



## FORMATION GIT



## Git au quotidien

- Créer un dépôt
- Configurer ses outils pour travailler
- Le commit local
- Le push, le pull
- Visualiser l'arbre de travail
- Une branche, pourquoi faire ?

## Les tickets dans GitHub

- Intérêts des tickets
- Définir une roadmap
- Affecter des tickets à des utilisateurs
- Les tickets envoyés par email

## Plus loin avec Git

- Créer des branches / Merge de branches
- Gérer les branches en local
- Gérer les conflits
- annuler un commit
- Git reset
- La révision de code avec Git
- Exporter son dépôt

## Le wiki dans GitHub

- Avantages d'un wiki
- La syntaxe wiki
- Créer et gérer des pages wiki dans son projet GitHub
- Les droits sur le wiki
- Créer son site public avec Github



## OBJECTIFS DE LA FORMATION

À la fin de cette formation, vous devriez être capable de:

- configurer et initialiser un dépôt GIT,
- commencer et arrêter le suivi de version de fichiers
- d'indexer et valider des modifications.

Nous verrons aussi comment :

- paramétriser Git pour qu'il ignore certains fichiers ou patrons de fichiers,
- revenir sur les erreurs rapidement et facilement,
- parcourir l'historique de votre projet et voir les modifications entre deux validations
- pousser et tirer les modifications avec des dépôts distants.



- Cette partie de formation sur les Bases de Git offre un bref aperçu des commandes Git les plus importantes.
- La section Configuration d'un dépôt explique tous les outils dont vous aurez besoin pour démarrer un nouveau projet avec un système de gestion de versions.
- Les sections suivantes vous présenteront des commandes Git usuelles.

À la fin de ce module, vous devriez être capable de :

- ✓ créer un dépôt Git,
- ✓ enregistrer des snapshots de votre projet pour les mettre en lieu sûr,
- ✓ consulter l'historique de votre projet.



# Les bases de Git

## Démarrer un dépôt Git

- Vous pouvez principalement démarrer un dépôt Git de deux manières.
  - ✓ La première consiste à prendre un projet ou un répertoire existant et à l'importer dans Git. ➔ **git init**
  - ✓ La seconde consiste à cloner un dépôt Git existant sur un autre serveur ➔ **git clone**



# Les bases de Git

## Démarrer un dépôt Git

La commande **git init** initialise un nouveau dépôt Git.

```
git init
```

Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt.

```
git init <rédertoire>
```

Crée un dépôt Git vide dans le répertoire précisé. Quand on exécute cette commande, cela crée un nouveau répertoire intitulé `<rédertoire>` ne contenant rien d'autre qu'un sous-répertoire `.git`.

```
git init --bare <rédertoire>
```

Par convention, les dépôts initialisés avec l'option `--bare` ont leur nom finissant en `.git`.



# Les bases de Git

## EXEMPLE D'UTILISATION PRATIQUE

➤ Si vous souhaitez commencer à suivre en version des fichiers existants (contrairement à un répertoire vide), vous devriez probablement:

- ✓ commencer par indexer ces fichiers
- ✓ faire une validation initiale.
- ✓ suivi d'un commit

```
$ git add *.c
$ git add README
$ git commit -m 'version initiale du projet'
```



# Les bases de Git

## Cloner un dépôt existant: git clone

Si vous souhaitez obtenir une copie d'un dépôt Git existant, par exemple, un projet auquel vous aimeriez contribuer:

la commande dont vous avez besoin s'appelle **git clone** .

### Lab.

1-Créez à la racine de C: un répertoire nommé git, déplacez-vous à la racine de ce répertoire

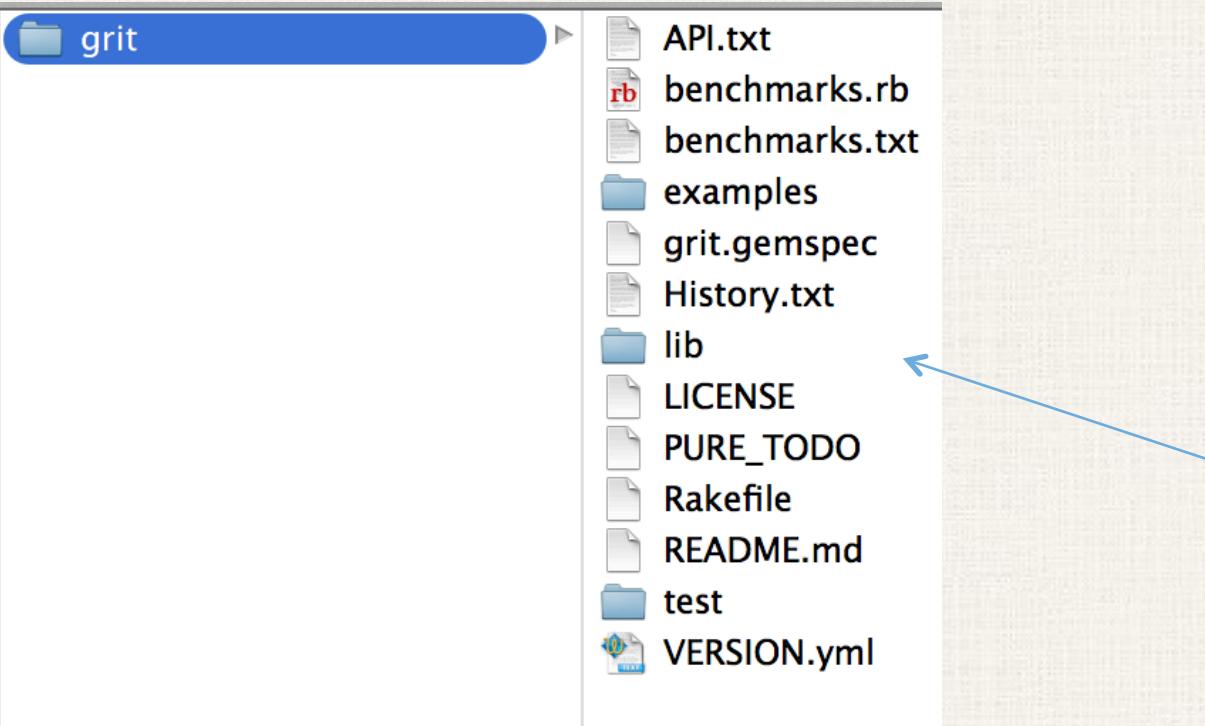
```
$ git clone git://github.com/schacon/grit.git
```

```
MacBook-Pro-de-Odellya:git odellya$ git clone git://github.com/schacon/grit.git
Cloning into 'grit'...
remote: Counting objects: 4051, done.
remote: Compressing objects: 100% (2824/2824), done.
remote: Total 4051 (delta 1170), reused 4051 (delta 1170)
Receiving objects: 100% (4051/4051), 2.10 MiB | 505.00 KiB/s, done.
Resolving deltas: 100% (1170/1170), done.
Checking connectivity... done.
MacBook-Pro-de-Odellya:git odellya$
```



## Les bases de Git

2-Vérifiez que vous avez la création d'un répertoire grit avec le contenu ci-dessous



Par commodité, le clonage d'un dépôt Git crée automatiquement une connexion distante intitulée « origin », qui pointe vers le dépôt d'origine.

Il est ainsi très facile d'interagir avec le dépôt central.

**Conclusion :** Vous avez à présent un dépôt Git valide et une extraction ou copie de travail du projet.

- Quand vous clonez un dépôt pour la première fois, **tous les fichiers seront sous suivi de version et inchangés** car vous venez tout juste de les enregistrer sans les avoir encore édités.

# Les bases de Git

## INFOS

Git dispose de différents protocoles de transfert que vous pouvez utiliser.

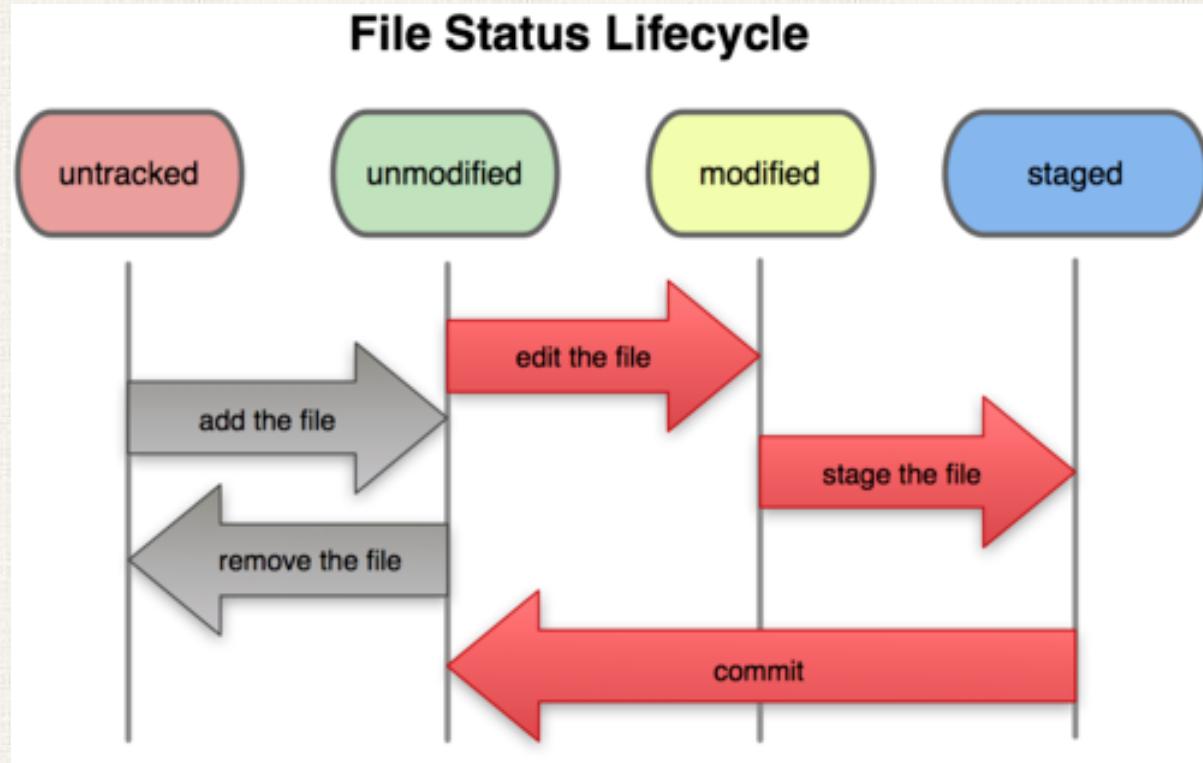
L'exemple précédent utilise le protocole **git://**, mais vous pouvez aussi voir **http(s)://** ou [utilisateur@serveur:/chemin.git](#)

qui utilise le protocole de transfert SSH.



# Les bases de Git

## Enregistrer des modifications dans le dépôt



- chaque fichier de votre copie de travail peut avoir deux états : sous suivi de version ou non suivi.
- Les fichiers suivis sont les fichiers qui appartenaient déjà au dernier instantané ;
- ils peuvent être inchangés, modifiés ou indexés.
  
- Tous les autres fichiers sont non suivis

# Les bases de Git

## Vérifier l'état des fichiers

- L'outil principal pour déterminer quels fichiers sont dans quel état est la commande **git status** .  
Si vous lancez cette commande juste après un clonage, vous devriez voir ce qui suit :

### Lab.

1-Saisir la commande **git status**

```
MacBook-Pro-de-Odella:y$ git status
On branch master

Initial commit

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

    grit/
nothing added to commit but untracked files present (use "git add" to track)
MacBook-Pro-de-Odella:y$
```

Questions :

- Que signifie le résultat obtenu ?
- Sur quelle branche êtes-vous?
- Quelle différence faites-vous entre git status et git log?



# Les bases de Git

## Lab.

2-Ajoutez dans le dossier grit le fichier **lisezmoi.txt**, et déplacez vous dans le répertoire du projet grit

3-faire git status de nouveau comme sur l'image ci-dessous.

```
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lisezmoi.txt

nothing added to commit but untracked files present (use "git add" to track)
MacBook-Pro-de-Odellya:grit odellya$ '
```

Expliquez la présence du fichier **lisezmoi.txt** dans la section « Untracked files » ?



# Les bases de Git

## Placer de nouveaux fichiers sous suivi de version

### Lab.

1- Pour commencer à suivre un nouveau fichier, vous utilisez la commande **git add** . Pour commencer à suivre le fichier LISEZMOI, vous pouvez entrer ceci :

```
MacBook-Pro-de-Odellya:grit odellya$ git add lisezmoi.txt
MacBook-Pro-de-Odellya:grit odellya$
```

2- lancez à nouveau la commande **git status**,  
Questions : quel constat faites-vous ?

```
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   lisezmoi.txt

MacBook-Pro-de-Odellya:grit odellya$
```

Analysez le résultat obtenu

# Les bases de Git

## Indexer des fichiers modifiés

### LAB.

- 1-modifiez un fichier qui est déjà sous suivi de version, **benchmarks.rb** et Lancez de nouveau la commande **git status** comme sur l'image ci-dessous.

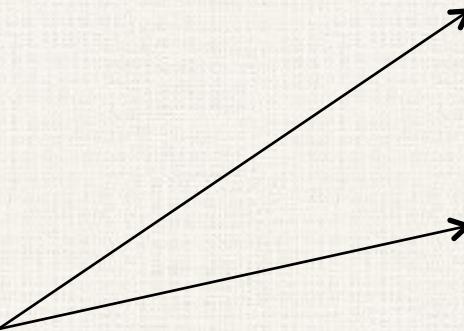
```
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:  lisezmoi.txt

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:   benchmarks.rb

MacBook-Pro-de-Odellya:grit odellya$
```



Analysez ce résultat

- 2- indexer le fichier **benchmarks.rb** pour être mis dans la zone tampon avec **git add benchmarks.rb**

```
MacBook-Pro-de-Odellya:grit odellya$ git add benchmarks.rb
MacBook-Pro-de-Odellya:grit odellya$
```



# Les bases de Git

## LAB

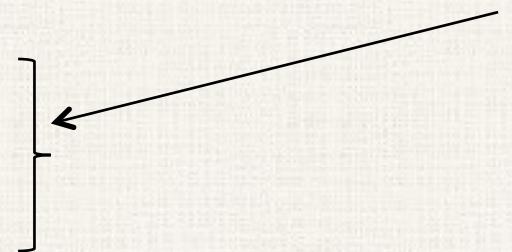
3-lancez de nouveau la commande **git status**

```
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   benchmarks.rb
    new file:   lisezmoi.txt

MacBook-Pro-de-Odellya:grit odellya$
```

analysez de nouveau le résultat



## Ignorer des fichiers

- Il apparaît souvent qu'un type de fichiers présent dans la copie de travail ne doit pas être ajouté automatiquement ou même apparaître comme fichier potentiel pour le suivi de version.
- Ce sont par exemple des fichiers générés automatiquement tels que les fichiers de journaux ou de sauvegardes produits par l'outil que vous utilisez.
- Dans un tel cas, on peut énumérer les patrons de noms de fichiers à ignorer dans un fichier **.gitignore**.

Voici ci-dessous un exemple de fichier **.gitignore** :

```
$ cat .gitignore
*.o [oa] ~
```

Ignore des fichiers se terminant par .o ou .a ou encore un patron de fichier se terminant par tilda



## Inspecter les modifications indexées et non indexées **git diff**

➤ Cette commande est utilisée le plus souvent pour répondre aux questions suivantes :

- ✓ qu'est-ce qui a été modifié mais pas encore indexé ?
- ✓ Quelle modifications a été indexée et est prête pour la validation ?

Là où git status répond de manière générale à ces questions, **git diff** montre les lignes exactes qui ont été ajoutées, modifiées ou effacées



## Inspecter les modifications indexées et non indexées **git diff**

### LAB

1- Supposons que vous éditez et indexez le fichier LISEZMOI et que vous éditez le fichier benchmarks.rb sans l'indexer.

2-lancer la commande **git diff**

### 3-Analyser le résultat en ligne de commande

```
MacBook-Pro-de-Odellya:grit odellya$ git diff
diff --git a/lisezmoi.txt b/lisezmoi.txt
index f9d9a7b..3c05139 100644
--- a/lisezmoi.txt
+++ b/lisezmoi.txt
@@ -6,7 +6,7 @@ commits 2      0.110000   0.170000   0.860000 ( 0.896371)
 log        0.350000   0.130000   0.850000 ( 0.875035)
 diff       0.190000   0.140000   1.940000 ( 2.031911)
 commit-diff 0.540000   0.220000   1.390000 ( 1.463839)
-heads     0.010000   0.070000   0.390000 ( 0.413918)
+
+
Grit (with GitRuby) :
MacBook-Pro-de-Odellya:grit odellya$
```

## Inspecter les modifications indexées et non indexées **git diff**

- Cette commande compare le contenu du répertoire de travail avec la zone d'index.
- Le résultat vous indique les modifications réalisées mais non indexées.
- Si vous souhaitez visualiser les modifications indexées qui feront partie de la prochaine validation, vous pouvez utiliser **git diff --cached**

## Valider vos modifications: git commit

- Votre zone d'index est dans l'état désiré, vous pouvez valider vos modifications.
- tous les fichiers qui ont été créés ou modifiés et qui n'ont pas subi de **git add** depuis ne feront pas partie de la prochaine validation.
- Ils resteront en tant que fichiers modifiés sur votre disque.

### LAB

1-réalisez la suite des commandes comme ci-dessous:

```
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   benchmarks.rb
    new file:   lisezmoi.txt

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:   lisezmoi.txt

MacBook-Pro-de-Odellya:grit odellya$ git add lisezmoi.txt
MacBook-Pro-de-Odellya:grit odellya$ █
```

## LAB

Réalisation du premier commit de vos changements: **git commit**

```
MacBook-Pro-de-Odellya:grit odellya$ git commit
[master f4bb5bf] premier commit réalisé :wq
Committer: Odellya Consulting <odellya@MacBook-Pro-de-Odellya.local>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:
```

```
git config --global user.name "Your Name"
git config --global user.email you@example.com
```

After doing this, you may fix the identity used for this commit with:

```
git commit --amend --reset-author
```

```
2 files changed, 22 insertions(+)
create mode 100644 lisezmoi.txt
MacBook-Pro-de-Odellya:grit odellya$
```

## LAB

2-Refaire la commande **git status** et faites-en une analyse du résultat en ligne de commande

```
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
MacBook-Pro-de-Odellya:grit odellya$ █
```

## Éliminer la phase d'indexation

- L'ajout de l'option **-a** à la commande **git commit** ordonne à Git de placer automatiquement tout fichier déjà en suivi de version dans la zone d'index avant de réaliser la validation, évitant ainsi d'avoir à taper les commandes git add :



## Effacer des fichiers

- Pour effacer un fichier de Git, vous devez l'éliminer des fichiers en suivi de version (plus précisément, l'effacer dans la zone d'index) puis valider.
- La commande **git rm** réalise cette action mais efface aussi ce fichier de votre copie de travail de telle sorte que vous ne le verrez pas réapparaître comme fichier non suivi en version à la prochaine validation.
- Si vous effacez simplement le fichier dans votre copie de travail, il apparaît sous la section “**Changed but not updated**” (c'est-à-dire, non indexé) dans le résultat de **git status** :

## LAB

1-Lancez successivement les commandes **rm <fichier à supprimer>** et **git status**

```
MacBook-Pro-de-Odellya:grit odellya$ rm grit.gemspec
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:    grit.gemspec

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-de-Odellya:grit odellya$ █
```

2-Analysez le résultat obtenu

3- lancez **git rm <fichier à supprimer>** puis **git status**



```
MacBook-Pro-de-Odellya:grit odellya$ git rm grit.gemspec
rm 'grit.gemspec'
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    grit.gemspec

MacBook-Pro-de-Odellya:grit odellya$ █
```

4-Analyser le résultat obtenu



Questions: Que va t-il se passer lors de la prochaine validation ?



## Déplacer des fichiers

- À la différence des autres VCS, Git ne suit pas explicitement les mouvements des fichiers.
- Si vous renommez un fichier suivi par Git, aucune méta-donnée indiquant le renommage n'est stockée par Git.
- Néanmoins, Git est assez malin pour s'en apercevoir après coup
- De ce fait, que Git ait une commande **mv** peut paraître trompeur. Si vous souhaitez renommer un fichier dans Git, vous pouvez lancer quelque chose comme

## LAB

Lancez successivement les commandes

1-git mv lisezmoi.txt LISEZMOI

2-git status

```
MacBook-Pro-de-Odellya:grit odellya$ git mv lisezmoi.txt LISEZMOI
MacBook-Pro-de-Odellya:grit odellya$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    lisezmoi.txt -> LISEZMOI
    deleted:   grit.gemspec

MacBook-Pro-de-Odellya:grit odellya$ █
```

## Visualiser l'historique des validations

- Après avoir créé plusieurs commits ou si vous avez cloné un dépôt ayant un historique de commits, vous souhaitez probablement revoir le fil des évènements. La commande **git log** est l'outil le plus basique et puissant pour cet objet.

## LAB

1-revenez sur le répertoire git à la racine de C:

2-lancez la commande comme sur l'image ci-dessous

```
MacBook-Pro-de-Odellya:git odellya$ cd ..
MacBook-Pro-de-Odellya:git odellya$ git clone git://github.com/schacon/simplegit-progit.git
Cloning into 'simplegit-progit'...
remote: Counting objects: 13, done.
remote: Total 13 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (3/3), done.
Checking connectivity... done.
MacBook-Pro-de-Odellya:git odellya$
```

3-Que fais cette commande ? Faites ensuite l'analyse du résultat de la commande **git log**



## Visualiser l'historique des validations

### LAB

```
MacBook-Pro-de-Odellya:git odellya$ cd simplegit-progit/
MacBook-Pro-de-Odellya:simplegit-progit odellya$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gmail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gmail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
MacBook-Pro-de-Odellya:simplegit-progit odellya$
```

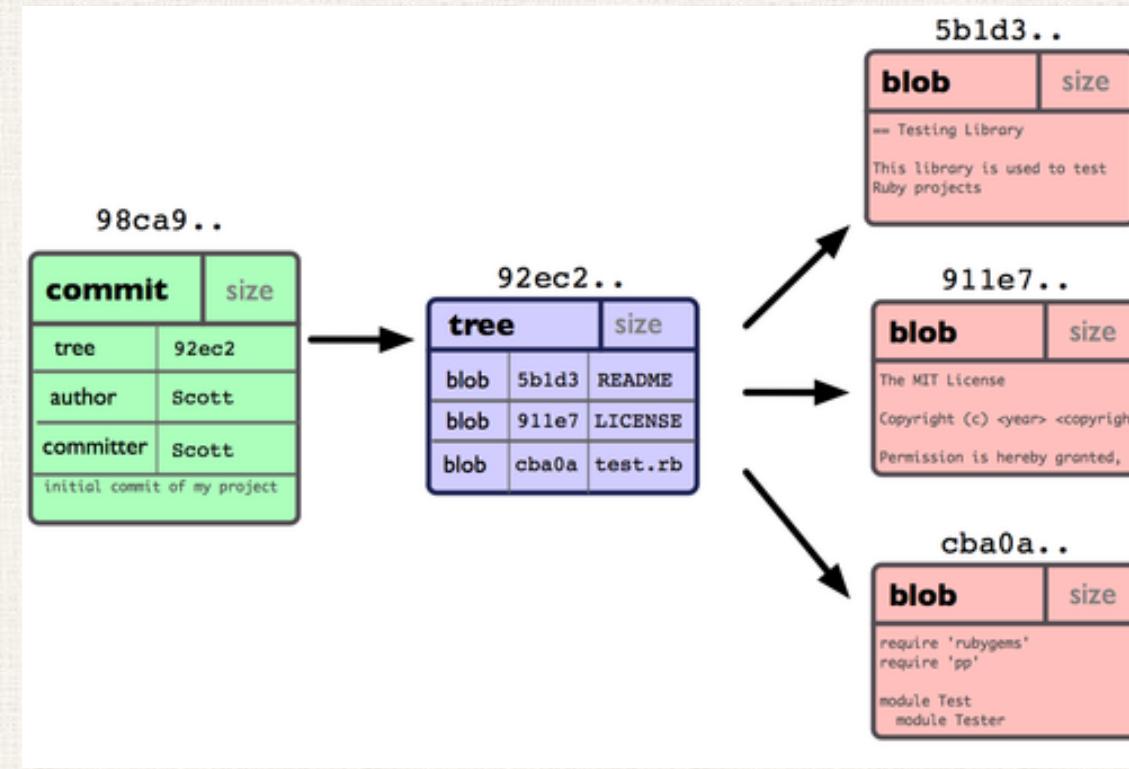
### INFOS

- Par défaut, git log invoqué sans argument énumère en ordre chronologique inversé les commits réalisés.
- Cela signifie que les commits les plus récents apparaissent en premier.
- Comme vous le remarquez, cette commande indique chaque commit avec sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur, la date et le message du commit.
- git log dispose d'un très grand nombre d'options permettant de paramétrier exactement ce que l'on cherche à voir.

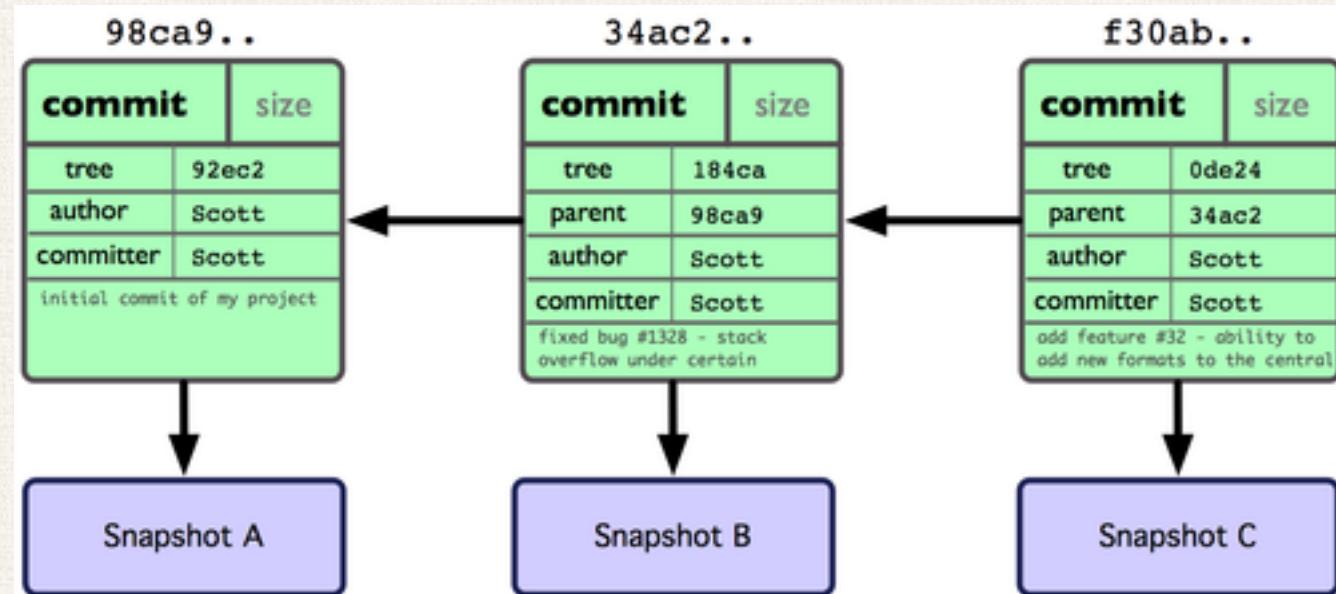


## Ce qu'est une branche

- Git ne stocke pas ses données comme une série de changesets ou deltas, mais comme une série d'instantanés.
- Pour visualiser ce concept, supposons un répertoire contenant trois fichiers, ces trois fichiers étant indexés puis validés.
- Indexer les fichiers signifie calculer la somme de contrôle pour chacun , stocker cette version du fichier dans le dépôt Git (Git les nomme blobs) et ajouter la somme de contrôle à la zone d'index :



- Si vous réalisez des modifications et validez à nouveau, le prochain commit stocke un pointeur vers le commit immédiatement précédent.
- Après deux autres validations, l'historique pourrait ressembler à la figure ci-dessous:



## Ce qu'est une branche

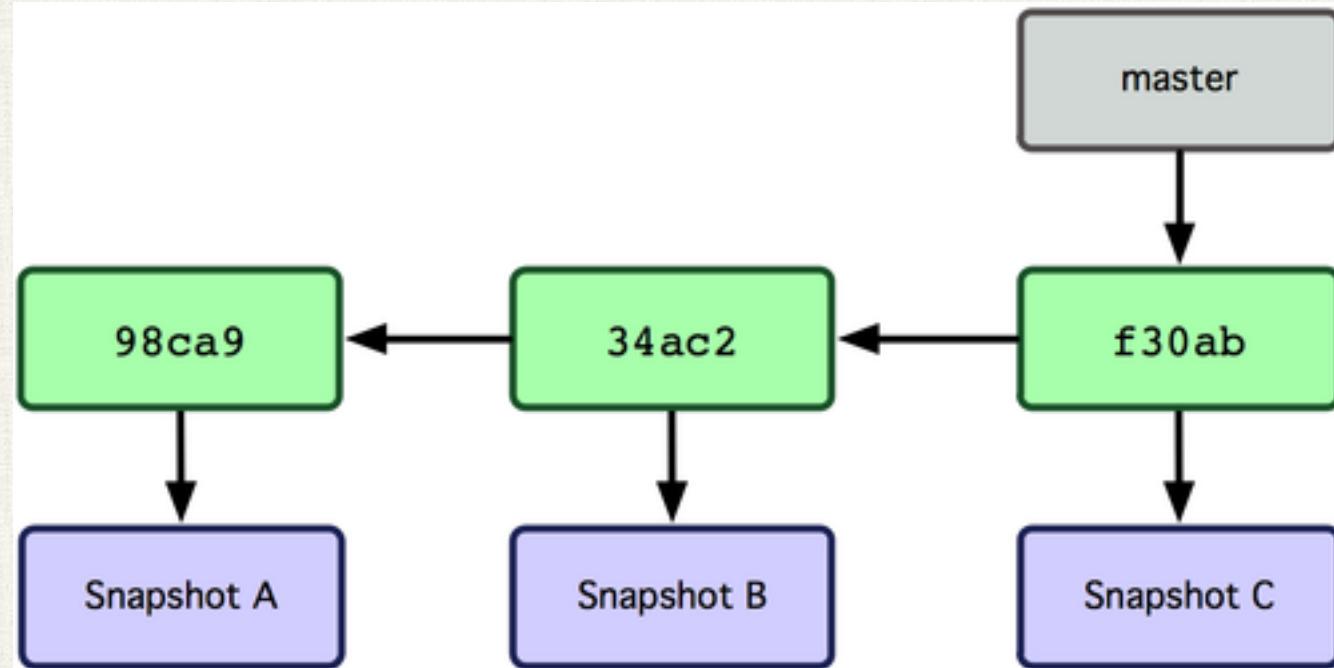
- Une branche dans Git est tout simplement un pointeur mobile léger vers un de ces objets commit.
- La branche par défaut dans Git s'appelle master.
- Au fur et à mesure des validations, la branche master pointe vers le dernier des commits réalisés.
- À chaque validation, le pointeur de la branche master avance automatiquement.

LAB: que se passe-t-il si vous créez une nouvelle branche ? Et bien, cela crée un nouveau pointeur à déplacer.

Supposons que vous créez une nouvelle branche nommée **testing**. Vous utilisez la commande **git branch** :



## Création d'une nouvelle branche

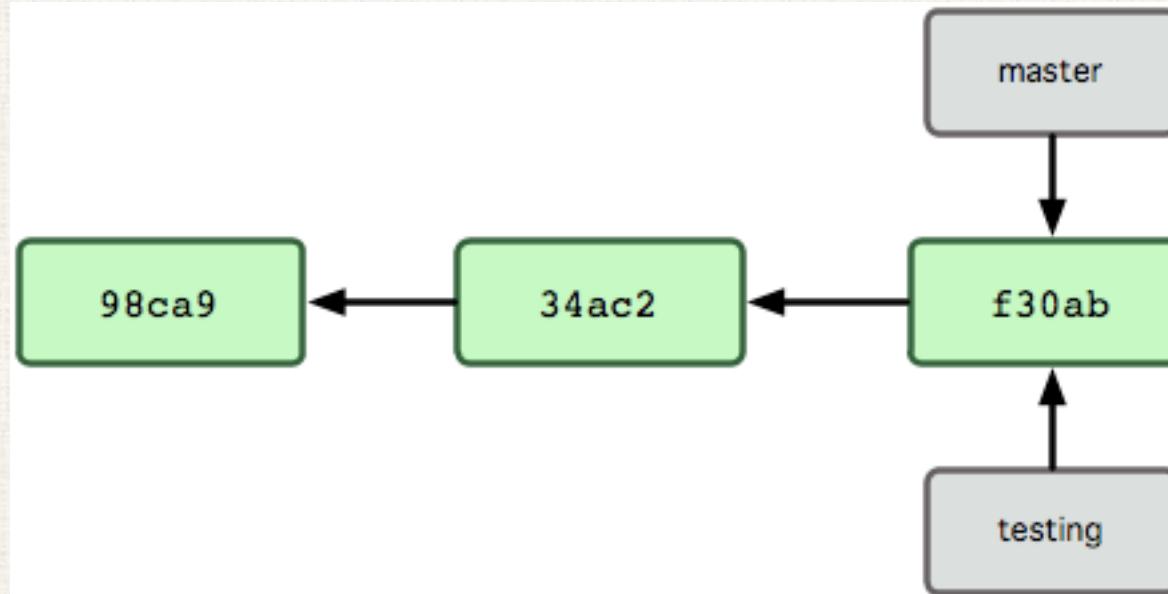


```
$ git branch testing
```

Lancez cette commande à la racine du projet git



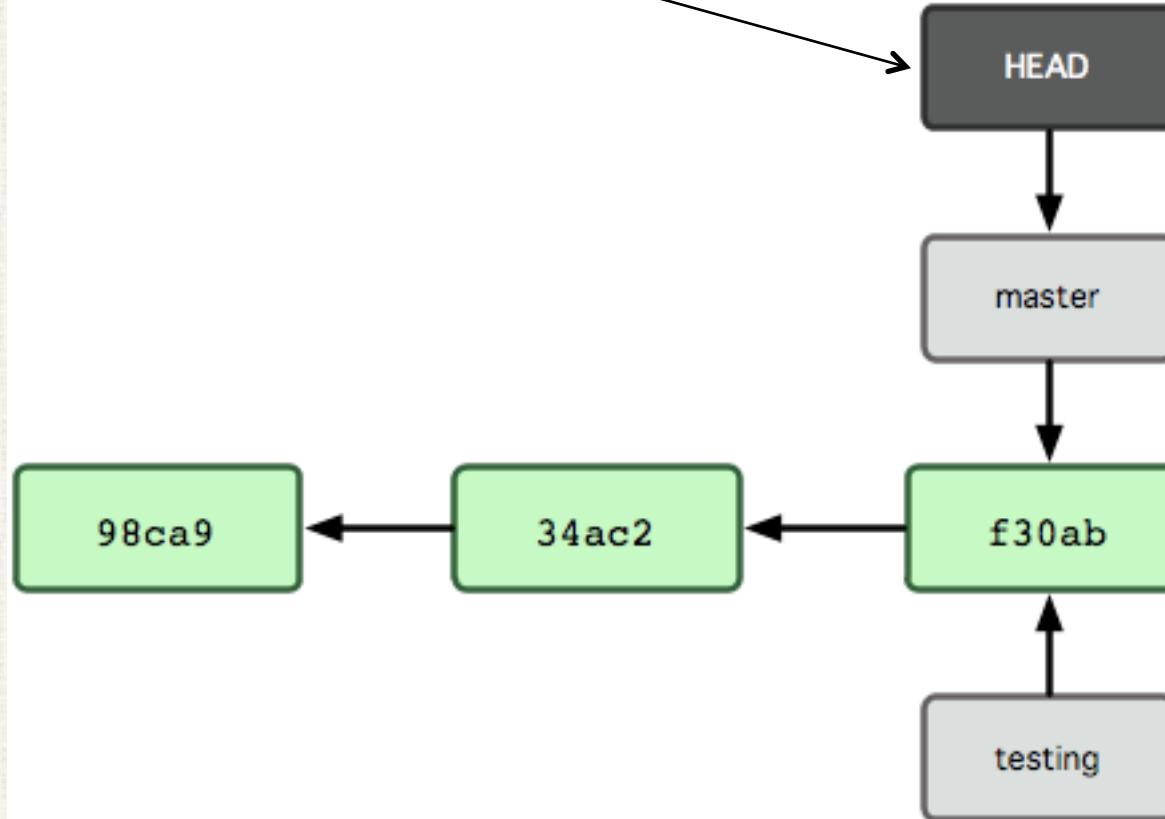
Ceci crée un nouveau pointeur vers le commit actuel



Comment Git connaît-il la branche sur laquelle vous vous trouvez ?



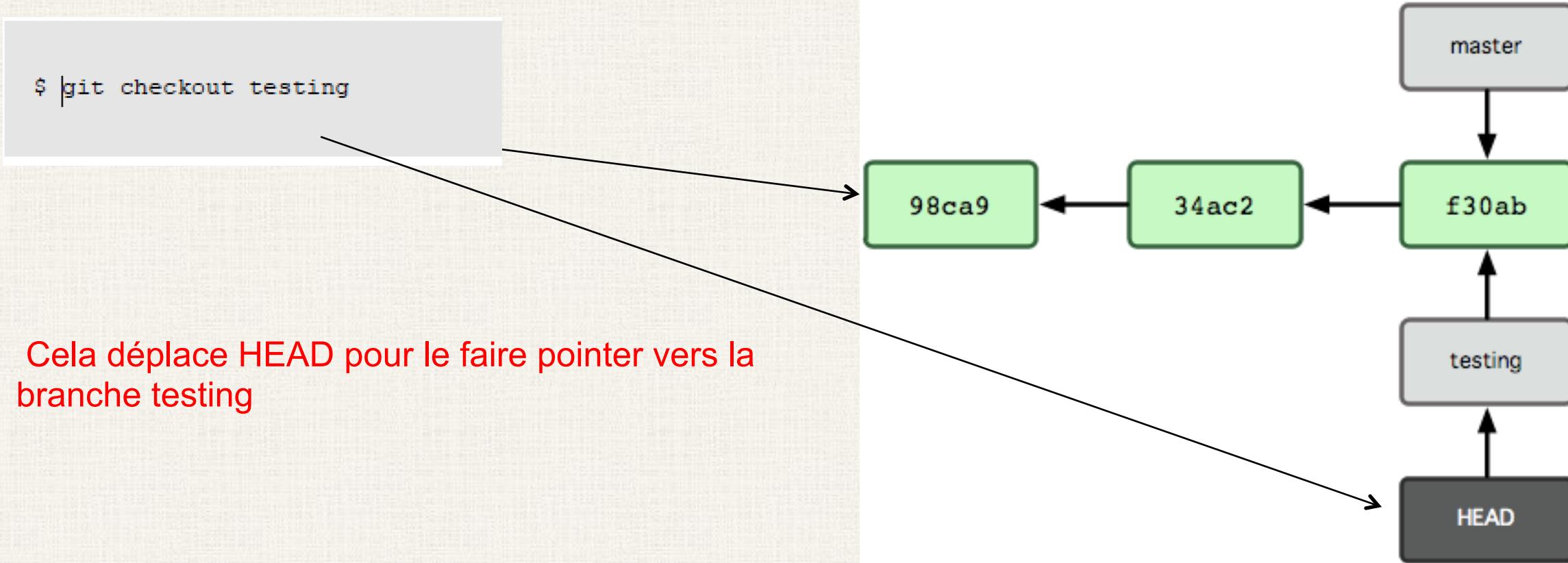
Il conserve un pointeur spécial appelé HEAD



- Remarquez que sous cette appellation se cache un concept très différent de celui utilisé dans les autres VCS tels que Subversion ou CVS.
- Dans Git, c'est un pointeur sur la branche locale où vous vous trouvez.
- Dans notre cas, vous vous trouvez toujours sur master.
- La commande **git branch** n'a fait que créer une nouvelle branche

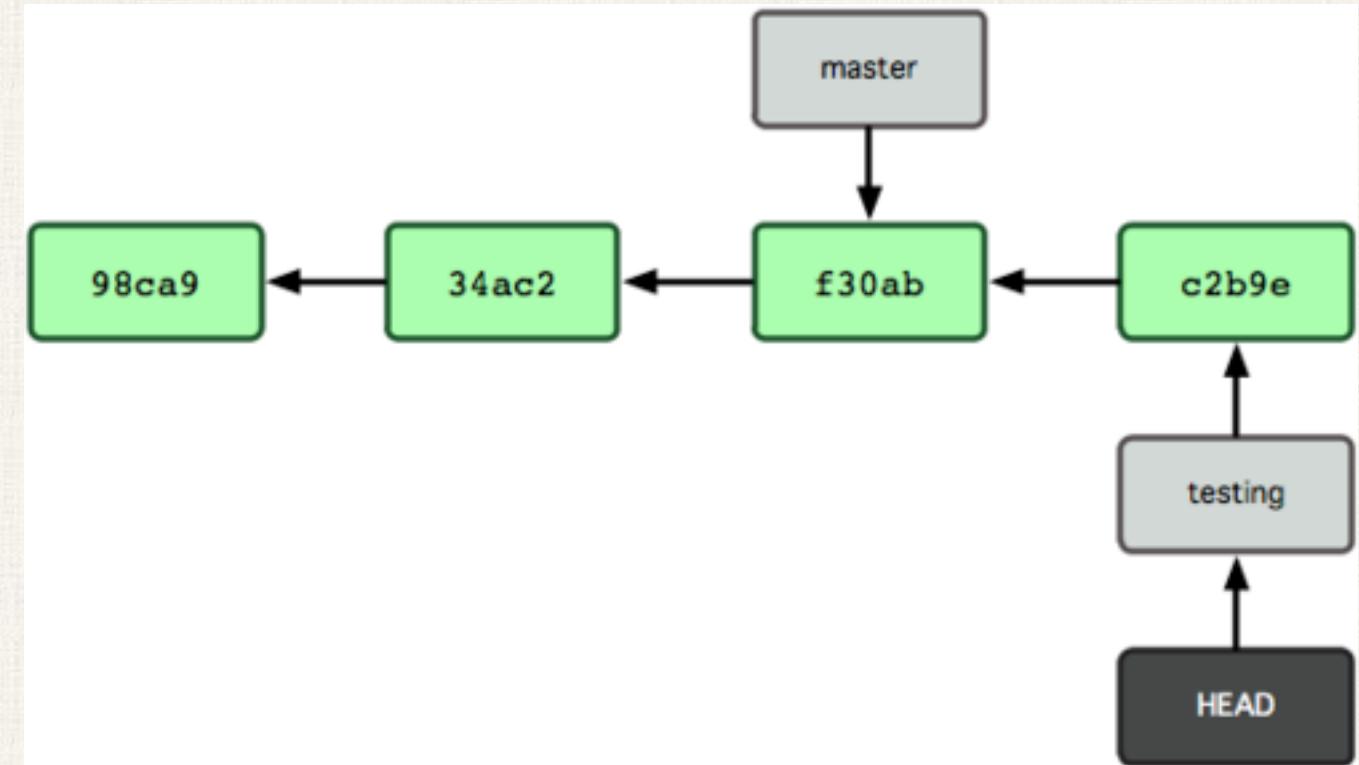


- Pour basculer vers une branche existante, il suffit de lancer la commande **git checkout** .



Modifions puis validons de nouveau un fichier se trouvant dans répertoire

```
$ vim test.rb  
$ git commit -a -m 'petite modification'
```

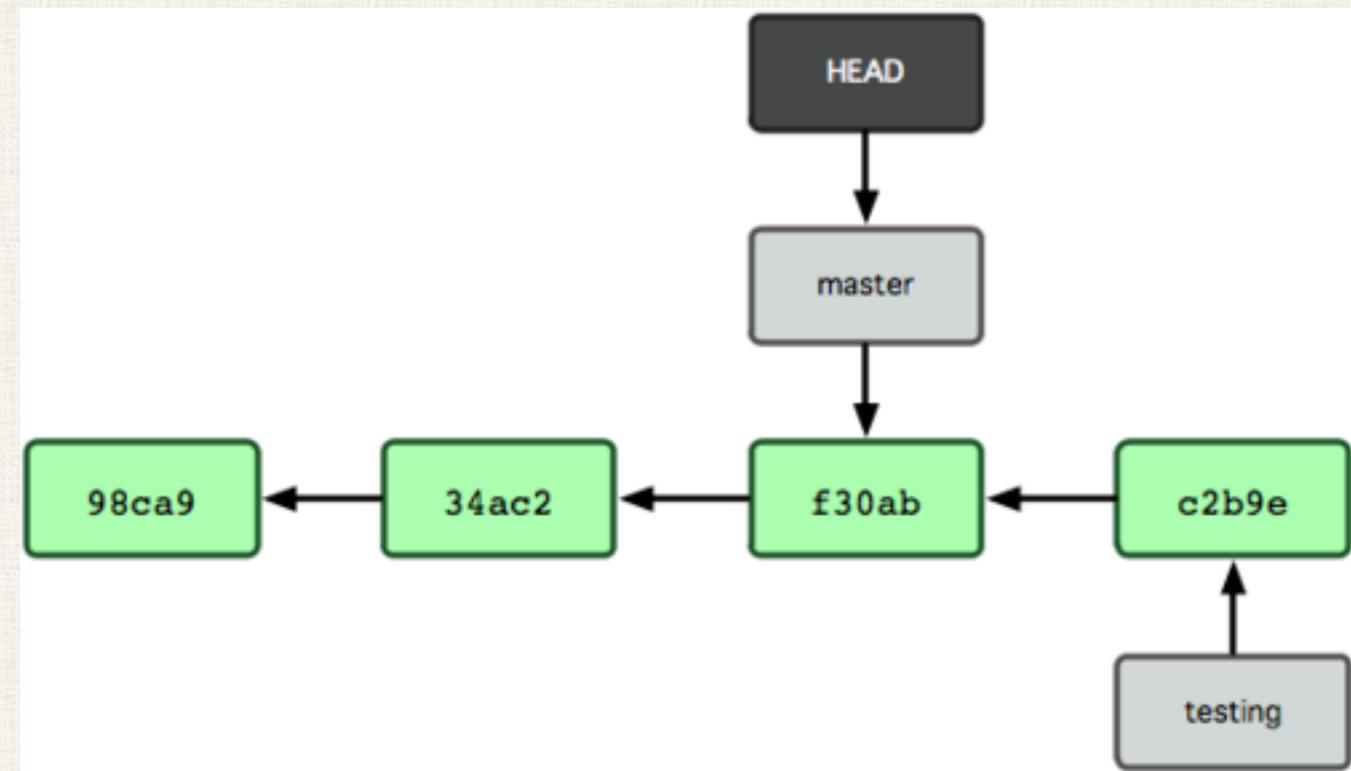


- C'est intéressant parce qu'à présent, votre branche testing a avancé, tandis que la branche master pointe toujours sur le commit sur lequel vous étiez lorsque vous avez lancé **git checkout** pour basculer de branche.

Retournons sur la branche master :

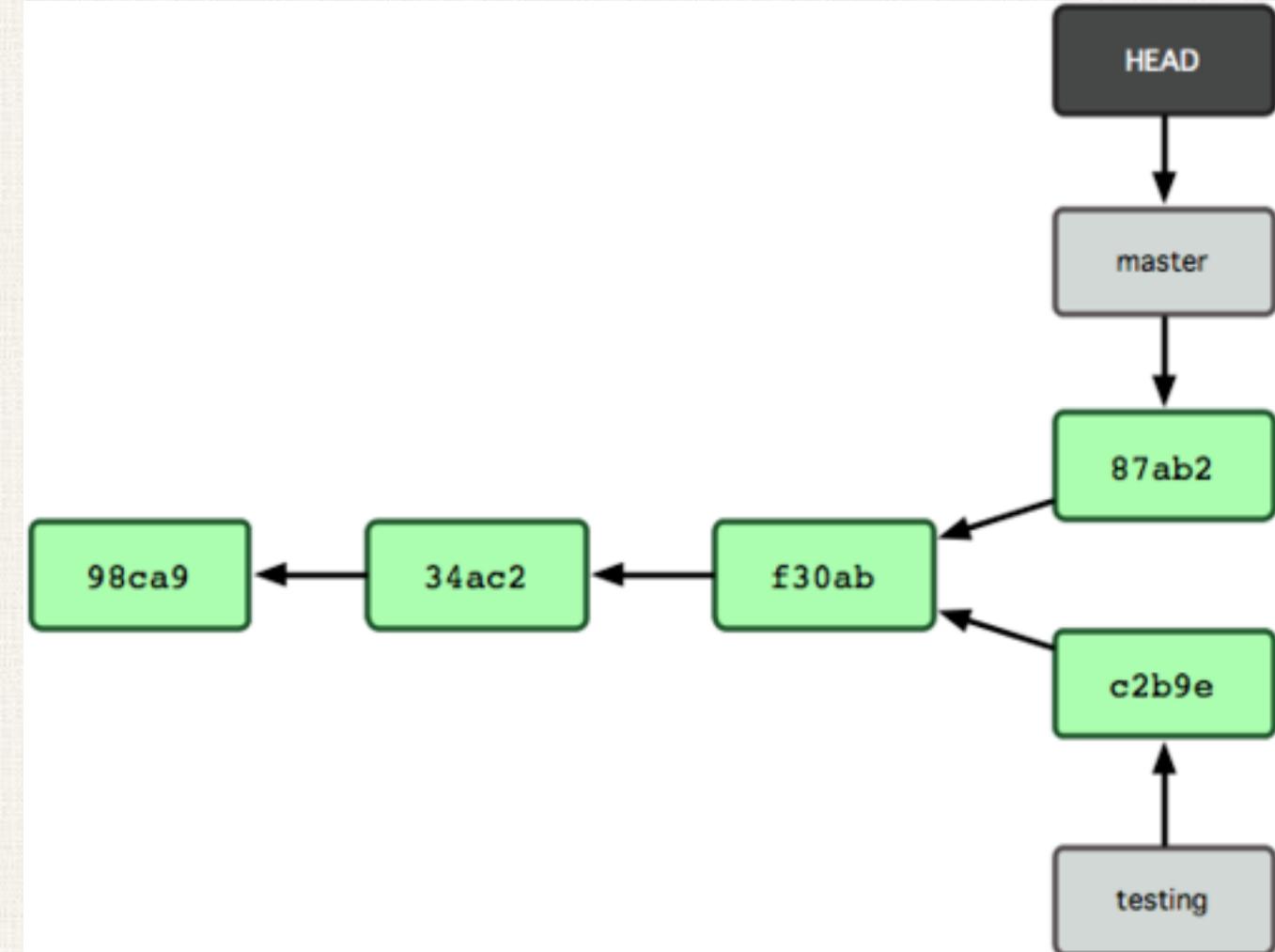
```
$ git checkout master
```

- Cette commande a réalisé deux actions. Elle a remis le pointeur **HEAD** sur la branche **master** et elle a replacé les fichiers de la copie de travail dans l'état pointé par master.
- Cela signifie aussi que les modifications que vous réalisez à partir de maintenant divergeront de l'ancienne version du projet.
- Cette commande retire les modifications réalisées dans la branche testing pour vous permettre de repartir dans une autre direction de développement.



Réalisons quelques autres modifications et validons à nouveau :

```
$ vim test.rb
$ git commit -a -m 'autres modifications'
```



- Maintenant, l'historique du projet a divergé .
- Vous avez créé une branche et basculé dessus, avez réalisé des modifications, puis avez rebasculé sur la branche principale et réalisé d'autres modifications.
- Ces deux modifications sont isolées dans des branches séparées.
- Vous pouvez basculer d'une branche à l'autre et les fusionner quand vous êtes prêt. Vous avez fait tout ceci avec de simples commandes **branch et checkout** .



## Brancher et fusionner : les bases

Suivons un exemple simple de branche et fusion dans une utilisation que vous feriez dans le monde réel. Vous ferez les étapes suivantes :

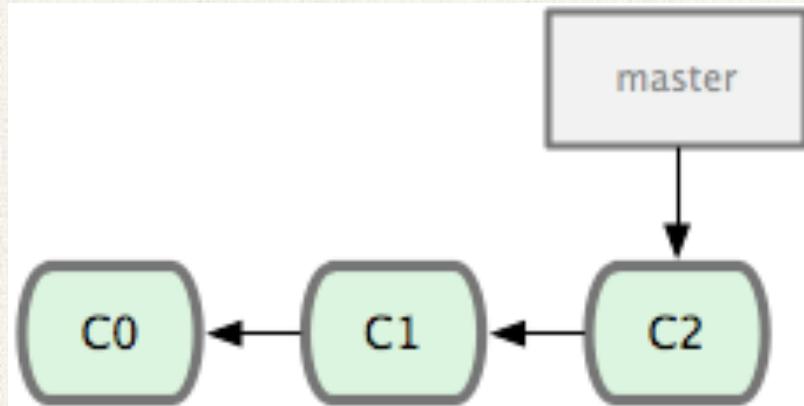
1. Travailler sur un site web
2. Créer une branche pour une nouvelle Story sur laquelle vous souhaiteriez travailler
3. Réaliser quelques tâches sur cette branche

À cette étape, vous recevez un appel pour vous dire qu'un problème critique a été découvert et qu'il faut le régler au plus tôt. Vous ferez ce qui suit :

1. Revenir à la branche de production
2. Créer une branche et y développer le correctif
3. Après un test, fusionner la branche de correctif et pousser le résultat à la production
4. Rebasculer à la branche initiale et continuer le travail



## Le branchement de base



Vous avez décidé de travailler sur le problème numéroté #53 dans le suivi de faits techniques que votre entreprise utilise. Pour clarifier, Git n'est pas lié à un gestionnaire particulier de faits techniques.

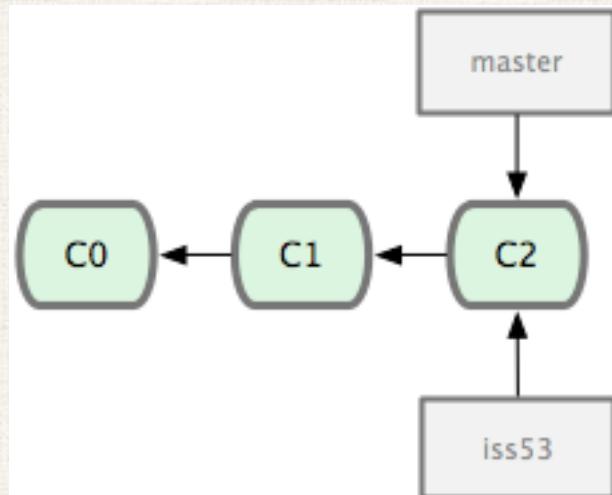
Mais comme le problème #53 est un problème ciblé sur lequel vous voulez travailler, vous allez créer une nouvelle branche dédiée à sa résolution. Pour créer une branche et y basculer tout de suite, vous pouvez lancer la commande **git checkout** avec l'option -b :



```
$ git checkout -b prob53
Switched to a new branch "prob53"
```

C'est un raccourci pour :

```
$ git branch prob53
$ git checkout prob53
```

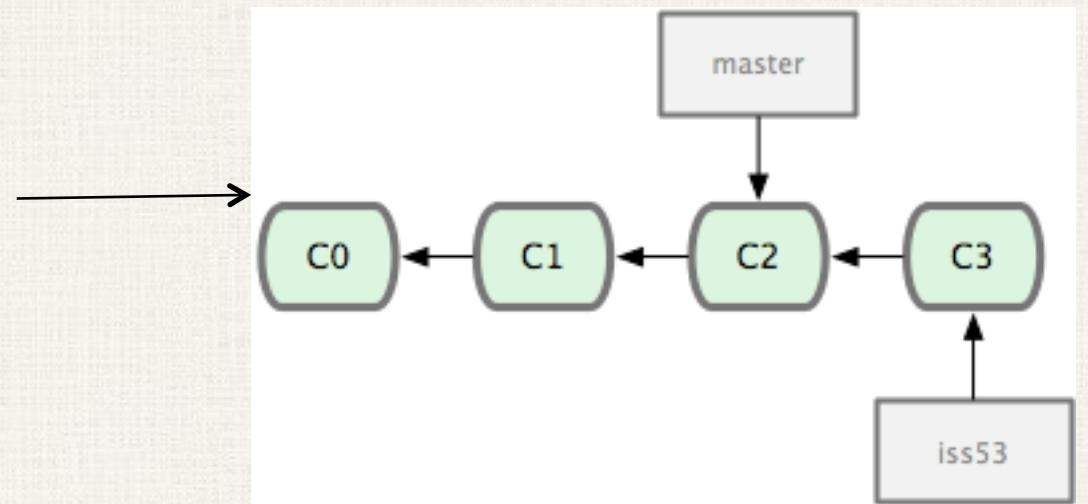


Vous travaillez sur votre site web et validez des modifications.  
Ce faisant, la branche prob53 avance, parce que vous l'avez extraite  
(c'est-à-dire que votre pointeur HEAD pointe dessus)



## La branche prob53 a avancé avec votre travail.

```
$ vim index.html
$ git commit -a -m 'ajout d'un pied de page [problème 53]'
```



## Le branchement de base

### PROBLEMES

- Maintenant vous recevez un appel qui vous apprend qu'il y a un problème sur le site web, un problème qu'il faut résoudre immédiatement.
- Avec Git, vous n'avez pas besoin de déployer les modifications déjà validée pour prob53 avec les correctifs du problème et vous n'avez pas non plus à suer pour éliminer ces modifications avant de pouvoir appliquer les correctifs du problème en production.
- Tout ce que vous avez à faire, c'est simplement rebasculer sur la branche master .
- Cependant, avant de le faire, notez que si votre copie de travail ou votre zone de préparation contient des modifications non validées qui sont en conflit avec la branche que vous extrayez, Git ne vous laissera pas basculer de branche.
- Le mieux est d'avoir votre copie de travail dans un état propre au moment de basculer de branche. Il y a des moyens de contourner ceci (précisément par la planque et l'amendement de commit) dont nous parlerons plus loin.
- Pour l'instant, vous avez validé tous vos changements dans la branche prob53 et vous pouvez donc rebasculer vers la branche master :

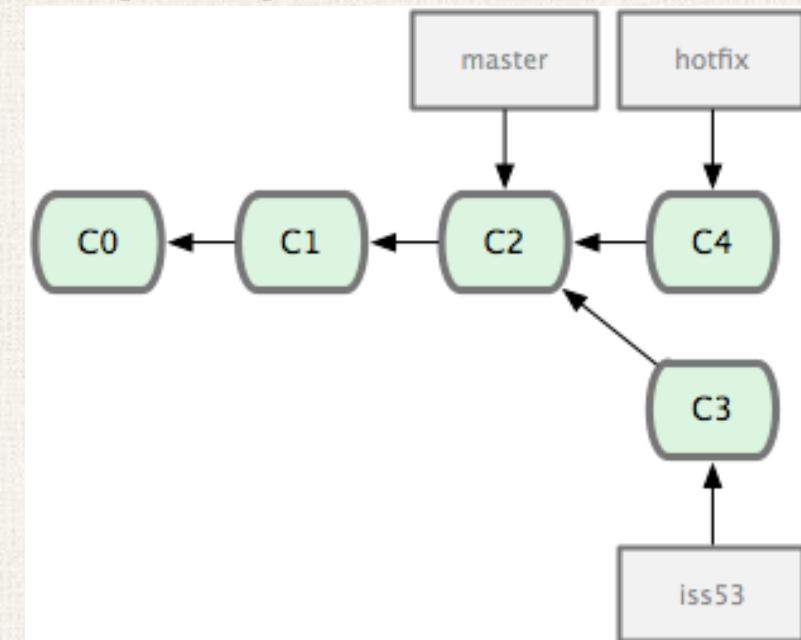


## Le branchement de base

```
$ git checkout master
Switched to branch "master"
```

Ensuite, vous avez un correctif à faire. Créons une branche de correctif sur laquelle travailler jusqu'à ce que ce soit terminé

```
$ git checkout -b 'correctif'
Switched to a new branch "correctif"
$ vim index.html
$ git commit -a -m "correction d'une adresse mail incorrecte"
[correctif]: created 3a0874c: "correction d'une adresse mail incorrecte"
 1 files changed, 0 insertions(+), 1 deletions(-)
```



## Le branchement de base

Vous pouvez lancer vos tests, vous assurer que la correction est efficace et la fusionner dans la branche master pour la déployer en production.

Vous réalisez ceci au moyen de la commande **git merge** :

```
$ git checkout master
$ git merge correctif
Updating f42c576..3a0874c
Fast forward
LISEZMOI |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```



## Les branches distantes

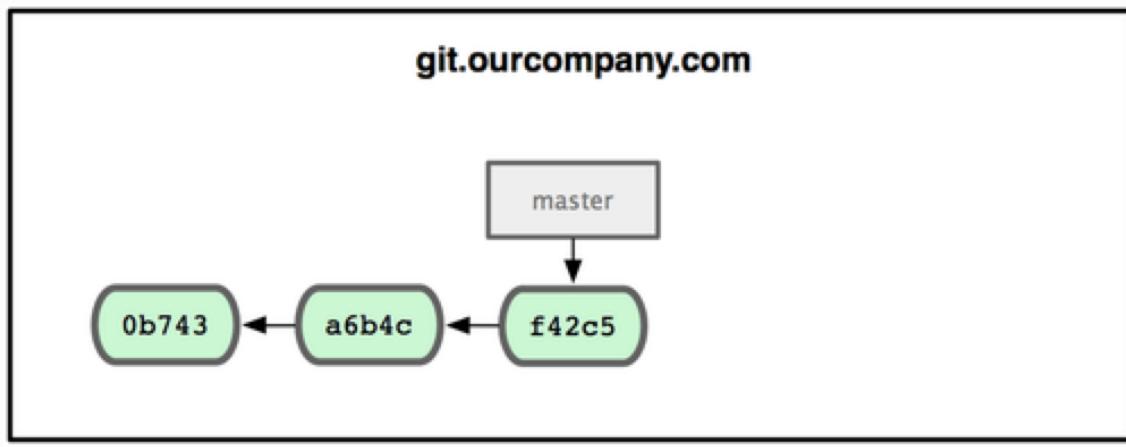
- Les branches distantes sont des références à l'état des branches sur votre dépôt distant.
- Ce sont des branches locales qu'on ne peut pas modifier ; elles sont modifiées automatiquement lors de communications réseau.
- Elles prennent la forme de (**distant**)/(**branche**). L'équivalent de (**origin** / **master**)

## Les branches distantes

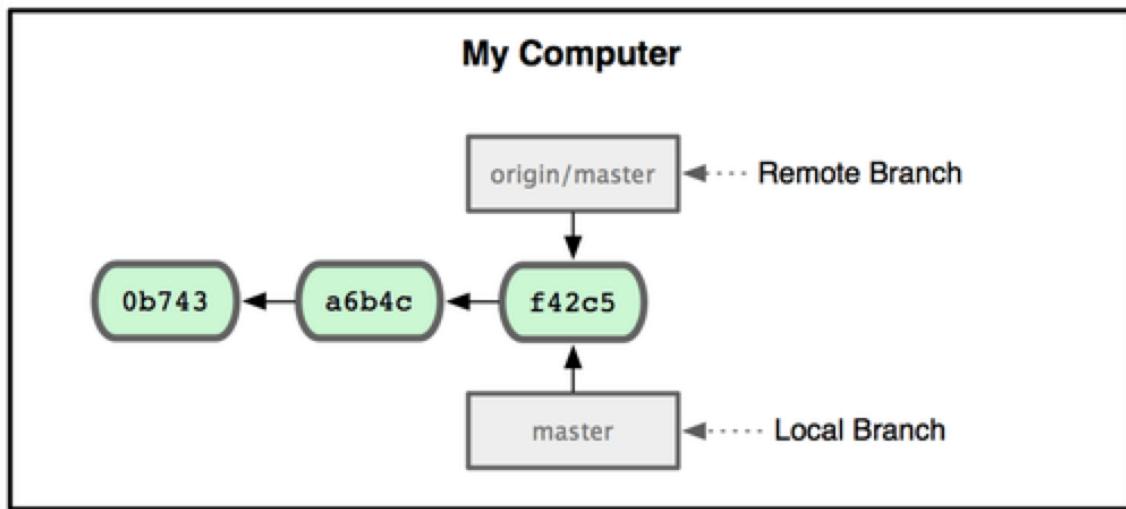
- Supposons que vous avez un serveur Git sur le réseau à l'adresse **git.notresociete.com**.
- Si vous clonez à partir de ce serveur, Git le nomme automatiquement origin et en tire tout l'historique, crée un pointeur sur l'état actuel de la branche master et l'appelle localement **origin/master** ; vous ne pouvez pas la modifier.
- Git vous crée votre propre branche master qui démarre au même commit que la branche master d'origine, pour que vous puissiez commencer à travailler



## Les branches distantes



↓  
`git clone schacon@git.ourcompany.com:project.git`



Lab.

Analysez et expliquez le résultat de la commande **git clone**.

Combien de pointeur Git avez vous ?

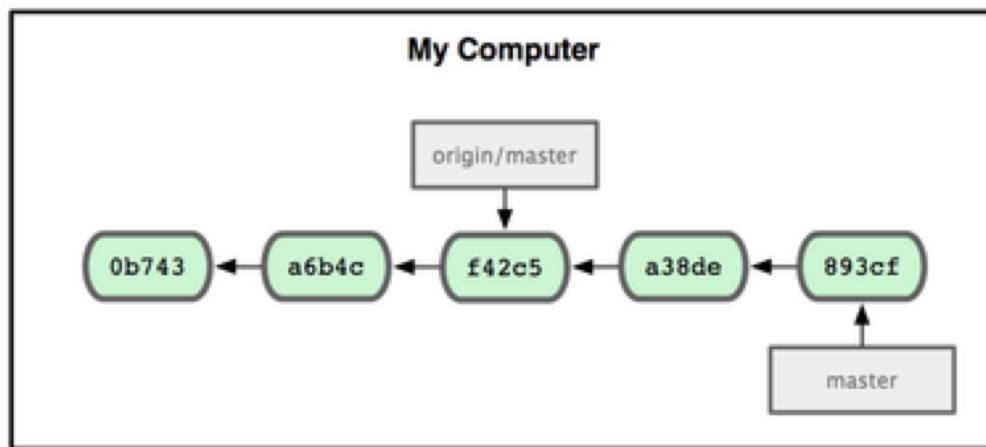
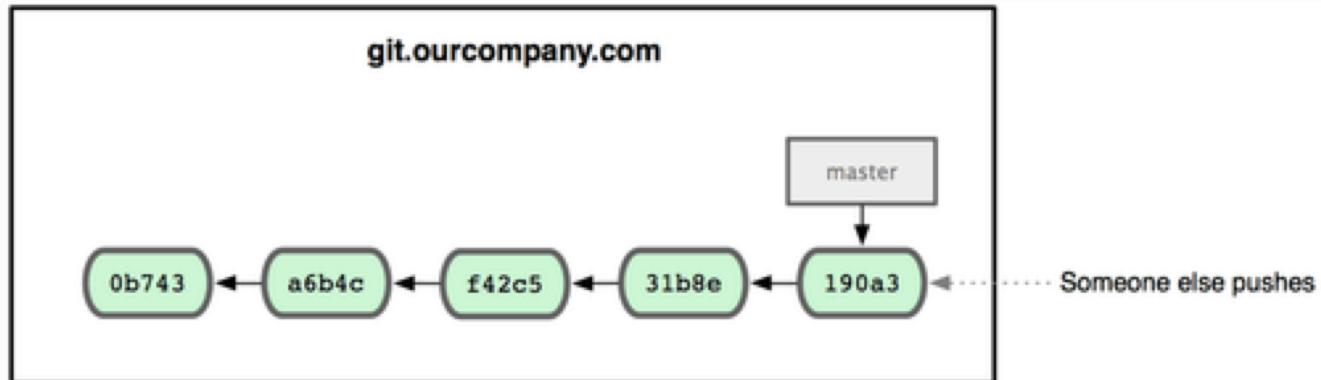
## Les branches distantes

Si vous travaillez sur votre branche locale master et que dans le même temps, quelqu'un pousse vers **git.notresociete.com** et met à jour cette branche, alors vos deux historiques divergent.

Tant que vous restez sans contact avec votre serveur distant, votre pointeur **origin/master** n'avance pas



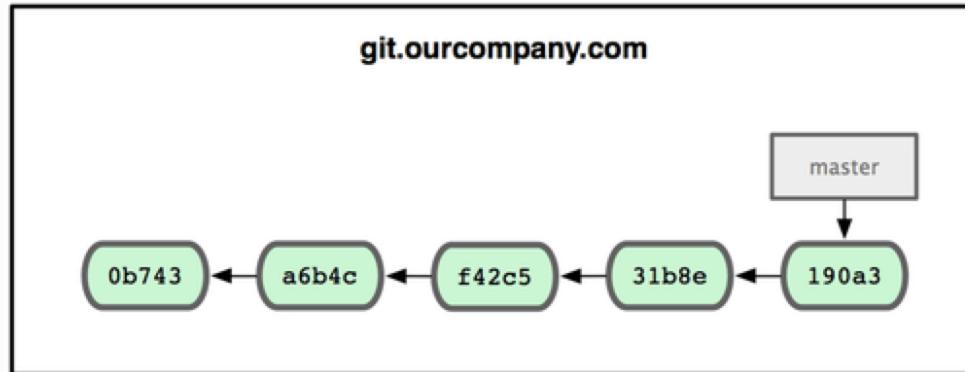
## Les branches distantes



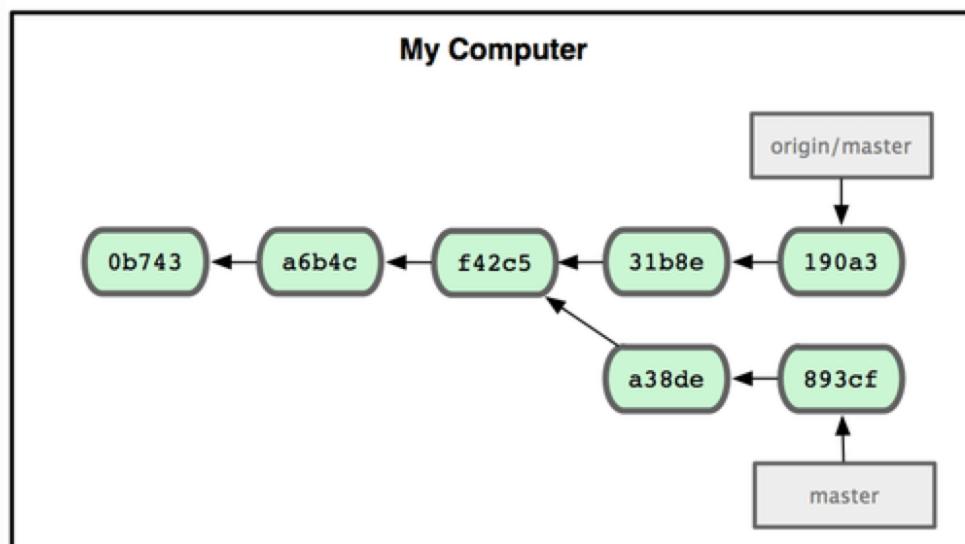
## Les branches distantes

- Lancez la commande **git fetch origin** pour synchroniser votre travail.
- Cette commande recherche le serveur hébergeant origin (dans notre cas, **git.notresociete.com**), en récupère toutes les nouvelles données et met à jour votre base de donnée locale en déplaçant votre pointeur **origin/master** à sa valeur nouvelle à jour avec le serveur distant.

## Les branches distantes



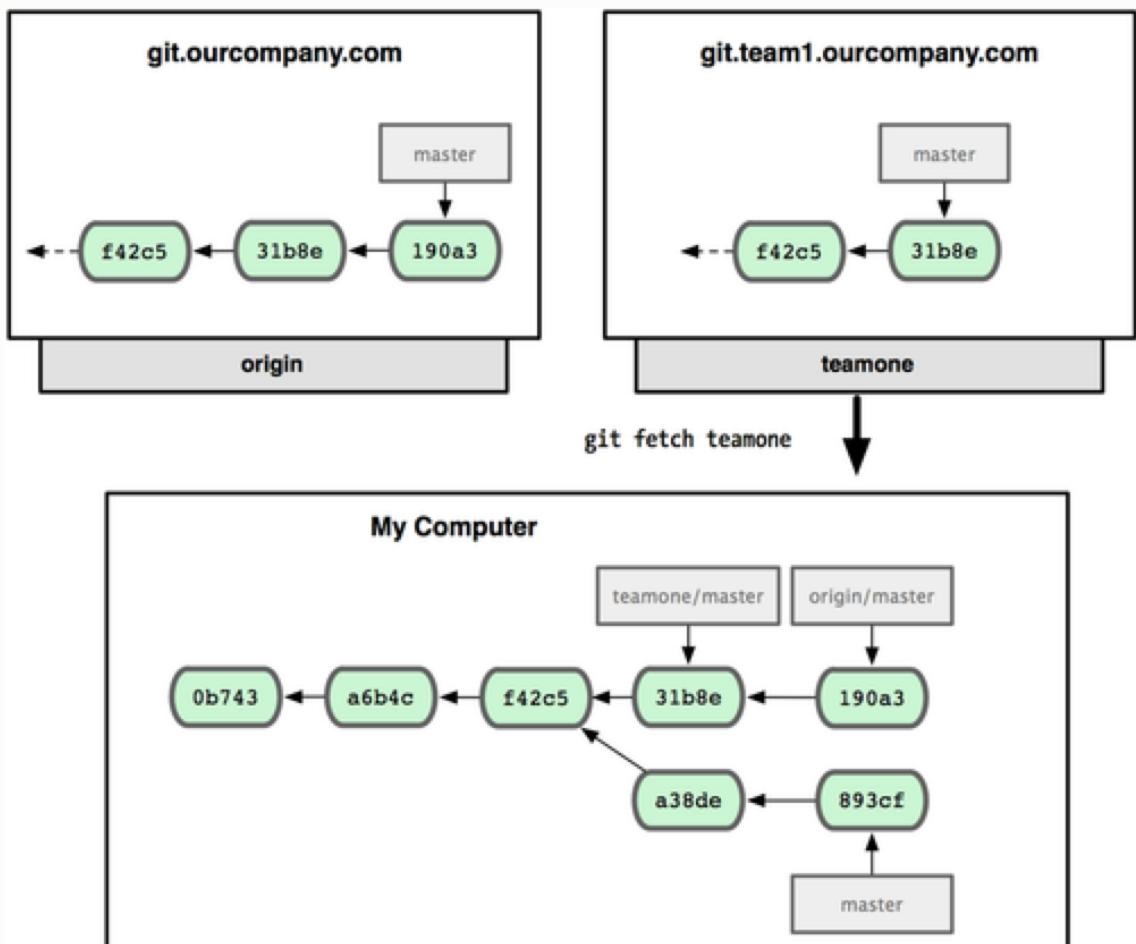
git fetch origin



## Les branches distantes

Maintenant, lancez **git fetch equipeun** pour récupérer l'ensemble des informations du serveur distant **equipeun** que vous ne possédez pas.

Comme ce serveur contient déjà un sous-ensemble des données du serveur origin, Git ne récupère aucune donnée mais positionne une branche distante appelée **equipeun/master** qui pointe sur le commit que **equipeun** a comme **branche master**



## Pousser sur le serveur

- Lorsque vous souhaitez partager une branche avec le reste du monde, vous devez la pousser sur le serveur distant sur lequel vous avez accès en écriture.
- Vos branches locales ne sont pas automatiquement synchronisées sur les serveurs distants —
- vous devez pousser explicitement les branches que vous souhaitez partager.
- De cette manière, vous pouvez utiliser des branches privées pour le travail que vous ne souhaitez pas partager et ne pousser que les branches sur lesquelles vous souhaitez collaborer.
- Si vous possédez une branche nommée **correctionserveur** sur laquelle vous souhaitez travailler avec des tiers, vous pouvez la pousser de la même manière que vous avez poussé votre première branche.



## Pousser sur le serveur

LAB.

Lancez git push [serveur distant] [branche] :

```
$ git push origin correctionserveur
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      correctionserveur -> correctionserveur
```

ATTENTION POSSIBLE ERROR FATAL si vous n'avez pas de compte GitHub



## Pousser sur le serveur

- En fait, Git étend le nom de branche **correctionserveur en refs/heads/correctionserveur:refs/heads/correctionserveur**, ce qui signifie « Prendre ma branche locale correctionserveur et la pousser pour mettre à jour la branche distante correctionserveur ».
- Vous pouvez aussi lancer git push origin **correctionserveur:correctionserveur**, qui réalise la même chose —
- ce qui signifie « Prendre ma branche **correctionserveur** et en faire la branche **correctionserveur distante** ».
- Vous pouvez utiliser ce format pour pousser une branche locale vers une branche distante nommée différemment.
- Si vous ne souhaitez pas l'appeler **correctionserveur** sur le serveur distant, vous pouvez lancer à la place git push origin **correctionserveur:branchegeniale** pour pousser votre branche locale **correctionserveur** sur la branche **branchegeniale** sur le dépôt distant.
- La prochaine fois qu'un de vos collaborateurs récupère les données depuis le serveur, il récupérera une référence à l'état de la branche distante **origin/correctionserveur** :



## Suivre les branches

- L'extraction d'une branche locale à partir d'une branche distante crée automatiquement ce qu'on appelle une branche de suivi.
- Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante. Si vous vous trouvez sur une branche de suivi et que vous tapez git push, Git sélectionne automatiquement le serveur vers lequel pousser vos modifications.
- De même, git pull sur une de ces branches récupère toutes les références distantes et fusionne automatiquement la branche distante correspondante dans la branche actuelle.
- Lorsque vous clonez un dépôt, il crée généralement automatiquement une branche master qui suit origin/master.
- C'est pourquoi les commandes git push et git pull fonctionnent directement sans plus de paramétrage.



## Suivre les branches

- Vous pouvez néanmoins créer d'autres branches de suivi si vous le souhaitez, qui ne suivront pas origin ni la branche master.
- Un cas d'utilisation simple est l'exemple précédent, en lançant **git checkout -b [branche] [nomdistant]/[branche]**.
- Si vous avez Git version 1.6.2 ou plus, vous pouvez aussi utiliser l'option courte **--track** :

```
$ git checkout --track origin/correctionserveur
Branch correctionserveur set up to track remote branch refs/remotes/origin
Switched to a new branch "correctionserveur"
```

- Pour créer une **branche locale avec un nom différent** de celui de la branche distante, vous pouvez simplement utiliser la première version avec un nom de branche locale différent :



## Effacer des branches distantes

- Supposons que vous en avez terminé avec une branche distante.
- Vous et vos collaborateurs avez terminé une fonctionnalité et l'avez fusionnée dans la branche master du serveur distant (ou la branche correspondant à votre code stable).
- Vous pouvez effacer une branche distante en utilisant la syntaxe plutôt **git push [nomdistant] :[branch]**.
- Si vous souhaitez effacer votre branche **correctionserveur** du serveur, vous pouvez lancer ceci :

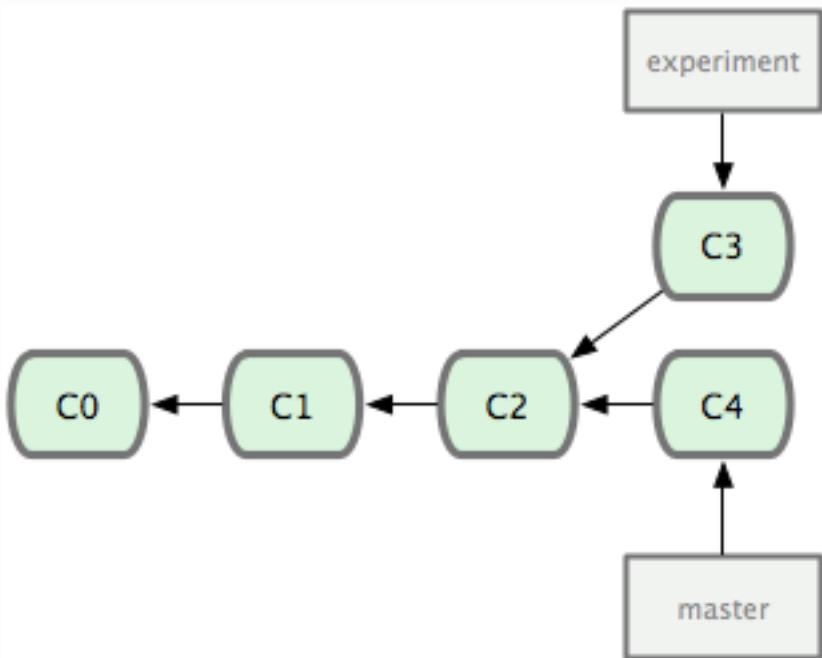
```
$ git push origin :correctionserveur
To git@github.com:schacon/simplegit.git
 - [deleted]           correctionserveur
```



## Les branches avec Git - Rebase

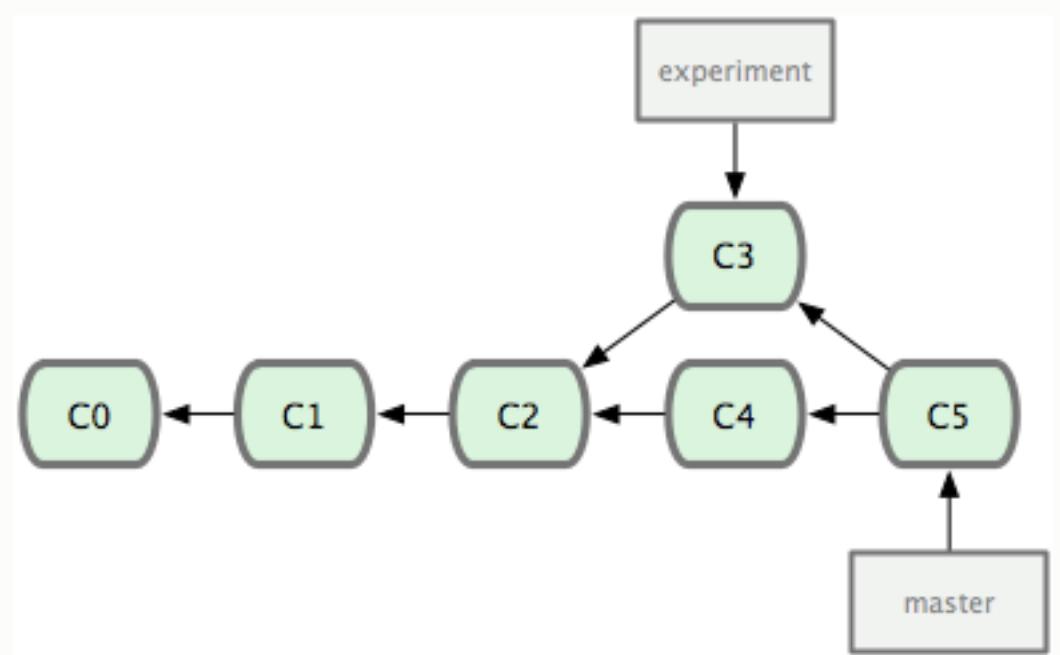
1. Dans Git, il y a deux façons d'intégrer les modifications d'une branche dans une autre : en fusionnant (**merge**) et en rebasant (**rebase**).
2. Il est donc important de comprendre pourquoi : rebaser ?, comment le faire ?, dans quels cas il est déconseillé de l'utiliser?

Partant de ces deux branches qui divergent



## Les branches avec Git - Rebase

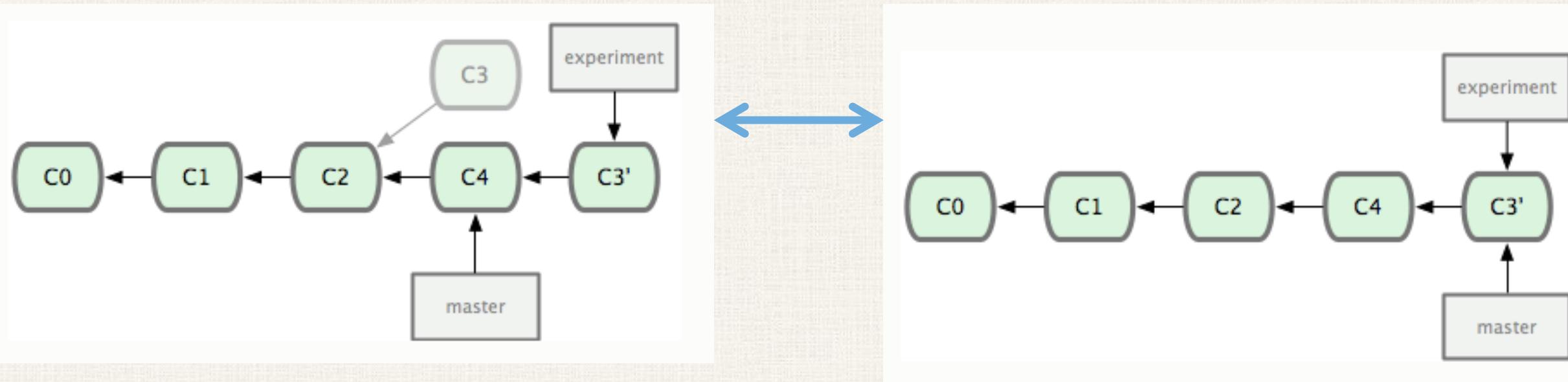
- le moyen le plus simple pour intégrer ensemble ces branches est la fusion via la commande **merge**.
- Cette commande réalise une fusion à trois branches entre les deux derniers instantanés de chaque branche (C3 et C4) et l'ancêtre commun le plus récent (C2), créant un



## Les branches avec Git - Rebase

- Cependant, il existe un autre moyen : vous pouvez prendre le patch de la modification introduite en C3 et le réappliquer sur C4.
- Dans Git, cette action est appelée rebaser.
- Avec la commande rebase, vous prenez toutes les modifications qui ont été validées sur une branche et vous les rejouez sur une autre.

## Les branches avec Git - Rebase



```
$ git rebase --onto master serveur client
```







# DES QUESTIONS ?

À présent, vous devriez être capable de réaliser la plupart des tâches quotidiennes impliquant Git.

Néanmoins, pour pouvoir collaborer avec d'autres personnes au moyen de Git, vous allez devoir disposer d'un dépôt distant Git

- Vous souhaitez que vos collaborateurs puissent accéder à votre dépôt de sources, y compris si vous n'êtes pas connecté disposer d'un dépôt accessible en permanence peut s'avérer utile.
- De ce fait, la méthode pour collaborer consiste à instancier un dépôt intermédiaire auquel tous ont accès, que ce soit pour pousser ou tirer.
- Nous nommerons ce dépôt le « **serveur Git** » mais vous vous apercevrez qu'héberger un serveur de dépôt Git ne consomme que peu de ressources et qu'en conséquence, on n'utilise que rarement une machine dédiée à cette tâche.



## Protocoles

- Git peut utiliser quatre protocoles réseau majeurs pour transporter des données :
  - ✓ local
  - ✓ Secure
  - ✓ Shell (SSH)
  - ✓ Git et HTTP.
- Nous allons voir leur nature et dans quelles circonstances ils peuvent (ou ne peuvent pas) être utilisés.
- Il est à noter que mis à part HTTP, tous le protocoles nécessitent l'installation de Git sur le serveur.



## Protocole Locale

- Le protocole de base est le protocole local pour lequel le dépôt distant est un autre répertoire dans le système de fichier.
- Il est souvent utilisé si tous les membres de l'équipe ont accès à un répertoire partagé.

Exemple: pour cloner un dépôt local, vous pouvez lancer ceci :

```
$ git clone /opt/git/projet.git
```

OU

```
$ git clone file:///opt/git/projet.git
```

- Pour ajouter un dépôt local à un projet Git existant, lancez ceci :

```
$ git remote add proj_local /opt/git/projet.git
```



## Protocole SSH

- Le protocole SSH, est probablement le protocole de transport de Git le plus utilisé.
- Cela est dû au fait que l'accès SSH est déjà en place à de nombreux endroits et que si ce n'est pas le cas, cela reste très facile à faire.
- Cela est aussi dû au fait que **SSH est le seul protocole permettant facilement de lire et d'écrire à distance**.



## Protocole SSH

- Les deux autres protocoles réseau (**HTTP et Git**) sont généralement en lecture seule et s'ils peuvent être utiles pour la publication, le protocole **SSH est nécessaire pour les mises à jour de par ce qu'il permet l'écriture.**
- SSH est un protocole authentifié suffisamment répandu et sa mise oeuvre est simplifiée.

Pour cloner une dépôt Git à travers SSH, spécifiez le préfixe **ssh://** dans l'URL comme ceci :

```
$ git clone ssh://utilisateur@serveur:projet.git
```

ou ne spécifiez pas de protocole du tout — Git choisit SSH par défaut si vous n'êtes pas explicite :

```
$ git clone utilisateur@serveur:projet.git
```

## Protocole GIT

- Pour qu'un dépôt soit publié via le protocole Git, le fichier git-export-daemonok doit exister mais mise à part cette condition sans laquelle le daemon refuse de publier un projet, il n'y a aucune sécurité.
  
- Soit le dépôt Git est disponible sans restriction en lecture, soit il n'est pas publié.
  
- Cela signifie qu'il ne permet de pousser des modifications. Vous pouvez activer la capacité à pousser mais étant donné l'absence d'authentification, n'importe qui sur internet peut pousser sur le dépôt.

Autant dire que ce mode est rarement recherché.



## Protocole HTTP

- Enfin, le protocole HTTP ou HTTPS tient dans la simplicité à le mettre en place.
- Tout ce qu'il y a à faire, c'est de simplement copier un dépôt Git nu sous votre racine de document HTTP et de paramétriser un crochet post-update et c'est prêt.
- À partir de ceci, toute personne possédant un accès au serveur web sur lequel vous avez copié votre dépôt peut le cloner.

Pour autoriser un accès en lecture à votre dépôt sur HTTP, faites ceci :

```
$ cd /var/www/htdocs/  
$ git clone --bare /chemin/vers/git_projet projetgit.git  
$ cd projetgit.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```



## Installation de Git sur un serveur

- Pour réaliser l'installation initiale d'un serveur Git, il faut exporter un dépôt existant dans un nouveau dépôt nu, un dépôt qui ne contient pas de copie de répertoire de travail.
- C'est généralement simple à faire.
- Pour cloner votre dépôt en créant un nouveau dépôt nu, lancez la commande clone avec l'option **--bare** .
- Par convention, les répertoires de dépôt nu finissent en `.git` , de cette manière :

```
$ git clone --bare mon_project mon_project.git
Initialized empty Git repository in /opt/projets/mon_project.git/
```



## Installation de Git sur un serveur

- La sortie de cette commande est un peu déroutante.
- Comme clone est un **git init** de base, suivi d'un **git fetch** , nous voyons les messages du **git init** qui crée un répertoire vide.
- Le transfert effectif d'objet ne fournit aucune sortie, mais il a tout de même lieu.
- Vous devriez maintenant avoir une copie des données de Git dans votre répertoire `mon_project.git` .
- C'est grossièrement équivalent à

```
$ cp -Rf mon_project/.git mon_project.git
```



## Copie du dépôt nu sur un serveur

- À présent que vous avez une copie nue de votre dépôt, il ne reste plus qu'à la placer sur un serveur et à régler les protocoles.
- Supposons que vous avez mis en place un serveur nommé **git.exemple.com** auquel vous avez accès par SSH et que vous souhaitez stocker vos dépôts Git dans le répertoire /opt/git .
- Vous pouvez mettre en place votre dépôt en copiant le dépôt nu :

```
$ scp -r mon_projet.git utilisateur@git.exemple.com:/opt/git
```

A partir de maintenant, tous les autres utilisateurs disposant d'un accès SSH au serveur et ayant un accès en lecture seule au répertoire /opt/git peuvent cloner votre dépôt en lançant la commande

```
$ git clone utilisateur@git.exemple.com:/opt/git/mon_projet.git
```



## Copie du dépôt nu sur un serveur

- Si un utilisateur se connecte par SSH au serveur et a accès en lecture au répertoire /opt/git/mon\_projet.git , il aura automatiquement accès pour tirer.
- Git ajoutera automatiquement les droits de groupe en écriture à un dépôt si vous lancez la commande **git init** avec l'option **-- shared** .

```
$ ssh utilisateur@git.exemple.com
$ cd /opt/git/mon_projet.git
$ git init --bare --shared
```

- Vous voyez comme il est simple de prendre un dépôt Git, créer une version nue et la placer sur un serveur auquel vous et vos collaborateurs avez accès en SSH.
- Vous voilà prêts à collaborer sur le même projet.



## Petites installations

- Si vous travaillez dans un petit groupe ou si vous n'êtes qu'en phase d'essai de Git au sein de votre société avec peu de développeurs, les choses peuvent rester simples.
- Un des aspects les plus compliqués de la mise en place d'un serveur Git est la gestion des utilisateurs.
- Si vous souhaitez que certains dépôts ne soient accessibles à certains utilisateurs qu'en lecture seule et en lecture/écriture pour d'autres, la gestion des accès et des permissions peut devenir difficile à régler.

















## Génération des clefs publiques SSH

- Cela dit, de nombreux serveurs Git utilisent une authentification par clefs publiques SSH.
- Pour fournir une clef publique, chaque utilisateur de votre système doit la générer s'il n'en a pas déjà.
- Le processus est similaire sur tous les systèmes d'exploitation.
- Premièrement, l'utilisateur doit vérifier qu'il n'en a pas déjà une. Par défaut, les clefs SSH d'un utilisateur sont stockées dans le répertoire **~/.ssh** du compte.
- Vous pouvez facilement vérifier si vous avez déjà une clef en listant le contenu de ce répertoire :

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa      known_hosts
config           id_dsa.pub
```



## Génération des clefs publiques SSH

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```



## Génération des clefs publiques SSH

- Premièrement, le programme demande confirmation pour l'endroit où vous souhaitez sauvegarder la clef (**.ssh/id\_rsa**) puis il demande deux fois d'entrer un mot de passe qui peut être laissé vide si vous ne souhaitez pas devoir taper un mot de passe quand vous utilisez la clef.
- Maintenant, chaque utilisateur ayant suivi ces indications doit envoyer la clef publique à la personne en charge de l'administration du serveur Git (en supposant que vous utilisez un serveur SSH réglé pour l'utilisation de clefs publiques).
- Ils doivent copier le contenu du fichier **.pub** et l'envoyer par e-mail. Les clefs publiques ressemblent à ceci :

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPL+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7x1ELEVf4h91FX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiXlnKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprrx88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTP189ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2sol01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```



## Génération des clefs publiques SSH

Pour un tutoriel plus approfondi sur la création de clef SSH sur différents systèmes d'exploitation, référez-vous au guide GitHub sur les clefs SSH à

<http://github.com/guides/providing-your-ssh-key> .

## Mise en place du serveur

- Parcourons les étapes de la mise en place d'un accès SSH côté serveur. Dans cet exemple, vous utiliserez la méthode des **authorized\_keys** pour authentifier vos utilisateurs.
- Nous supposerons également que vous utilisez une distribution Linux standard telle qu'Ubuntu.
- Premièrement, créez un utilisateur 'git' et un répertoire **.ssh** pour cet utilisateur.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```



## Mise en place du serveur

- Ensuite, vous devez ajouter la clef publique d'un développeur au fichier **authorized\_keys** de l'utilisateur git.
- Supposons que vous avez reçu quelques clefs par e-mail et les avez sauvées dans des fichiers temporaires. Pour rappel, une clef publique ressemble à ceci :

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpgW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
o7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

- Il suffit de les ajouter au fichier `authorized_keys` :



## Mise en place du serveur

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys  
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys  
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Maintenant, vous pouvez créer un dépôt vide nu en lançant la commande **git init** avec l'option **--bare** , ce qui initialise un dépôt sans répertoire de travail :

```
$ cd /opt/git  
$ mkdir projet.git  
$ cd projet.git  
$ git --bare init
```



## Mise en place du serveur

- Alors, John, Josie ou Jessica peuvent pousser la première version de leur projet vers ce dépôt en l'ajoutant en tant que dépôt distant et en lui poussant une branche.
- Notons que quelqu'un doit se connecter au serveur et créer un dépôt nu pour chaque ajout de projet. Supposons que le nom du serveur soit **gitserveur** .
- Si vous l'hébergez en interne et avez réglé le DNS pour faire pointer **gitserver** sur ce serveur, alors vous pouvez utiliser les commandes suivantes telle quelle :

```
# Sur l'ordinateur de John
$ cd monproject
$ git init
$ git add .
$ git commit -m 'premiere validation'
$ git remote add origin git@gitserveur:/opt/git/projet.git
$ git push origin master
```

À présent, les autres utilisateurs peuvent cloner le dépôt et y pousser leurs modifications aussi simplement :



## Mise en place du serveur

```
$ git clone git@gitserveur:/opt/git/projet.git
$ vim LISEZMOI
$ git commit -am 'correction fichier LISEZMOI'
$ git push origin master
```

- De cette manière, vous pouvez rapidement mettre en place un serveur Git en lecture/écriture pour une poignée de développeurs.
- En précaution supplémentaire, vous pouvez simplement restreindre l'utilisateur '**git**' à des actions Git avec un **shell** limité appelé **git-shell** qui est fourni avec Git.
- Si vous positionnez ce **shell** comme **shell** de login de l'utilisateur '**git**', l'utilisateur **git** ne peut pas avoir de **shell** normal sur ce serveur.
- Pour utiliser cette fonction, spécifiez **git-shell** en lieu et place de **bash** ou **csh** pour **shell** de l'utilisateur. Cela se réalise généralement en éditant le fichier **/etc/passwd** :



## Accès Public

- Et si vous voulez permettre des accès anonymes en lecture ? Peut-être souhaitez-vous héberger un projet open source au lieu d'un projet interne privé.
- Ou peut-être avez-vous quelques serveurs de compilation ou d'intégration continue qui changent souvent et vous ne souhaitez pas avoir à régénérer des clefs SSH tout le temps — vous avez besoin d'un accès en lecture seule simple.
- Le moyen le plus simple pour des petites installations est probablement d'installer un serveur web statique dont la racine pointe sur vos dépôts Git puis d'activer **le crochet post-update**.

## GitWeb

- Après avoir réglé les accès de base en lecture/écriture et en lecture seule pour vos projets, vous souhaiterez peut-être mettre en place une interface web simple de visualisation.
- Git fournit un script CGI appelé **GitWeb** qui est souvent utilisé à cette fin.
- Vous pouvez voir **GitWeb** en action sur des sites tels que <http://git.kernel.org>

/pub/scm /git/git.git / summary

summary | shortlog | log | commit | commitdiff | tree

commit search:

description The core git plumbing  
owner Junio C Hamano  
last change Fri, 20 Feb 2009 07:44:07 +0000  
URL git://git.kernel.org/pub/scm/git/git.git  
http://www.kernel.org/pub/scm/git/git.git

shortlog

Time Ago	Author	Commit Message	Actions
33 hours ago	Junio C Hamano	Merge branch 'maint' master	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
34 hours ago	Matthieu Moy	More friendly message when locking the index fails. <a href="#">maint</a>	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
34 hours ago	Matthieu Moy	Document git blame --reverse.	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
34 hours ago	Marcel M. Cary	gitweb: Hyperlink multiple git hashes on the same commi ...	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
34 hours ago	Johannes Schindelin	system_path(): simplify using strip_path_suffix(), ...	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
34 hours ago	Johannes Schindelin	Introduce the function strip_path_suffix()	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Todd Zullinger	Documentation: Note file formats send-email accepts	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Junio C Hamano	Merge branch 'maint'	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Junio C Hamano	tests: fix "export var=val"	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Lars Noschinski	filter-branch -d: Export GIT_DIR earlier	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Jay Soffian	Disallow providing multiple upstream branches to rebase ...	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Michael Spang	Skip timestamp differences for diff --no-index	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Junio C Hamano	git-svn: fix parsing of timestamp obtained from svn	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Marcel M. Cary	gitweb: Fix warnings with override permitted but no ...	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Gerrit Pape	Documentation/git-push: --all, --mirror, --tags can ...	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
2 days ago	Thomas Rast	bash completion: only show 'log --merge' if merging	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>



## GitWeb

- Si vous souhaitez vérifier à quoi GitWeb ressemblerait pour votre projet, Git fournit une commande pour démarrer une instance temporaire de serveur si vous avez un serveur léger tel que lighttpd ou webrick sur votre système.
- Sur les machines Linux, **lighttpd** est souvent préinstallé et vous devriez pouvoir le démarrer en tapant **git instaweb** dans votre répertoire de travail.
- Si vous utilisez un Mac, Ruby est installé de base avec Léopard, donc **webrick** est une meilleure option.
- Pour démarrer **instaweb** avec un gestionnaire autre que lighttpd, vous pouvez le lancer avec l'option **--httpd** .

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```



## Git Hébergé

- Si vous ne voulez pas investir dans la mise en place de votre propre serveur Git, il reste quelques options pour héberger vos projets Git chez un site externe dédié à l'hébergement.
- Aujourd'hui, vous avez à disposition un nombre impressionnant d'options d'hébergement, chacune avec différents avantages et désavantages.

Pour une liste à jour, référez-vous à la page GitHosting du wiki principal sur Git :

<http://git.or.cz/gitwiki/GitHosting>



## GitHub: Création d'un compte utilisateur

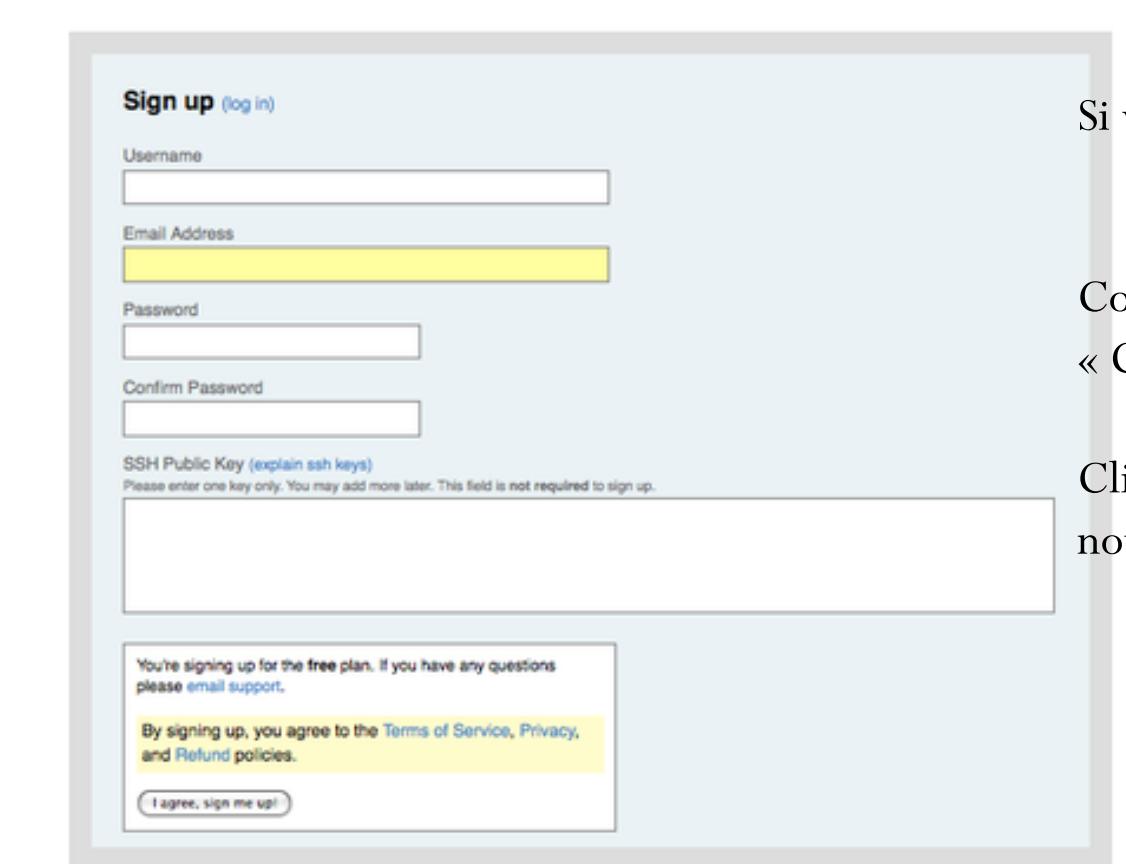
- La première chose à faire, c'est de créer un compte utilisateur gratuit.
- Visitez la page « Prix et inscription » à <http://github.com/plans> et cliquez sur le bouton « Crée un compte gratuit » de la zone « Gratuit pour l'open source qui vous amène à la page d'enregistrement



The screenshot shows the GitHub Pricing and Signup page. At the top, there's a navigation bar with the GitHub logo, a search bar, and links for Home, Pricing and Signup, Repositories, Blog, and Login. Below the navigation, a large yellow box contains the heading "Choose the plan that's right for you." and a note: "Paid plans are billed monthly and can be upgraded/downgraded/terminated at any time without penalty." A green box highlights the "Open Source" plan, which is labeled "Free!". It includes features: Unlimited Public Repositories, Unlimited Public Collaborators, and 300 MB<sup>1</sup> Disk Space. There's also a "Sign Up!" button.

## GitHub: Création d'un compte utilisateur

- Vous devez choisir un nom d'utilisateur qui n'est pas déjà utilisé dans le système et saisir une adresse e-mail qui sera associée au compte et un mot de passe.



Sign up (log in)

Username

Email Address

Password

Confirm Password

SSH Public Key (explain ssh keys)  
Please enter one key only. You may add more later. This field is not required to sign up.

You're signing up for the free plan. If you have any questions please [email support](#).

By signing up, you agree to the [Terms of Service](#), [Privacy](#), and [Refund](#) policies.

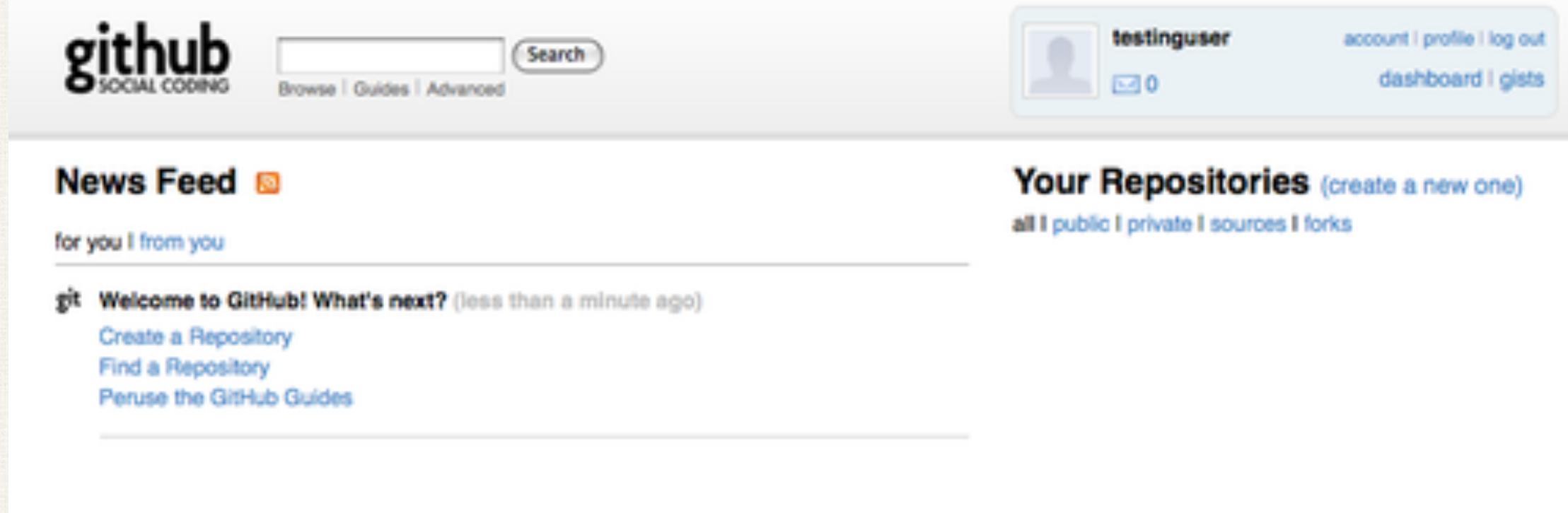
I agree, sign me up!

Si vous l'avez, c'est le bon moment pour ajouter votre clef publique SSH.

Copiez le contenu de la clef publique et collez-le dans la boîte à texte « Clés SSH publiques ».

Cliquez sur « Créeer un compte » pour avoir accès au tableau de bord du nouvel utilisateur

## GitHub: Création d'un compte utilisateur



The screenshot shows the GitHub homepage. At the top left is the GitHub logo with "SOCIAL CODING". To its right is a search bar with a "Search" button. Below the search bar are links for "Browse", "Guides", and "Advanced". On the far right of the header are links for "account", "profile", "log out", "dashboard", and "gists". A user profile for "testinguser" is displayed, showing a placeholder profile picture, 0 notifications, and a link to "testinguser's profile".

**News Feed** 

for you | from you

---

git Welcome to GitHub! What's next? (less than a minute ago)

[Create a Repository](#)  
[Find a Repository](#)  
[Peruse the GitHub Guides](#)

---

**Your Repositories** [\(create a new one\)](#)

[all](#) | [public](#) | [private](#) | [sources](#) | [forks](#)

## GitHub: Crédit d'un nouveau Dépôt

- Commencez en cliquant sur « Nouveau dépôt » juste à côté de vos dépôts sur le tableau de bord utilisateur.
- Un formulaire « Crédit un nouveau dépôt » apparaît pour vous guider dans la création d'un nouveau dépôt
- Le strict nécessaire consiste à fournir un nom au projet, mais vous pouvez aussi ajouter une description.
- Ensuite, cliquez sur le bouton « Crédit un dépôt ».
- Voilà un nouveau dépôt sur GitHub.



## GitHub: Création nouveau Dépôt

**Create a New Repository**

Create a new empty repository into which you can push your local git repo.

**NOTE:** If you intend to push a copy of a repository that is already hosted on GitHub, then you should fork it instead.

Project Name

Description

Homepage URL.

Who has access to this repository? (You can change this later)

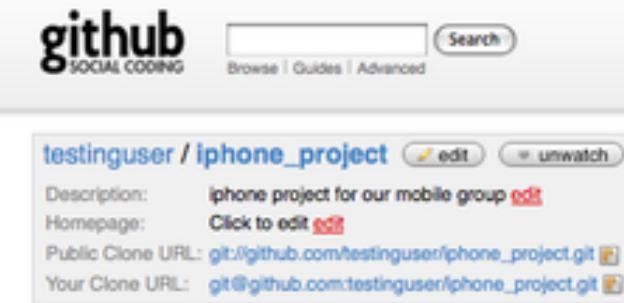
Anyone (learn more about public repos)  
 Upgrade your plan to create more private repositories!

Comme il n'y a pas encore de code, GitHub affiche les instructions permettant de créer un nouveau projet, de pousser un projet Git existant ou d'importer un projet depuis un dépôt Subversion

Ces instructions sont similaires à ce que nous avons déjà décrit. Pour initialiser un projet qui n'est pas déjà dans Git, tapez

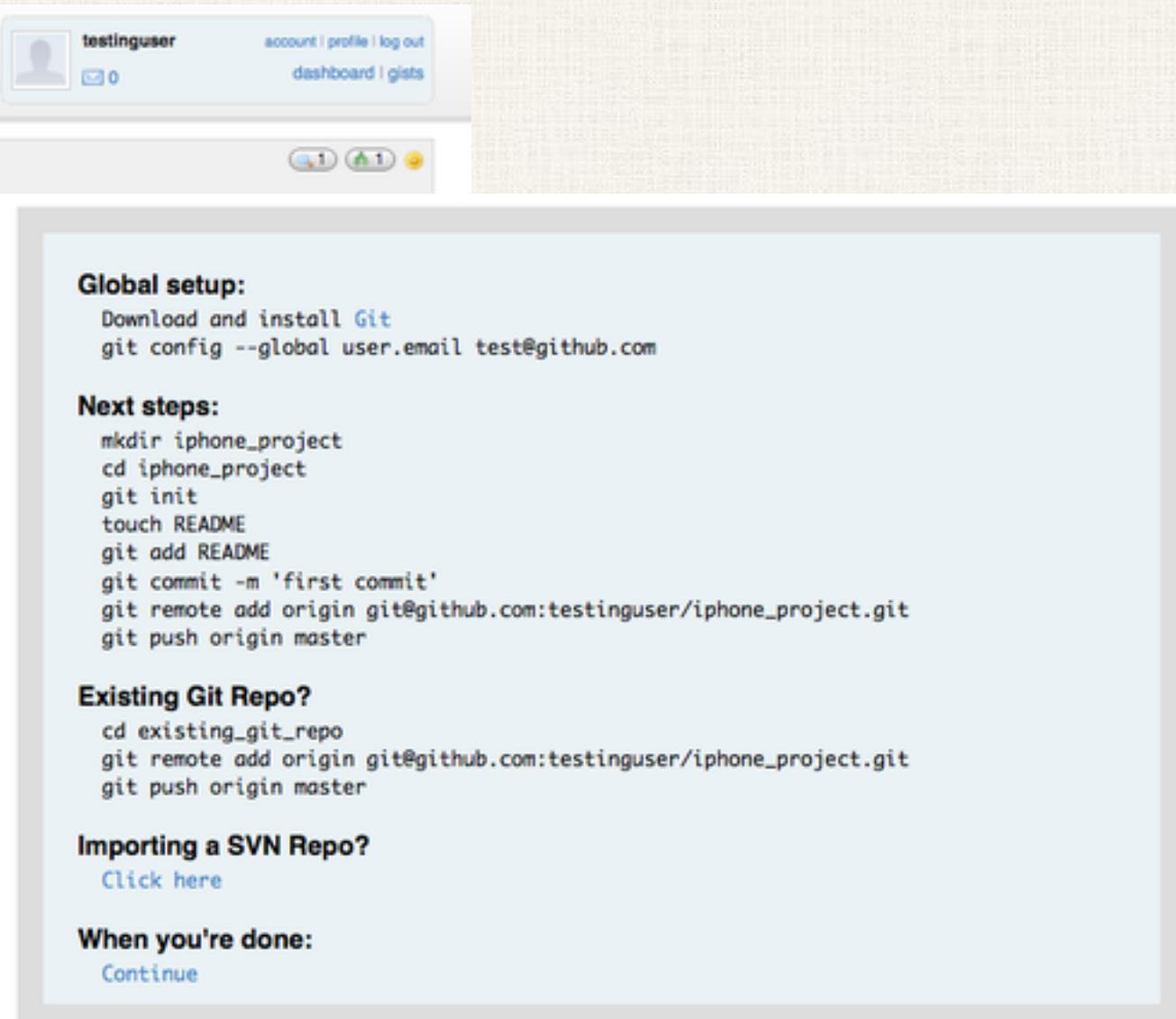


## GitHub: Création nouveau Dépôt



Description: iphone project for our mobile group [edit](#)  
Homepage: [Click to edit](#)  
Public Clone URL: [git://github.com/testinguser/iphone\\_project.git](git://github.com/testinguser/iphone_project.git)  
Your Clone URL: [git@github.com:testinguser/iphone\\_project.git](git@github.com:testinguser/iphone_project.git)

Information générale du projet local



**Global setup:**  
Download and install [Git](#)  
`git config --global user.email test@github.com`

**Next steps:**  
`mkdir iphone_project  
cd iphone_project  
git init  
touch README  
git add README  
git commit -m 'first commit'  
git remote add origin git@github.com:testinguser/iphone_project.git  
git push origin master`

**Existing Git Repo?**  
`cd existing_git_repo  
git remote add origin git@github.com:testinguser/iphone_project.git  
git push origin master`

**Importing a SVN Repo?**  
[Click here](#)

**When you're done:**  
[Continue](#)











