

Ignorer des fichiers ou des répertoires

Vous pouvez demander à git d'ignorer certains fichiers, – *pratique quand on travaille avec l'option* **–A** -, en créant un fichier nommé **`.gitignore`** dans la racine de votre répertoire de travail. Ce fichier contiendra ce type d'information:

```
# Les ligne commençant par '#' sont des commentaires.
# Ignorer tous les fichiers nommés foo.txt
foo.txt
# Ignorer tous les fichiers du répertoire bin
bin
# Ignorer tous les fichiers html
*.html
# à l'exception de foo.html
!foo.html
```

Patchs

Un **patch** est un fichier texte qui rassemble des changements opérés par rapport à un code source.

Un patch est créé avec la commande `git format-patch`. Le patch incluera non seulement la différence entre les fichiers, mais aussi toutes les méta-informations relatives aux changements : la date des commit, les messages correspondants et le nom du « commiter ».

Par exemple

```
git format-patch master --stdout > nom_fichier.patch
```

Cela va avoir pour effet de créer un fichier "`nom_fichier`" qui contiendra tous les changements opérés sur la branche actuelle par rapport à la branche master.

Manipulation des branches

```
$ git branch
```

Lister les branches, la branche active est marquée par une étoile (*)

```
$git branch nouvelle_branche
```

Pour créer une nouvelle branche

```
$git checkout une_branche
```

Pour changer la branche active en restaurant dans votre espace de travail l'état de la branche en question

```
$git checkout -b branch nouvelle_branche
```

Pour créer une nouvelle branche qui devient en même temps la nouvelle branche active

```
git branch -d nom_branche
```

Pour effacer une branche. Ne fonctionne que si la branche a été déjà fusionnée

```
git branch -D nom_branche
```

Pour effacer une branche. Fonctionne en force, même si la branche n'a pas encore été fusionnée

```
git branch -m ancien_nom nouveau_nom
```

Pour renommer une branche

Fusionner des branches

```
$ git checkout master
$ git merge nouvelle_branche
```

Pour fusionner le résultat obtenu de la `nouvelle_branche` avec la branche master :

1. Revenir sur la branche master
2. Puis lui fusionner le contenu de `nouvelle_branche`

Fusionner des fichiers spécifiques

Dans l'exemple ci-dessous, on désire fusionner un nouveau fichier qui a été dans la branche `une_branche`

```
$ git branch
* master    # Branche active
  une_branche
$ git checkout une_branche chemin_du_fichier1 chemin_du_fichier2
       # Le checkout place les deux fichiers dans la zone d'assemblage, à contrôler avec un git status
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   chemin_du_fichier1      # Il s'agit d'un nouveau fichier
#   modified:   chemin_du_fichier2      # Ce fichier a été modifié
#
```

```
$ git commit -m "Merge partiel de 2 fichiers"  # sans l'option -a !!!
[master]: created 4d3e37b: " Merge partiel de 2 fichiers"
1 file changed, 72 insertions(+), 0 deletions(-)
```

- Deux remarques :
  - Il est possible d'utiliser des Wildcards. Comme par exemple :
    `git checkout une_branche *`
  - Ou encore de signaler tout un dossier. Comme par exemple :
    `git checkout branche chemin/vers/dossier`
    Tous les chemins sont relatifs. Si vous ne vous trouvez pas à la racine de votre dossier, il est nécessaire de signaler les chemins relatifs à vos fichiers.

```
git branch --merged
```

- Pour visualiser les branches secondaires déjà fusionnées

```
git branch --no-merged
```

- Pour visualiser les branches secondaires non encore fusionnées

Comparer deux branches

```
git diff une_branche..une_autre_branche
```

- Pour comparer deux branches en terme de modifications textuelles

```
git diff une_branche votre_branche
```

- Pour comparer `votre_branche` avec une branche. Cette commande ne montrera que les modifications qui ont été effectuées dans `votre_branche` depuis qu'elle s'est désolidarisée de `une_branche`.

```
git log une_branche votre_branche
```

- Pour comparer deux branches en terme de commits effectués

```
git diff --name-status une_branche..une_autre_branche
```

- Pour afficher la liste des fichiers qui présentent des différences

Divers

```
$ git branch -v
```

- Pour visualiser les dernières validations sur chaque branche

Merge & Gestion des conflits

Voir le paragraphe correspondant : *Merge & Gestion des conflits* en p. 22.

Supprimer un fichier

La façon la plus propre de supprimer un fichier consiste à utiliser la commande `git rm`.

Si on supprime un fichier en faisant un simple `delete` dans l'espace de travail, ce dernier apparait à l'occasion d'un `git status` dans les fichiers « `Changed but not staged` », à savoir « Modifié mais non placé dans la zone d'assemblage », ce qui peut sembler un peu lourd.

```
$ git rm path_fichier
```

Le fichier est alors supprimé de l'espace de travail. Si on opère un `git status`, il apparaît simplement en tant que fichier « `deleted` ». Le fichier n'apparaitra plus à l'occasion du prochain commit.

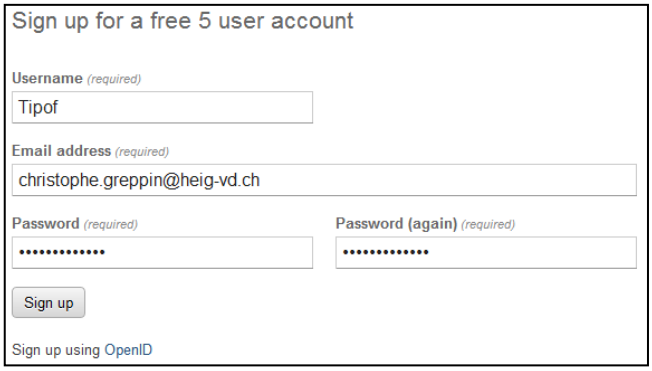
3 Synchroniser avec un serveur distant

3.1 CREATION DU COMPTE

Dans le cadre de ce laboratoire, nous allons utiliser un service gratuit nous mettant à disposition un serveur centralisé fonctionnant avec GIT afin de stocker nos sources et les partager facilement avec d'autres membres du projet. Il existe plusieurs services gratuits mais dans le cadre de ce laboratoire nous allons utiliser : **https://bitbucket.org/**

Ce dernier a l'avantage de fournir des comptes illimités (sans limite d'utilisateurs) gratuitement pour les développeurs d'un milieu académique.

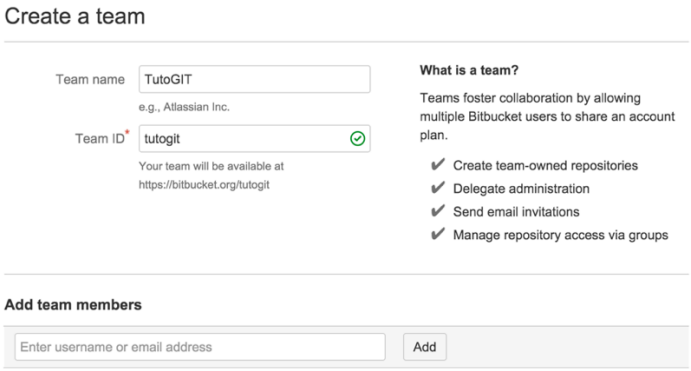
En tout premier, créer un compte gratuit (limité à 5 utilisateurs) sur le site mentionné ci-dessus en mettant votre adresse e-mail de l'école comme dans l'exemple ci-dessous. Attention, le « look » du site peut avoir changé entre temps !!! Mais le principe reste le même.



BitBucket vous demande de compléter un formulaire où vous serez amené notamment à indiquer le site WEB de l'école : http://www.heig-vd.ch

Créer le team en ajoutant les emails de tous les membres du team

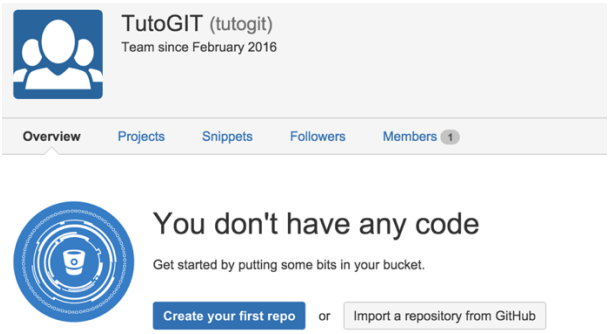
```
Menu Teams > Create a team
```



3.2 CREATION D'UN DEPOT DISTANT

Cette opération dépend bien entendu du système utilisé.

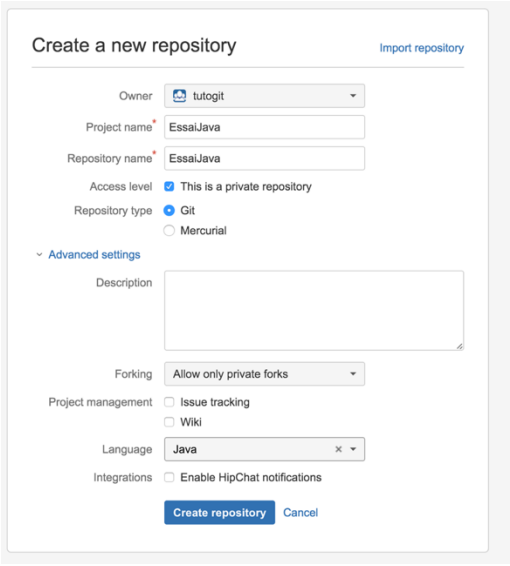
Avec BitBucket, dès qu'un nouveau team est créé, vient aussitôt une fenêtre d'invitation à créer votre premier dépôt distant associé à votre team :



Créer le dépôt distant

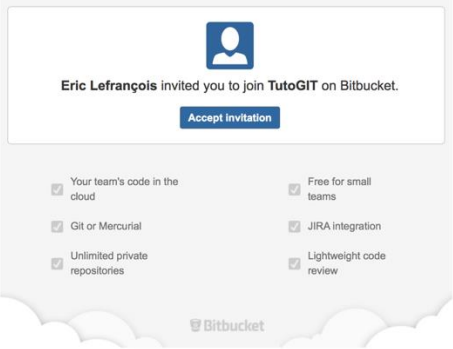
- Un **team** peut contenir plusieurs **projets**.
- Un **projet** peut contenir plusieurs **dépôts distants**.

→ Un dépôt distant est toujours créé au sein d'un projet spécifique.



Par email, les autres membres du team reçoivent une invitation à devenir membre du team.

Il sera toujours temps de refuser..

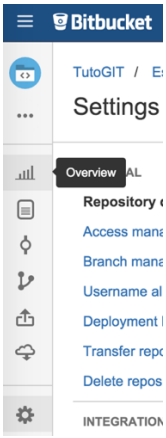


BitBucket vous invite alors à cloner l'état actuel de votre dépôt local sur le dépôt distant en vous indiquant 4 commandes à saisir :

```
$ cd /path/to/my/repogit          # Vous placer dans votre espace de travail
$ git remote add origin https://ericlef@bitbucket.org/tutogit/essaijava.git
$ git push -u origin --all        # Push up du dépôt local
$ git push -u origin --tags       # Push up de tous les marqueurs
```

EXPLICATIONS : Voir *Clonage d'origine d'un dépôt local sur un nouveau dépôt distant en page 38*

Sur bitbucket, dans le menu Overview du dépôt distant, vous pouvez alors consulter l'état du dépôt : les branches, le source, etc..



3.3 PARTAGER LE CODE AU SEIN D'UN TEAM

Sont présentés dans ce paragraphe uniquement les principes clé du partage de code. La mise en œuvre est explicitée dans le chapitre suivant : *Résumé des commandes GIT relatives au serveur distant* en page 36

Démarrer un projet en partage

On supposera que « l'administrateur » possède un projet initial, déposé dans un dépôt local personnel. L'objectif consiste à partager le développement de ce projet entre les différents membres de son team qui possèdent tous un droit en écriture.

Le mode opérationnel consiste en les étapes suivantes :

- L'administrateur du team crée un nouveau dépôt distant.

  Voir *Création d'un dépôt distant* en p. 32

- L'administrateur clone l'état actuel de son dépôt local sur le dépôt distant.

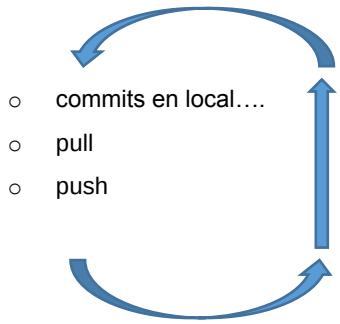
  Voir *Clonage d'origine d'un dépôt local sur un nouveau dépôt distant* en p. 38

- Les membres du team créent chacun de leur côté un nouveau dépôt local qui leur est personnel, vide de tout code, puis opèrent un clone du dépôt distant dans leur dépôt local
- Voir *$ git push -u origin --tags # Push up de* tous les marqueurs

  Upload de tous les marqueurs sur le serveur distant **origin**.
  Notamment, cette opération enregistre l'historique de vos commits sur le dépôt distant.

  Clonage d'origine du dépôt distant sur un nouveau dépôt local en p. 39

- Et le travaille peut commencer..



Voir le *Résumé des commandes GIT relatives au serveur distant* en page 36, en commençant pas la **terminologie**.

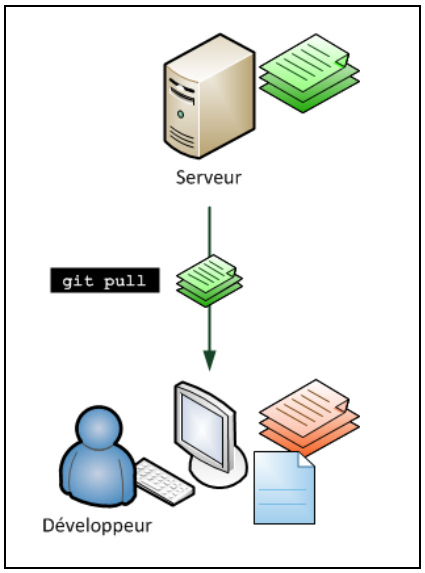
Fusionner les modifications apportées par les autres membres du groupe

Deux manières de procéder sont mises à disposition pour télécharger les dernières modifications apportées par les autres sur le serveur distant.

Première méthode : `pull`

La première méthode est en général effectuée si on travaille sur une branche spécifique ou encore si on n'a pas modifié son code depuis le dernier téléchargement. On utilise alors la commande `git pull`.

```
$git pull
```



La commande git pull opère un **download** des données de la branche du dépôt distant correspondant à la branche courante.

Les changements effectués par les autres sont alors **fusionnés** aux vôtres automatiquement.

Si deux personnes modifient en même temps deux endroits distincts d'un même fichier, les changements sont intelligemment fusionnés par Git.

S'il arrive que la même zone de code ait été modifiée en même temps, Git vous avertit d'un conflit en indiquant le nom des fichiers concernés.

Deuxième méthode : `fetch & merge`

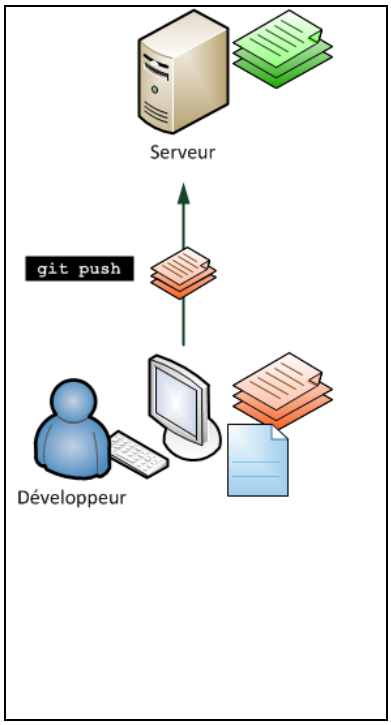
La seconde méthode consiste à opérer un `git fetch` puis un `get merge`.

- La commande `git fetch` ne fait que rapatrier toutes les données du dépôt distant dans leur état actuel.
- La commande `git merge` vous permettra de fusionner les données téléchargées avec la destination de votre choix, comme par exemple la branche master : `git merge origin/master`.

Cette méthode est souvent conseillée car elle opère en deux étapes, ce qui permet d'effectuer certains contrôles avant d'effectuer le merge.

Partager nos modifications avec les autres membres du groupe

Pour envoyer le résultat de vos différents commits au dépôt distant, il vous suffit d'utiliser la commande `git push`.



La commande push opère un **upload** qui **écrase les données** existantes du dépôt distant.

⚠ Comme le serveur Git du dépôt distant ne peut régler les conflits, personne ne doit avoir opéré de push depuis votre dernier pull !

Le mode opérationnel consiste ainsi à opérer successivement les deux commandes:

1. `pull`
2. `push`

Il est recommandé de faire régulièrement des commits (donc travailler de manière locale) et d'effectuer des pushs quand cela est nécessaire uniquement.

⚠ Il faut faire attention au fait que vous pouvez annuler les commits que vous avez effectuer au niveau local. Une opération push, par contre, est **irréversible** !!

3.4 RESUME DES COMMANDES GIT RELATIVES AU SERVEUR DISTANT

3.4.1 PRINCIPE ESSENTIELS

Terminologie

⚠ La terminologie est un avant-propos dont la lecture est absolument nécessaire pour comprendre le fonctionnement des différentes commandes mises à disposition.

Origin server

Le dépôt distant à partir duquel a été opéré le premier clone

Remote branches (branches distantes)

- Le dépôt local contient des références sur les branches distantes avec lesquelles il y a eu un échange
- Syntaxiquement, ces références prennent la forme suivante : `nom_dépôt_distant/nom_branche`, Par exemple, la référence `origin/master` désigne la branche `master` du dépôt distant « `origin` ».

Ajustement des références

Pour voir la liste des références aux branches distantes

```
$ git branch -vv
```

```
MacBook-Pro-de-Eric:essaijava Eric$ git branch -vv
* master    1fa2f71 [origin/master] Pise en compte
  secondary dbd96a1 [origin/secondary] Commit secon
```

On peut voir :

- que la dernière synchronisation de la branche master se **réfère** au commit `1fa2f71` du dépôt distant
- que la dernière synchronisation de la branche secondaire se **réfère** au commit `dbd96a1` du dépôt distant

⚠ Les références aux branches distantes (`1fa2f71`, `dbd96a1`) ne sont pas ajustables par vous-mêmes !

- Elles sont ajustées automatiquement à chaque fois qu'une communication aura eu lieu avec le dépôt distant (`git pull`, `push`, `fetch`).
- En quelque sorte, il s'agit de signets qui vous permettent de vous rappeler la version dans laquelle se trouvaient les branches du dépôt distant à l'issue de votre dernière synchronisation.

Par exemple, si votre dernier clone a porté sur la version `v1234` de la branche master du dépôt distant, votre marqueur `origin/master` pointera sur `v1234`, alors qu'entre temps votre branche master locale a peut-être évolué et correspond à une nouvelle version locale autre que `v1234`.

Connaître l'état du dépôt distant

```
$git fetch origin     → Récupérer les informations actuelles
$git log origin/nom_branche
```

Comparer la branche locale avec la branche distante.

Objectif : Savoir si notre dépôt local est en retard (un `git pull` à opérer ?)

```
$git fetch origin     → Récupérer les informations actuelles
$git log nom_branche..origin/nom_branche
```

```
MacBook-Pro-de-Eric:essaijava Eric$ git log master..origin/master
commit f513d7aadbe4c3eeb8ba8c698101c50a3f2248b4
Author: ErnestDujardin <eric.lefrancois@hispeed.ch>
Date:   Sat Feb 6 16:29:55 2016 +0100

    Commit A5
```

On peut voir dans cet exemple que le dépôt distant a reçu un commit, qu'il se trouve à la version f513d7 suite au commit « Commit A5 » d'un autre développeur.


Si notre dépôt local n'a pas de retard, le `git log` n'affiche rien

Comparer la branche distante avec la branche locale (inverse)

Objectif : Savoir si notre dépôt local est en avance (un `git push` à opérer ?)

```
$git fetch origin     Récupérer les informations actuelles ⚠
$git log origin/nom_branche..nom_branche
```


Si le dépôt distant n'a pas de retard sur le dépôt local, rien n'est affiché.

Tracking branches (Branches de suivi)

Une **branche de suivi** est une branche locale **mise en relation** avec une branche distante.

Le fait d'avoir défini une branche en tant que « branche de suivi» permet par la suite d'opérer des `git pull` et des `git push` sans avoir à spécifier aucune branche : Par défaut, c'est la branche de suivi qui sera prise en compte !

Comment définir une branche de suivi?

- Utiliser la commande `git branch -u` comme suit :

```
$git checkout branche_locale
$git branch -u origin/branche_distante
```

Une mise en relation est alors opérée entre la branche locale courante et la branche distante.

- Quand on clone un dépôt distant, l'opération crée en local une branche master, cette dernière est mise automatiquement en relation avec la branche `origin/master`

- Si on désire créer une **nouvelle** branche locale à partir d'un `fetch` que l'on a opéré depuis le dépôt distant

```
$git checkout -b nouv_branche_locale --track origin/branche_distante
```

La nouvelle branche créée en local est mise automatiquement en relation avec la branche correspondante distante.

3.4.2 LES COMMANDES

Clonage d'origine d'un dépôt local sur un nouveau dépôt distant

Cette opération est mise en œuvre typiquement par l'administrateur d'un projet de groupe qui vient de créer un nouveau dépôt distant et qui décide d'installer sur ce dernier un code d'origine à partir d'un dépôt local qui lui appartient.

Commandes à effectuer :

```
$ cd /path/to/my/repogit          # Vous placer dans votre espace de travail
$ git remote add origin https://ericlef@bitbucket.org/tutogit/essaijava.git
$ git push -u origin --all        # Push up du dépôt local sur le dépôt distant
$ git push -u origin --tags       # Push up de tous les marqueurs
```

EXPLICATIONS :

```
$ git remote add origin https://ericlef@bitbucket.org/tutogit/essaijava.git
```

Cette commande permet de créer un lien avec le dépôt distant. Ce lien s'appelant « **origin**». Ainsi, au niveau du dépôt local, le serveur distant sera simplement désigné par « `origin` », son url n'aura plus besoin d'être mentionné.

L'URL https://ericlef@bitbucket.org/tutogit/essaijava.git se décompose comme suit:

- `ericlef` → Votre nom d'utilisateur BitBucket (celui que vous avez saisi)
- `tutogit` → Le nom du team
- `essaijava.git` → le nom du dépôt distant

```
$ git push -u origin --all
```

Upload de toutes les branches de votre dépôt local sur le serveur distant **origin**.

Option `--all`

Signale que l'on désire « pousser » toutes les branches de votre dépôt local.
En l'absence de cette option, seule la branche courante est uploadée.

Option `-u`

Pour mettre en place les références « **u**pstream ».
Enregistre notamment le fait que chacune des branches de votre dépôt local sera **suivie** (mise en correspondance) par la branche du dépôt distant qui porte le même nom.
Ce suivi des branches, opéré par défaut, peut être adapté par la suite par vos soins.

```
$ git push -u origin --tags        # Push up de tous les marqueurs
```

Upload de tous les marqueurs sur le serveur distant **origin**.
Notamment, cette opération enregistre l'historique de vos commits sur le dépôt distant.

Clonage d'origine du dépôt distant sur un nouveau dépôt local

Cette opération est mise en œuvre typiquement par un nouveau membre qui se joint au développement d'un projet de groupe dont le code est partagé sur un dépôt distant existant.

1. Cloner le dépôt distant

```
$git clone https://ericlef@bitbucket.org/tutogit/essaijava.git
```

Pour rapatrier tout le code du dépôt distant avec tout son historique des versions dans un dossier portant le nom du dépôt distant.

Ce dossier, qui correspond à votre nouveau dépôt local, peut alors être déplacé là où vous le désirez.

2. Une variante :

```
$git clone https://ericlef@bitbucket.org/tutogit/essaijava.git nom_dossier
```

Le code est rapatrié dans un dossier local portant le nom qui a été spécifié.

3. Contrôler le nouveau dépôt local

Vous placer dans ce dossier contenant le clone

```
$cd essaijava
```

1er contrôle: un `git log` vous affiche tout l'historique des versions

2ème contrôle: un `git remote show origin` pour afficher les relations existantes entre votre dépôt local et le dépôt distant

```
MacBook-Pro-de-Eric:essaijava Eric$ git remote show origin
* remote origin
  Fetch URL: https://ericlef@bitbucket.org/tutogit/essaijava.git
  Push  URL: https://ericlef@bitbucket.org/tutogit/essaijava.git
  HEAD branch: master
  Remote branches:
    master    tracked
    secondary tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
MacBook-Pro-de-Eric:essaijava Eric$
```

Ce show..

- Affiche les branches du dépôt distant
- Signale que la branche master est une **branche de suivi**:

- Un simple **`git push`** aura pour effet d'uploader la branche locale master sur la branche master distante.
- Un simple **`git pull`** aura pour effet de de downloader la branche master et de la fusionner avec votre branche locale master

Afficher l'url du dépôt distant

```
git remote -v
```

Rapatrier depuis le Dépôt distant

```
$git fetch origin
```

La commande `git fetch` ne fait que télécharger les données depuis le dépôt distant. Il faudra par la suite opérer un `git merge` pour fusionner les données téléchargées avec la destination de votre choix.

Plus en détail, cette commande télécharge depuis le dépôt distant toutes les données que vous ne possédez pas encore (qui ont été « pushées » par d'autres entre temps).

Pour la commodité des opérations si on désire une fusion avec l'état du dépôt distant

Si telle ou telle branche locale est configurée pour **suivre** telle ou telle branche distante. Il est alors plus pratique, plutôt que d'opérer un `fetch` + `merge`, d'utiliser la commande **`git pull`** qui, automatiquement, va opérer un fetch puis un merge de la ou des branches observées.

```
$git pull
```

Pour importer uniquement les données d'une branche en particulier

```
$git fetch origin nom_branche
```

```
$git merge origin/nom_branche
```

Pour fusionner les données de telle ou telle branche, téléchargée au préalable par un `fetch`.

Le merge :

- Si vous avez fait des push en même temps que d'autres personnes, les changements qu'ils ont effectués sont alors fusionnés aux vôtres automatiquement.
- Si deux personnes ont modifié en même temps deux endroits distincts d'un même fichier, les changements sont intelligemment fusionnés par Git. S'il arrive que la même zone de code ait été modifiée en même temps, Git vous avertit d'un conflit en indiquant le nom des fichiers.

  A vous de gérer le conflit ! Voir *Merge & Gestion des conflits* en page 22

```
$git pull
```

Cette commande est un raccourci pour `fetch&merge`

Elle télécharge **la branche suivie** du dépôt distant correspondant à la branche courante, puis opère un **merge** de ces données.

Le merge :

- Si vous avez fait des push en même temps que d'autres personnes, les changements qu'ils ont effectués sont alors fusionnés aux vôtres automatiquement.
- Si deux personnes ont modifié en même temps deux endroits distincts d'un même fichier, les changements sont intelligemment fusionnés par Git. S'il arrive que la même zone de code ait été modifiée en même temps, Git vous avertit d'un conflit en indiquant le nom des fichiers.

Sauvegarder sur le dépôt distant

```
$git push nom_dépôt_distant nom_branche
```

➪ Pour sauvegarder une branche locale sur la branche correspondante du dépôt distant.

Typiquement

```
$git push origin une_branche
```

➪ Sauvegarder la branche « une_branche » sur le dépôt distant d'origine

⚠ Attention, cette opération est irréversible !!

⚠ Le changement vers le serveur doit être propre car le serveur ne peut régler les conflits ! Donc personne ne doit avoir fait un `push` depuis votre dernier `pull`.

D'ailleurs, si un autre que vous a opéré un push depuis votre clone dernier clone, votre push sera rejeté.

Concrètement, cela signifie que si vous désirez opérer un merge de vos fichiers sur le dépôt distant, il vous faudra :

1. Opérer un pull depuis le serveur distant et régler les conflits éventuels
2. Opérer un push du résultat

```
$git push
```

➪ Idem. Pour sauvegarder la **branche suivie** courante sur le dépôt distant « origin »

```
$ git push origin nouvelle_branche
```

➪ Pour ajouter la branche `nouvelle_branche` sur le dépôt distant.

Supprimer une branche du dépôt distant

```
$ git push origin --delete une_branche
```

➪ Pour supprimer une branche du dépôt distant.

Inspecter le dépôt distant

```
git remote show origin
```

Pour contrôler la configuration par défaut quand on opère des git pull et git push en fonction des branches. Affiche également l'url du dépôt distant.

Cela donnera par exemple pour un projet contenant une branche master et une branche secondaire nommée »secondary ».

```
MacBook-Pro-de-Eric:EssaiJava Eric$ git remote show origin
* remote origin
  Fetch URL: https://ericlef@bitbucket.org/tutogit/essaijava.git
  Push  URL: https://ericlef@bitbucket.org/tutogit/essaijava.git
  HEAD branch: master
  Remote branches:
    master    tracked
    secondary tracked
  Local branches configured for 'git pull':
    master    merges with remote master
    secondary merges with remote secondary
  Local refs configured for 'git push':
    master    pushes to master    (up to date)
    secondary pushes to secondary (up to date)
MacBook-Pro-de-Eric:EssaiJava Eric$
```

On peut voir dans cet exemple que les branches master et secondary sont des **branches suivies** (`Local branches.. Local refs..`): un simple `git pull` va opérer par défaut un `fetch` et un merge des deux branches master et secondary, et un simple `git push` va opérer également avec les deux branches en question.

4 A essayer par vos soins

Maintenant que vous avez pu tester l'utilisation de GIT pour vos projets, exécuter les opérations suivantes pour vous familiariser avec les procédures GIT.

Faites ces étapes avec les membres de votre projet.

1. Partagez un nouveau dépôt distant avec les autres membres du groupe (avec droit écriture)
2. Créez un projet partagé (au moins 2 ordinateurs)
3. Créez une nouvelle branche à votre projet avec une version différente de la classe « Hello.java »
4. Travaillez sur le même fichier via 2 ordinateurs différents et montrez comment résoudre les conflits entre les 2 versions.

5 Annexe : pull-request

Une bonne pratique de collaboration consiste à utiliser les **pull-request**.

Avant de « pusher » définitivement une nouvelle version personnelle sur le dépôt distant, sur la branche principale par exemple, l'idée consiste dans une première étape à proposer ces modifications à l'ensemble du groupe avec lequel on travaille. Cette opération s'appelle un « pull-request ».

Plutôt que d'être « pushées » sur la branche principale, ces modifications sont pushées sur une branche annexe et les différents collaborateurs reçoivent une invitation à contrôler les modifications de cette branche annexe. Ils peuvent alors les approuver ou, si cela pose problème, le signifier, voire pusher des correctifs sur cette branche annexe.

C'est une fois seulement que le pull-request est approuvé par tout le monde que la branche annexe peut être fusionnée avec la branche principale.

Github

Voir par exemple l'implémentation sur Github à partir du lien suivant :

https://help.github.com/articles/creating-a-pull-request/

Bitbucket

Voir https://www.atlassian.com/git/tutorials/making-a-pull-request

6 Annexe : Installer Git

Vous trouvez ci-après des indications vous permettant d'installer GIT sous Windows, Linux et Mac-Os.

6.1 Installation Windows

Pour utiliser Git sous Windows, il vous faut installer msysgit : https://gitforwindows.org/

Prenez la dernière version, qui installera au passage `msys`, un système d'émulation de commandes Unix sous Windows, et Git.

Les options par défaut conviennent bien, mais vous êtes libre de les modifier (à vos risques).

Une fois l'installation finie, vous pouvez lancer `Git Bash` qui permet d'utiliser Git et les commandes de base d'Unix.

Astuce : Configuration de GIT pour la console windows

A l'usage, le système d'émulation `msys` peut s'avérer un peu lent…

Aussi, il est possible de le court-circuiter..

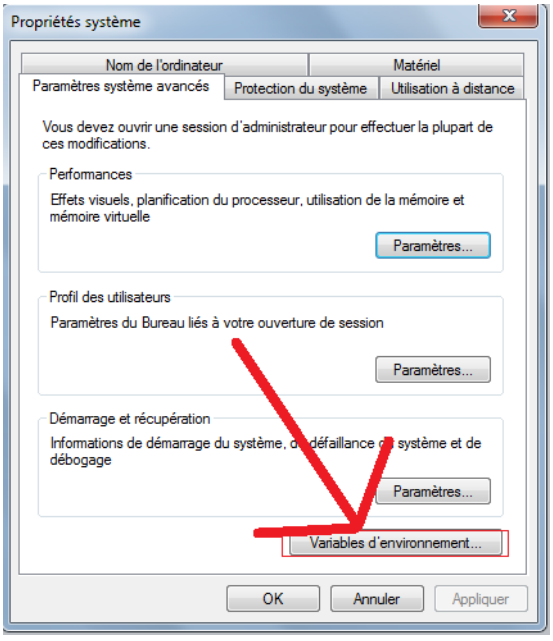
Inconvénient : on ne disposera plus de l'éditeur par défaut.

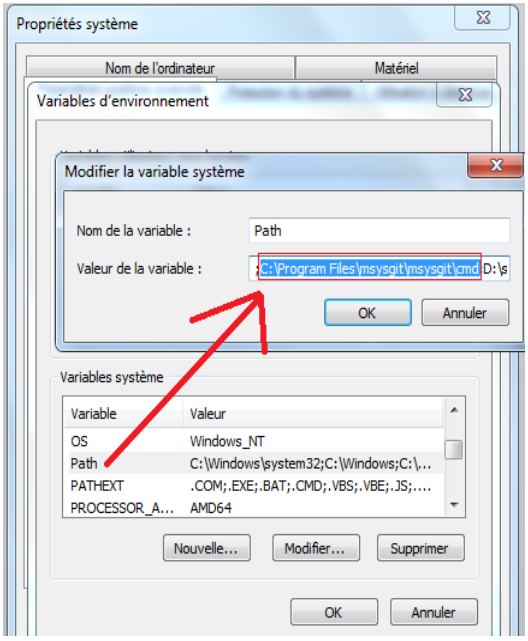
Configuration de la variable d'environnement PATH

`Démarrer` > Click droit sur « `Ordinateur` » > `Propriétés`

Puis choisir à gauche : « `Paramètres système avancés` » et « `Variables d'environnement…` »

Choisir « `PATH` » et cliquer sur « `Modifier` »





Ajouter après les points-virgules le chemin du dossier ‘cmd‘ de msysgit, normalement :

```
C:\Program Files\msysgit\cmd
```

Ou

```
C:\Program Files (x86)\msysgit\cmd
```

Lancer une console Windows : Démarrer > puis rechercher : « cmd.exe » et l’ouvrir.

FACULTATIF : Configuration des clés SSH

Si vous désirez utiliser les clés SSH pour plus de sécurité, il faudra les placer dans un nouveau dossier « .ssh » dans votre répertoire personnel, comme par exemple : c:\Users\YL\.shh

Remarque

Pour créer un dossier commençant par un point, il faudra utiliser la console et taper la commande mkdir .ssh dans votre dossier personnel. Placez les clés publiques et privées à l’intérieur de ce dossier.

Ajouter aussi le fichier « config » dans le dossier .ssh en modifiant les lignes suivantes avec votre pseudo (et votre nom de clé privée s’il est différent de key.private).

```
Host bitbucket.org
User votre_pseudo
IdentityFile ~/.ssh/key.private
```

6.2 INSTALLATION LINUX

Sous Linux Debian/Ubuntu rien de plus simple :

```
sudo apt-get install git-core gitk
```

Sinon, pour plus de renseignements, voir consulter le site :

```
http://git-scm.com/download/linux
```

6.3 INSTALLATION MACOS

Il vous suffit de télécharger cet installeur à l’adresse suivante :

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

Télécharger le paquetage et installer..

Lire éventuellement le fichier README.txt si vous rencontrez plus tard l’avertissement *'The "git" command requires the command line developer tools. Would you like to install the tools now?"* au moment où vous utiliserez GIT.

6.4 CONFIGURER GIT

Maintenant que vous avez installé Git, pour vous aider dans la suite de votre travail, affectez un nom et email à votre dépôt, cela vous permettra d'identifier vos travaux sur le serveur.

```
git config --global user.name "votre_pseudo"
git config --global user.email moi@email.com
```

Configurer également Git pour lui signifier que l'on adopte le nouveau comportement de Git version 2, intitulé « simple behaviour ». Si on ne le fait pas, on court le risque de recevoir des messages d'avertissement intempestifs.

```
git config --global push.default simple
```

6.5 CREER LE DEPOT LOCAL

```
mkdir /path/to/your/project
cd /path/to/your/project
git init            -- création de votre repository local
```

7 Annexe : Revenir à une ancienne version

Restauration globale d'une ancienne version

Les commandes suivantes sont à disposition, en prenant comme exemple la restauration d'une ancienne version sur la branche **`master`**, en s'étant placé au préalable sur la branche master:

- `git checkout master~1` : pour revenir sur l'avant dernier commit (notation équivalente)
- `git checkout master~2` : pour revenir sur l'avant-avant dernier commit (notation équivalente)
- etc..
- `git checkout 36830fa21` : pour revenir sur un numéro de commit précis (seul les premiers caractères sont suffisant tant que l'identifiant utilisé du commit est unique

Raccourcis

- `git checkout master^` : avant-dernier commit
- `git checkout master^^` : avant-avant-dernier commit

Notes importantes

- Cette opération restaure l'état d'une ancienne version dans l'espace de travail.
- **Elle ne supprime aucun commit !** *(→ cette opération est donc réversible)*
  On se retrouve dans un état « DETACHED HEAD », signifiant que le pointeur HEAD pointe sur un commit qui n'est pas le commit le plus récent de la branche courante.

  Rappelons que la suppression de commit s'opère avec un git reset
  Voir Annuler un commit : git reset HEAD^ en page 18

Restaurer l'ancienne version d'un fichier

Nous avons vu comment revenir à la **dernière version d'un fichier de la branche en cours**:

⇨ `git checkout nom_fichier`

Pour restaurer l'état que ce fichier possédait à un certain moment dans une branche en particulier, il suffit de compléter la commande précédente de restauration de version par le nom du fichier.

Comme par exemple :

```
$ git checkout master^^ Hello.java
```

⇨ Restauration de l'avant-avant dernier commit du fichier `Hello.java` de la branche `master`.

8 Annexe : Travailler avec IntellIJ

L'IDE **IntelliJ** de JetBrains intègre d'office un certain nombre de fonctionnalités Git.

À vous de les découvrir ! Voici cependant ci-dessous quelques petites indications afin de bien démarrer.

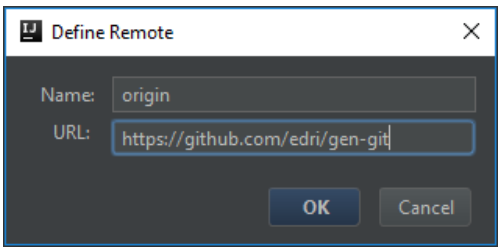
Initialisation du contrôle de version dans IntelliJ

Il existe deux méthodes quant à l'initialisation du contrôle de version dans IntelliJ : créer un nouveau projet (ce que vous allez faire dans ce tutoriel) ou alors récupérer un projet existant depuis Git.

Créer un nouveau projet et y intégrer un contrôle de version Git

Il est possible de créer un dépôt local et d'initialiser Git au niveau de son répertoire source (répertoire `src`) directement depuis IntelliJ. Pour rappel, il s'agit de ce que vous allez faire dans ce tutoriel.

1. Ouvrir le projet (ou le créer).
2. Onglet `VCS` > `Import into Version Control > Create Git Repository`.
3. Sélectionner le répertoire dans lequel créer le dépôt (celui de l'application),
4. Créer un nouveau projet vierge sur le serveur Git (BitBucket, GitHub, …) et noter l'URL.
5. Configurer l'URL du dépôt distant via l'onglet `VCS` > `Git` > `Remotes`….

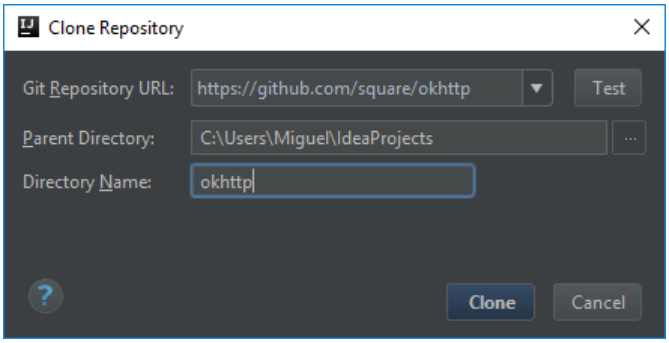


Charger un projet Git déjà existant

À titre informatif, il est possible de créer un projet local en récupérant le code directement depuis un dépôt distant :

1. Ouvrir IntelliJ, et choisir l'option `Check out from Version Control`, ou alors onglet `File` > `New` > `Project from Version Control` > `Git`.

2. Indiquer les détails du projet, puis cloner le dépôt distant et créer un nouveau projet avec ces sources.

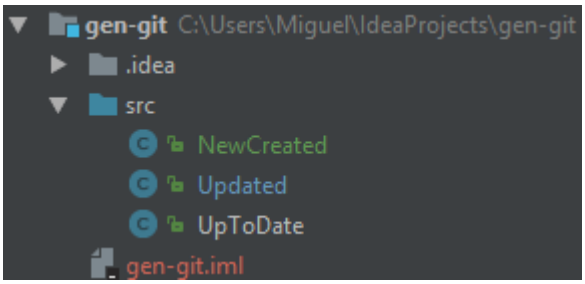


3. Le statut du clonage apparaît dans la barre de tâches situé en bas de l'IDE.

Informations de base

Voici quelques informations de base quant à l'utilisation du contrôle de version dans IntelliJ :

- L'onglet **VCS** (**V**ersion **C**ontrol **S**ystem) rassemble l'accès aux différentes fonctionnalités Git.
- Dans l'arbre de fichiers du projet, les fichiers créés mais pas encore `commit` apparaissent en **vert**, les fichiers édités mais pas encore `commit` apparaissent en bleu, et les fichiers à jour apparaissent en **blanc**. Les fichiers qui ne sont pas suivis par Git et qui ne seront donc pas `push` apparaissent en **rouge**.



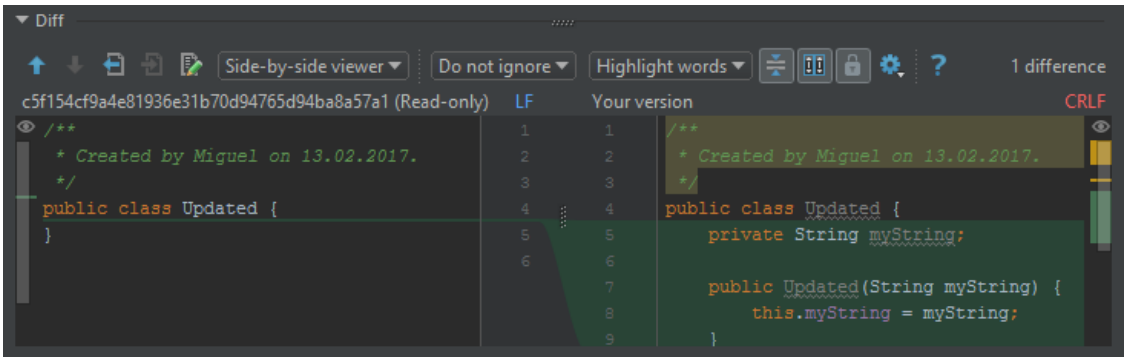
- Lors de la création d'un nouveau fichier, l'IDE vous demande si vous souhaitez l'ajouter à la liste des fichiers suivis par Git, et si vous souhaitez donc ne pas ignorer le nouveau fichier. Pensez donc à l'ajouter, si vous souhaitez pouvoir l'intégrer au VCS.
- Lorsque des changements ont été `commit`, il ne faut pas oublier de `pull` la dernière version du dépôt, puis de `push` les modifications, afin d'éviter des conflits lors de cette dernière commande. Rappelez-vous, donc : `commit` > `pull` > `push`.
- Veillez à régulièrement `commit` vos fichiers !

Liste des actions réalisables via IntelliJ

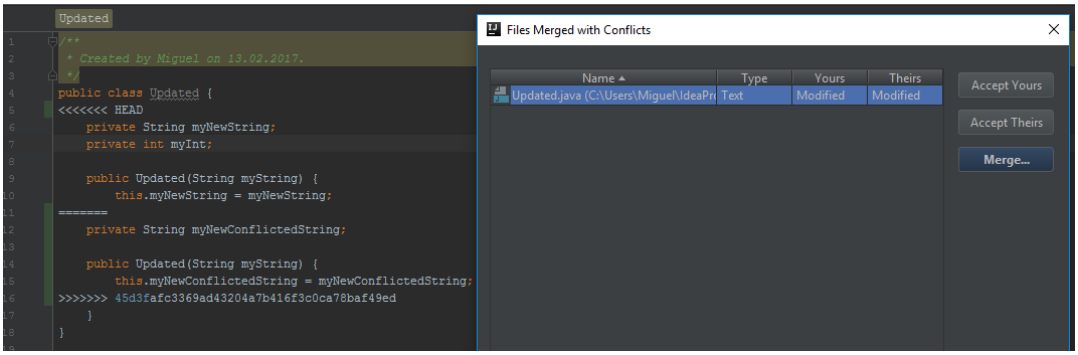
À partir de là, vous allez être volontairement lâchés dans le vaste monde d'IntelliJ, car nous souhaitons vous inciter à découvrir les choses intéressantes par vous-mêmes, vous permettant d'apprendre plus facilement. A vous donc de trouver toutes les fonctionnalités intéressantes (et n'oubliez pas d'aller fouiller les options disponibles dans l'onglet `VCS`) !

Voici cependant une liste non-exhaustive de ce qu'il est possible de faire, avec quelques explications :

- `commit` les changements apportés au code ; veillez à toujours mettre des messages de commit explicites pour vos collègues. Avant l'opération, il est possible de voir un petit résumé des modifications depuis le dernier commit :



- `pull` la dernière version du code ; des conflits peuvent apparaître avec la dernière version du code :

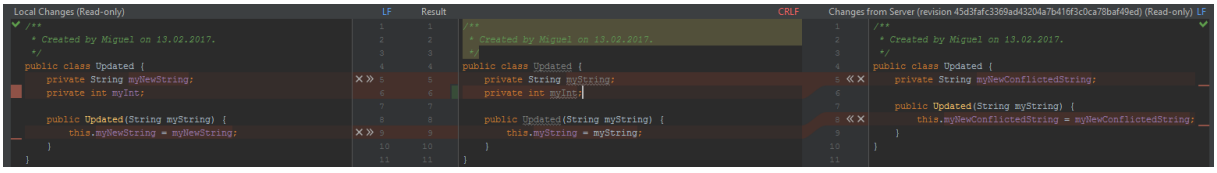


À partir de là, il est possible de :

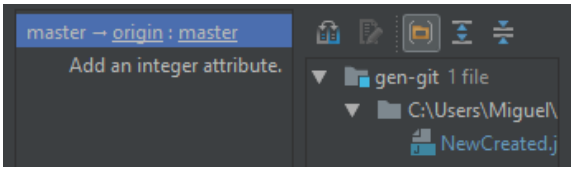
1. Garder vos changements et d'effacer ceux qui sont sur le serveur (`Accept Yours`) => à éviter, si vous ne voulez pas effacer tout le code écrit par votre cher collègue, et si vous ne souhaitez pas affronter son courroux.

2. Effacer vos changements et de garder ceux qui sont sur le serveur (`Accept Theirs`) => à éviter aussi, si vous ne voulez pas perdre tous les changements locaux.

3. Fusionner manuellement les changements => il s'agit de l'option la plus sage, qui vous permet de vois les deux versions, et de sélectionner vous-mêmes les changements à garder ou non, après avoir discuté avec la personne dont le code crée un conflit.

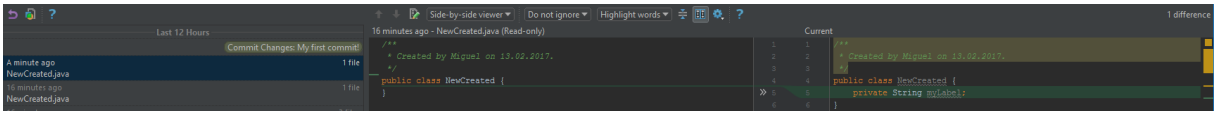


- `push` les modifications sur le serveur Git.



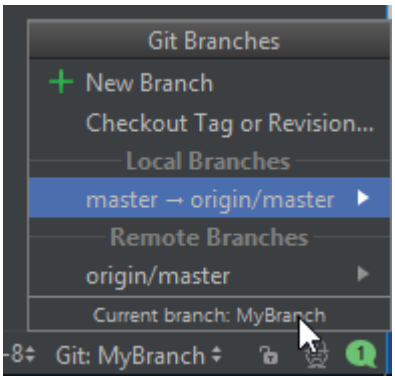
Attention, si vous avez oublié de `pull` les dernières données et qu'un conflit survient, le serveur refusera votre `push`.

- Afficher l'historique des changements locaux du projet ou des changements effectués depuis le dernier `commit`. Il s'agit là d'une fonctionnalité très intéressante, car elle vous permet de comparer les différentes versions du code, afin de pouvoir rapidement visualiser les changements.



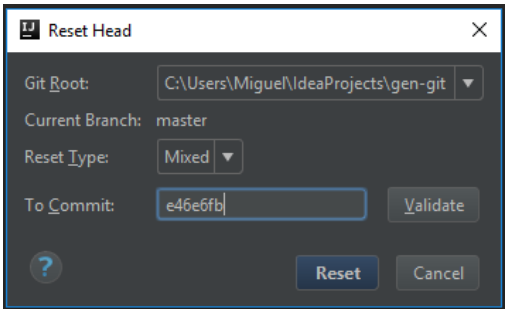
- Comparer la version actuelle du code avec d'autres versions antérieures.

- Créer et gérer des branches. Voici un indice : il est possible de voir le nom de la branche dans laquelle vous vous trouvez, ainsi que d'ouvrir une fenêtre de gestion des branches grâce à la barre de tâches d'IntelliJ.



- Ajouter des tags de version.

- Réinitialiser le code à un état antérieur à l'aide d'un numéro de commit. Attention, la réinitialisation est irréversible !



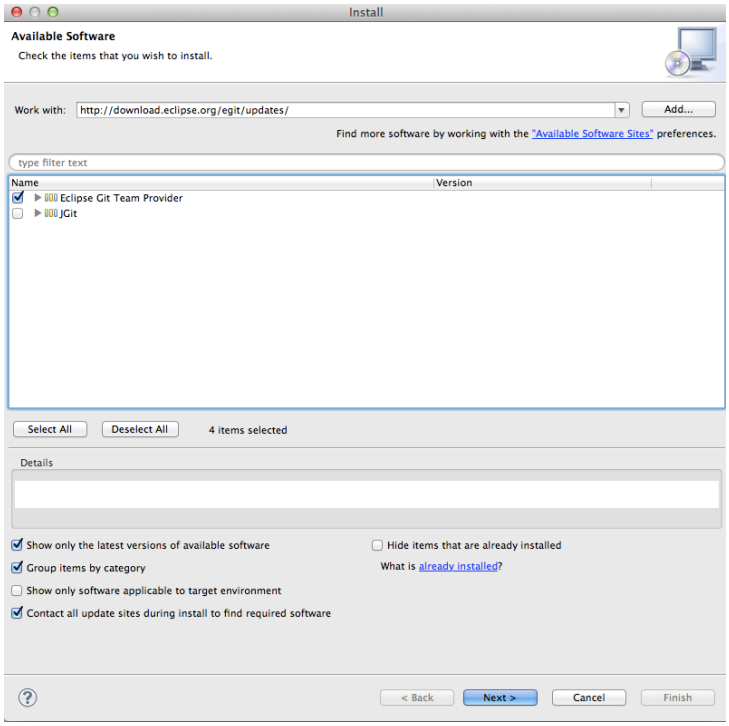
9 Annexe : Travailler avec le plugin Egit pour Eclipse

Pour les allergiques du travail en console, Eclipse met à votre disposition un plugin tout adapté à Git, le plugin EGit. EGit s'appuie sur la librairie JGit, une librairie qui implémente la fonctionnalité GIT en Java.

Pour l'installation, allez dans le menu `Help` → `Install new Softwares…`

Il faut alors ajouter l'URL suivante : `http://download.eclipse.org/egit/updates/`

Puis installer le plugin comme dans l'image ci-dessous :

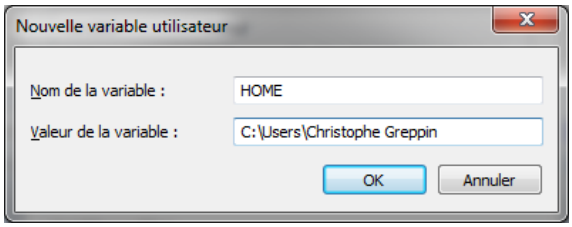


Veuillez cocher la case «`Contact all update sites during install to find required software`», si vous obtenez une erreur du type « `Missing requirement: EGit Mylyn …` »

9.1 VARIABLE D'ENVIRONNEMENT HOME

Le plugin a besoin de la variable d'environnement « `HOME` » pour fonctionner.

Sous Windows, contrôlez que cette variable existe en lançant le panneau de configuration des variables d'environnement. Le cas échéant, ajoutez ensuite la variable « HOME » qui pointe sur votre dossier utilisateur.



Fermez et relancez Eclipse afin que le plugin se lance et que la variable d'environnement soit prise en compte. Nous sommes maintenant prêt à utiliser GIT avec Eclipse.

Sous Mac-OS & Linux, contrôlez l'existence de cette variable en tapant la commande « `env` » dans une fenêtre de commandes.

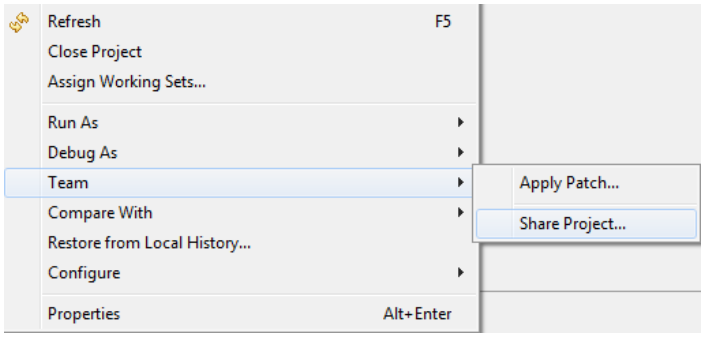
9.2 PROJET DANS ECLIPSE

Comme vu précédemment, GIT repose à la base sur un système non centralisé. Il va donc falloir créer un répertoire de travail local pour stocker notre projet et un repository local GIT qui sera dédié au versionning.

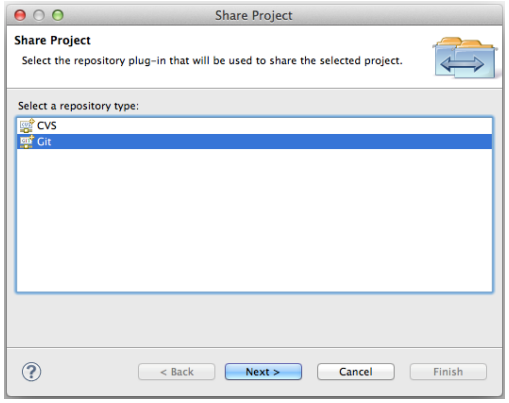
Nous associerons ce dernier au repository distant afin de pouvoir sauvegarder nos différentes versions du projet sur le serveur distant. Lors de la création de votre repository chez bitbucket, vous avez dû recevoir un URL qui nous servira à faire cette liaison.

Création du projet JAVA et association au répertoire local

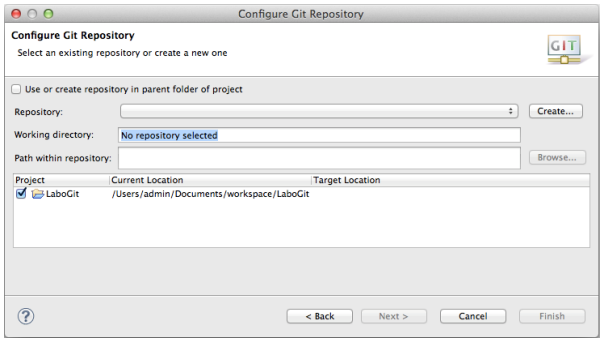
La première étape consiste à créer votre nouveau projet JAVA comme vous avez l'habitude de le faire via `File`→ `New` → `Java Project`. Une fois le projet créé, mettez-le en mode versionning avec GIT. Faites un clic droit sur le nom du projet et sélectionnez l'option `Team`→ `Share Project`…



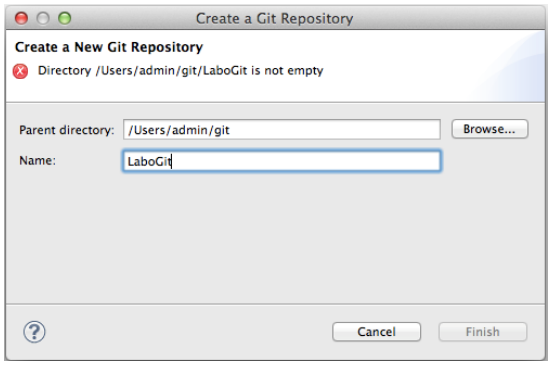
Dans la fenêtre qui s'ouvre, choisissez le plug-in « `Git` » dans la liste puis cliquez sur « `Next >` ».



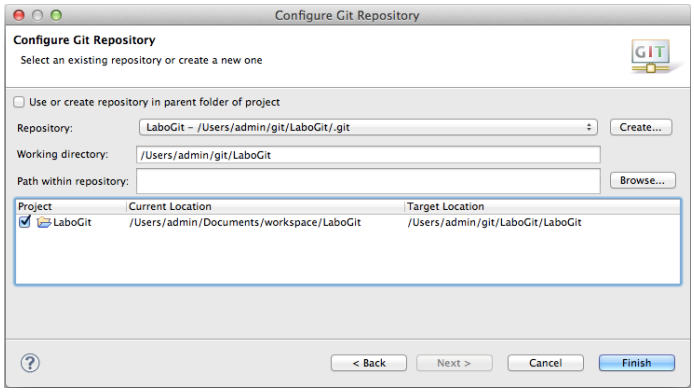
Une nouvelle fenêtre apparaît :



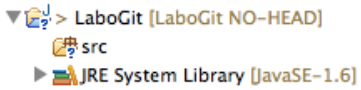
Utilisez le bouton « `Create` » pour créer votre répertoire local où vous souhaitez sauvegarder les sources de vos projets et donnez-lui un nom : « `LaboGit` », comme dans l'exemple ci-dessous :



Appuyez sur « Finish »..

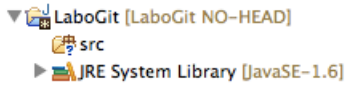


Appuyez sur « Finish » pour terminer la création du répertoire local.



Ajouter le projet au système de gestion de versions

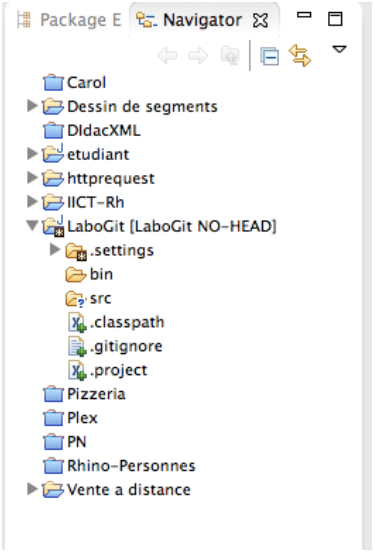
Allez dans le menu « Team » (clic droit sur le projet) et choisissez l'option « `Add to Index` » ou « `Add` » (selon la version) pour ajouter l'ensemble du projet au système de gestion de version.



« Ignorer » certains fichiers

Assurez-vous que le répertoire /bin du projet sera ignoré par le gestionnaire GIT.

Dans la fenêtre Navigateur, vous voyez apparaître le fichier « **`.gitignore`** ».



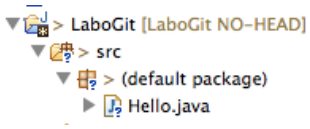
Ouvrez-le, vous verrez qu'il contient déjà le répertoire `/bin`.

Sinon, pour rajouter un dossier ou un fichier à l'ensemble des fichiers à ignorer, opérez un click droit sur l'objet à ignorer et choisissez l'option « `Ignore` » du menu « `Team` ».

Créer une première classe « Hello.java » dans notre projet et la sauvegarder

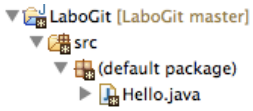
Maintenant, créez votre première classe JAVA dans votre projet..

L'icône associée à votre nouveau fichier contient alors une icône avec un point d'interrogation. Cela permet de signaler que le fichier est nouveau et encore inconnu du système de gestion de version de GIT.

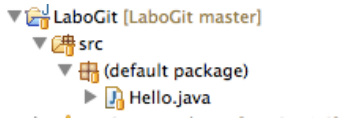


Afin de sauvegarder son état :

1/ Ajoutez-le à la zone d'assemblage : `Team > Add to Index`



2/ Opérez un commit : `Team > commit`



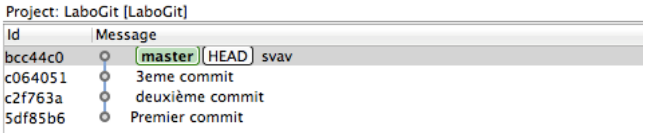
Notez qu'il vous aura fallu saisir un commentaire avant de pouvoir terminer l'action.

En l'état, l'état de votre fichier est sauvegardé, vous avez créé une nouvelle version de votre projet.

Dans notre cas, nous allons envoyer tous les fichiers de la liste. Une fois terminé, votre projet contient le label `[Master]` correspondant au nom de la branche. Grâce à cette commande, le fichier est sauvegardé dans GIT dans la version locale uniquement.

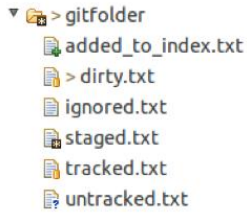
Voir l'historique des versions de votre projet

Pour voir l'historique des versions : `Team > Show` in `history`



9.3 L'ETAT D'UN FICHIER DANS LE MONDE EGIT

La vue « Package Explorer » de Eclipse associe à chacun des fichiers une petite icône qui signale l'état du fichier.



Au sein de EGit, un fichier peut se trouver dans l'un ou l'autre des états suivants, ces états pouvant être cumulatifs :

- **added to index**
  Etat d'un fichier qui a a été ajouté à la zone d'assemblage (et donc en attente d'un commit) et qui était inconnu du Git jusque là (ce qui le distingue d'un état « staged », voir ci-dessous).
- **dirty**
  Le symbole « dirty » est simplement indiqué par le caractère « > ».

  Un fichier « dirty » est un fichier qui a été modifié depuis son dernier placement dans la zone d'assemblage ou depuis son dernier commit.
- **ignored**
  Etat d'un fichier non pris en compte par Git (sur notre demande)
- **staged**
  Etat d'un fichier qui a a été ajouté à la zone d'assemblage (et donc en attente d'un commit).
- **tracked**
  Etat d'un fichier qui n'a pas été modifié depuis son dernier commit
- untracked

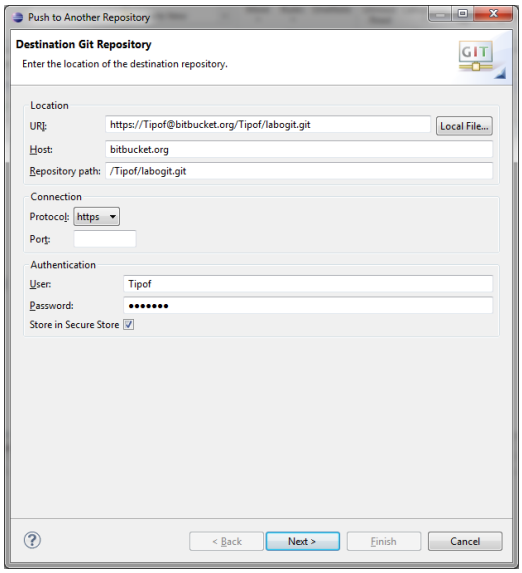
Etat d'un fichier qui n'a pas encore été ni committé (sauvegardé), ni placé dans la zone d'assemblage (en bref, un nouveau fichier..)

Remarquons qu'un fichier qui se trouve à la fois dans les états `dirty` et `staged` signifie que ce fichier a été modifié depuis qu'il a été placé dans la zone d'assemblage.

9.4 SYNCHRONISER LE PROJET AVEC LE SERVEUR DISTANT

La prochaine étape consiste à synchroniser notre version locale avec celle du serveur distant. Pour ce faire, il faut choisir l'option `Remote`→`Push du menu` « `Team` » de notre projet JAVA.

Entrez l'URL donné par bitbucket lors de la création du repository ainsi que votre login et mot de passe.



Cliquez sur « Next ».

> La nouvelle fenêtre vous permet de spécifier la liste branches que vous désirez synchroniser.

Puis cliquez à nouveau sur « Next ».

> Un dialogue de confirmation vous montre un résumé de ce qui va être envoyé sur le serveur distant.

Cliquez sur « Finish » pour terminer l'opération.

Vous devrez **faire un commit** chaque fois que vous souhaiterez enregistrer une nouvelle version dans GIT. D'ailleurs tout fichier qui a subi un changement et qui n'est pas encore enregistré dans GIT contient le symbole « > » devant le nom.

⚠ **L'opération « `push` » sur le serveur se fait en général quand une version de vos sources est considérée comme stable et prête à être partagée avec d'autres membres du projet.**

9.5 IMPORTER UN PROJET EXISTANT

Dans le cas où le projet existe déjà sur le répertoire distant, il vous faut opérer un import via `File→Import` en choisissant l'option « `Projects from Git` » dans la liste.

Sélectionnez l'option URI et tapez votre URL donné par bitbucket dans le premier champ. Les autres champs vont se remplir automatiquement.

Pensez à mettre également votre mot de passe associé à votre compte.

- Sélectionnez les branches du projet à importer que vous souhaitez synchroniser. Cliquez ensuite sur le bouton « `Next >` ».
- Choisissez ensuite la destination où vous souhaiter enregistrer la version locale du système de gestion de version. Cliquer ensuite sur « `Next >` ».
- Il vous faut maintenant créer le projet JAVA dans Eclipse, pour cela choisissez l'option « `Import existing projects` » et appuyez sur « `Finish` ».