

Formation Git pour développeurs

Pierre Bretéché

Dawan

pbreteche@dawan.fr

March 10, 2016



Plan d'intervention I

- 1 Introduction
 - Présentation
 - Classification
 - Git
- 2 Fondamentaux
 - Le dépôt
 - Historiques des validations
 - Annuler une action
 - Dépôt distant
 - Étiquettes
 - Alias
- 3 Les branches avec Git
 - Vue d'ensemble



Plan d'intervention II

- Fusion
- Gestion
- Workflow
- Branches distantes
- Bases

4 Git sur un serveur

- Installation sur un serveur
- GitWeb
- Git Gui
- GitLab

5 GitHub

- Créer un compte
- Contribuer à un projet



Plan d'intervention III

- 6 Git distribué
 - Développements distribués



Introduction



Objectifs

- Mettre en place une solution de contrôle de version basée sur Git
- Gérer les versions des projets du dépôt de données
- Mettre en place une stratégie de collaboration entre développeurs
- Préparer les versions potentiellement distribuables



Documents

illustrations provenant de git-scm.com/doc sous licence CC

Sources:

- Référence : <https://git-scm.com/docs>
- Book : <https://git-scm.com/book>
- GitHub guides : <https://guides.github.com/>



Définition

- Version Control System (Système de contrôle de version)
 - enregistre les modifications d'un ensemble de fichiers
 - permet de revenir sur une version spécifique
 - permet de revenir en arrière sur un fichier spécifique
 - permet de retrouver la dernière modification qui a pu introduire un problème

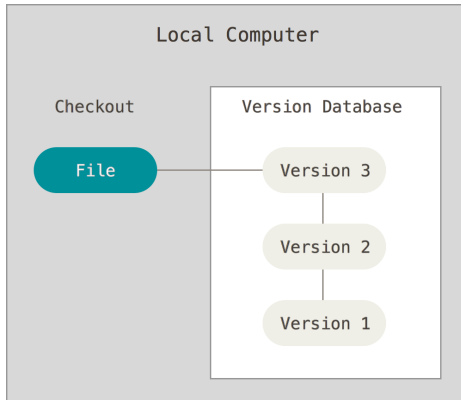


Contrôle de version local

- peut être fait manuellement
 - copie des fichiers dans un répertoire spécifique
 - pros: très simple
 - cons: erreur de manipulation facile et non-réversible
- archivage des fichiers
 - pros: toujours très simple
 - cons: lourdeur de la manipulation



Contrôle de version local



Contrôle de version local

- RCS (Revision Control System)
 - successeur de SCCS (Source Code Control System)
 - 1982, Walter F. Tichy (4.3BSD)
 - depuis 1990, FSF, GPL
 - mémorise des ensembles de « patches » afin de recréer n'importe quelle version du fichier à la demande

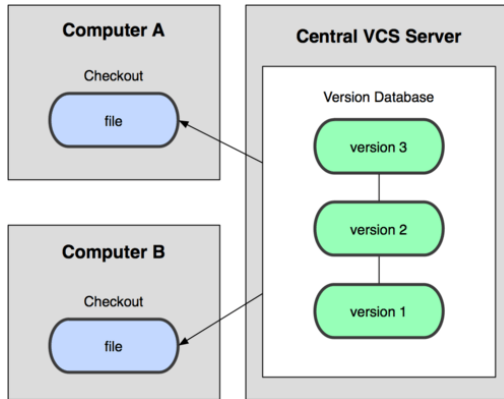


Contrôle de version centralisé - fonctionnement

- Permet le travail collaboratif
- Serveur unique contenant toutes les versions
- Clients multiples empruntant des fichiers
- Mode de fonctionnement standard durant des années
- CVS, Subversion, Perforce . . .



Contrôle de version centralisé -schéma



Contrôle de version centralisé - pros/cons

- pros: chacun peut savoir qui fait quoi (dans une certaine mesure)
- pros: un administrateur peut gérer des droits
- cons: point unique de panne
 - coupure du serveur : plus personne ne peut collaborer
 - corruption du disque: données définitivement perdue

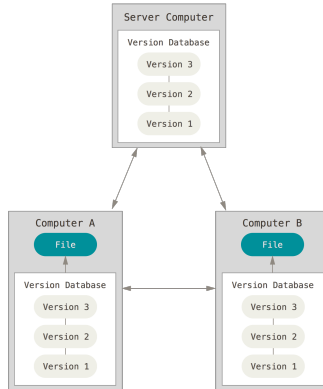


Contrôle de version distribué - fonctionnement

- Plus d'extraction de la dernière version du projet
- Mais réplication du dépôt
- pros: sécurité grâce à la redondance des dépôts
- pros: permet l'organisation de «groupes de travail»



Contrôle de version centralisé -schéma



Histoire

- 1991 - 2002 noyau Linux gérer via des patches et des archives
- 2002 - 2005 DCVS Bitkeeper
- 2005 Bitkeeper devient payant, Linus Torvalds crée un successeur de Bitkeeper, objectifs:
 - vitesse
 - simplicité
 - développements non-linéaires (branches)
 - compacité des données



Mode de stockage

Git gère les versions sous formes d'instantanés (snapshot) et non des différences (deltas).

Un système de référence permet d'éviter de stocker plusieurs fois un fichier non-modifié



Checks Over Time



Checks Over Time



Travailler en local

La plupart des opérations se déroulent localement. Pas de ralentissement dû à la latence du réseau.

Exemple:

- parcourir l'historique d'un fichier
- générer un patch entre deux versions arbitraires d'un fichier
- travailler en mode «hors connexion» (dans le train, en dehors du réseau d'entreprise, etc.)



Intégrité

Tout est vérifié via une somme de contrôle (SHA-1).

Pas de risque de modification non-gérée d'un fichier, ni de corruption lors d'un transfert.

Sert également d'identifiant de fichier dans la base de données Git. Quasiment que des ajouts de données, tout est réversible.

Perte ou corruption de modifications uniquement quand elles n'ont pas été rentrées en base

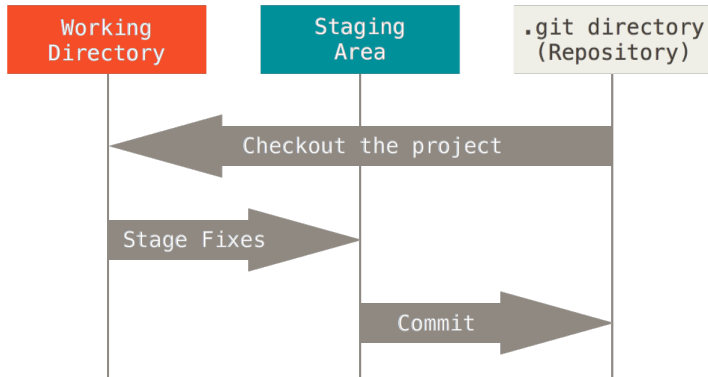


Les trois états d'un fichier

- validé (committed): stockées en sécurité localement
répertoire Git, ensemble des données du projet
- modifié (modified): pas encore validé en base
répertoire de travail, fichiers en cours d'édition
- indexé (staged): marqué pour le prochain instantané
zone de préparation



Stockage par flux d'instantanés



Personalisation

Définir quelques options basiques (nom d'utilisateur, éditeur de texte par défaut ...)

3 niveaux de configuration

- local: spécifique à un dépôt dans `.git/config` (défaut)
- utilisateur: spécifique à un dépôt dans `/.gitconfig` (`-global`)
- système: spécifique à un dépôt dans `/etc/gitconfig` (`-system`)

```
$ git config --global user.name "John_Doe"  
$ git config --global user.email johndoe@example.com  
$ git config --global core.editor nano
```



Personalisation

Vérifier vos options

```
$ git config --list
```

Aide sur une commande

```
$ git help config
```



Fondamentaux



Création

Un nouveau dépôt peut se créer à partir de:

- un répertoire existant

```
$ git init
```

```
$ git add *.php  
$ git add LICENSE  
$ git commit -m 'initial project version'
```

- un dépôt existant

```
$ git clone https://github.com/pbreteche/jekyll --test
```

télécharge toutes les données du dépôt et génère une copie de travail



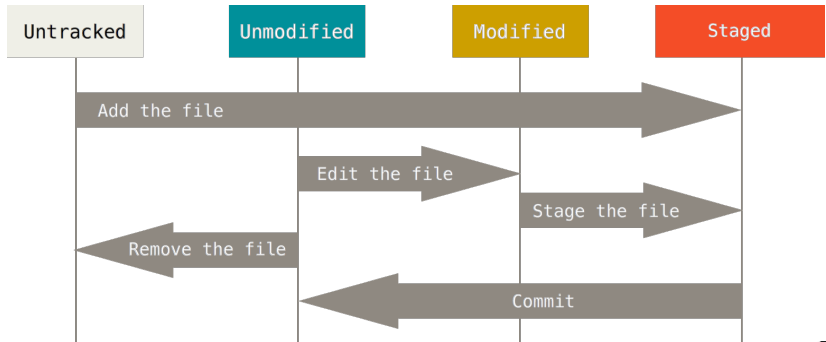
Enregistrement

Atelier: clonez le projet jekyll-test

<https://github.com/pbreteche/jekyll-test>



Enregistrement



Vérifier le dépôt

Détermine quels fichiers sont dans quel état.

```
$ git status
```



Ajouter un fichier non-suivi

- 1 Créez un nouveau fichier README.md
- 2 Vérifiez votre dépôt
- 3 Qu'observez-vous ?



Ajouter un fichier non-suivi

- 1 Créez un nouveau fichier README.md
- 2 Vérifiez votre dépôt
- 3 Qu'observez-vous ?
- 4 Ajoutez le fichier via la commande

```
$ git add README.md
```

- 5 Vérifiez à nouveau votre dépôt



Modifier un fichier validé

- 1 Modifiez un fichier de votre choix
- 2 Vérifiez votre dépôt
- 3 Qu'observez-vous ?



Modifier un fichier validé

- 1 Modifiez un fichier de votre choix
- 2 Vérifiez votre dépôt
- 3 Qu'observez-vous ?
- 4 Ajoutez le fichier à l'index toujours via la commande

```
$ git add <votre-fichier>
```

- 5 Vérifiez à nouveau votre dépôt



Modifier un fichier indexé

- 1 Modifiez un fichier à nouveau ce fichier
- 2 Vérifiez votre dépôt
- 3 Qu'observez-vous ?



Modifier un fichier indexé

- 1 Modifiez un fichier à nouveau ce fichier
- 2 Vérifiez votre dépôt
- 3 Qu'observez-vous ?
- 4 Actualisé l'index encore via la commande

```
$ git add <votre-fichier>
```

- 5 Vérifiez à nouveau votre dépôt



Vérifier le dépôt - version courte

```
$ git status --short
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

- ❶ 2 colonnes: index puis wd
- ❷ ?: non suivi
- ❸ A: Ajouté
- ❹ M: Modifié



Ignorer des fichiers

Il est possible de définir des règles pour ignorer certains fichiers (exemples: journaux, caches, paramètres locaux ...)

```
$ cat .gitignore
```

- une ligne est une règle d'exclusion
- patron standard de fichier
 - «?» caractère quelconque
 - «*» suite quelconque de caractères
 - «[abc]», «[a-f]» caractères alternatifs
 - «/**/» hiérarchie de répertoires quelconque
- terminer par «/», répertoire
- commencer par «!», exception
- commenter avec «#»



Inspecter les modifications

Pour aller plus loin que status, diff permet de calculer un différentiel entre le wd et l'index (modification non-indexée)

```
$ git diff
```

Pour les modifications indexées:

```
$ git diff --staged
```



Valider les modifications

Une fois l'index dans l'état désiré, il peut être validé.

```
$ git commit
```

Un message est utile pour expliquer la modification

```
$ git commit -m "Story 182: Fix benchmarks for speed"
```

L'option «a» permet de valider tous les fichiers modifiés du wd sans passer nécessairement par l'index

```
$ git commit -a -m "Story 182: Fix benchmarks for speed"
```



Valider les modifications

ProTip de GitHub

Commit messages are important, especially since Git tracks your changes and then displays them as commits once they're pushed to the server. By writing clear commit messages, you can make it easier for other people to follow along and provide feedback.



Supprimer un fichier

Effacement dans l'index avant validation

Le fichier est également effacé de la copie de travail

```
$ git rm UNUSED_FILE.md
```

Le fichier est conservé dans la copie de travail

```
$ git rm --cached UNUSED_FILE.md
```

Autres exemples:

```
$ git rm log/\*.log
```

```
$ git rm -fr /cache/\*/
```

Déplacer un fichier

Concrètement, il n'y a pas de suivi du renommage dans Git. Il faut retirer l'ancien fichier de l'index et ajouter le nouveau:

```
$ mv LISEZMOI.txt LISEZMOI  
$ git rm LISEZMOI.txt  
$ git add LISEZMOI
```

Il est toutefois possible d'effectuer ces trois actions en une commande

```
$ git mv LISEZMOI.txt LISEZMOI
```



Travaillons avec le projet simplegit-progit
<https://github.com/schacon/simplegit-progit>



Visualiser l'historique

```
$ git log
```

Liste en ordre chronologique inversé les validations réalisées.

Beaucoup d'options disponibles

<https://git-scm.com/docs/git-log>

Exemple: les patches sur les deux derniers *commits*

```
$ git log -p -2
```

Exemple: personnalisation du format avec somme de contrôle abrégée, sujet et représentation du graphe

```
$ git log --pretty=format:"%h %s" --graph
```

Limiter la longueur l'historique

Spécifier une date limite de début|fin relative|absolue

```
$ git log --since=2.weeks --until=2016-02-29
```

Filtrer selon l'auteur ou le contenu du message

```
$ git log --author=pbreteche --grep=templating  
$ git log --grep=templating --grep=style --all --match
```

Filtrer sur le contenu du patch -S(String),ajout|suppression d'une portion de code ou -G(regex)ajout|suppression|modification d'une portion de code:

```
$ git log -S"if _ \($name_=="
```


Annuler une action

Corriger un commit incomplet
(particulièrement utile pour les committers précoces)

```
$ git commit -m 'validation initiale '  
$ git add fichier_oublie  
$ git commit --amend
```

Désindexer un fichier

```
$ git reset HEAD CONTRIBUTING.md
```

Annuler les mods sur un fichier: le redescendre depuis le dépôt

```
$ git checkout -- CONTRIBUTING.md
```

Astuce: la command status propose des actions en fonction de l'état des fichiers

Limiter la longueur l'historique

Afficher le dépôt distant

```
$ git remote
```

Ajouter un dépôt distant

```
$ git remote add pb https://github.com/pbreteche/awesome-pro
```



Tirer

Tirer depuis un dépôt distant dans une nouvelle branche

```
$ git fetch origin
```

Tirer depuis un dépôt distant dans une fusionner automatiquement dans la même branche locale

```
$ git pull
```



Pousser

Pousse votre branche sur une autre

```
$ git push origin master
```

- Des droits d'écriture peuvent être exigés depuis le dépôt distant
- Personne ne doit avoir poussé depuis la dernière fois que vous avez tiré. Si c'est le cas, il faudra tirer et fusionner avant de pouvoir pousser



Inspecter

```
$ git remote show origin
* distante origin
URL de rapatriement : https://github.com/schacon/ticgit
URL push : https://github.com/schacon/ticgit
Branche HEAD : master
Branches distantes :
master suivi
ticgit suivi
Branche locale configuree pour 'git pull' :
master fusionne avec la distante master
Reference locale configuree pour 'git push' :
master pousse vers master (a jour)
```



Renommer, supprimer

Renomme votre référence vers un dépôt distant

```
$ git remote rename pb pierre
```

Supprime votre référence vers un dépôt

```
$ git remote rm pierre
```



Lister

Les étiquettes permettent de marquer des états particuliers de vos versions (par exemple: version de publication) Tout lister

```
$ git tags
```

Lister selon un motif

```
$ git tags -l 'v1.8.5*'
```



Créer

Il existe deux types d'étiquettes: allégées et annotées.

- allégée: pointeur vers un commit spécifique

```
$ git tag v1.4
```

- annotée: ajoute une somme de contrôle, nom et email du créateur, un message et une signature GPG

```
$ git tag -a v1.4 -m 'ma version 1.4'
```



Créer

Étiquetage d'un commit historique:
Rechercher le hash abrégé du commit
Puis:

```
$ git tag -a v1.4 9fceb02 -m 'ma version 1.4'
```



Pousser

Les étiquettes ne partent pas automatiquement avec les push.
Pousser une étiquette en particulier

```
$ git push origin v1.5
```

Pousser toutes les nouvelles étiquettes

```
$ git push origin --tags
```



Pousser

Permet de personnaliser / raccourcir / prédéfinir des commandes dans Git

Raccourcir le nom d'une commande standard:

```
$ git config --global alias.st status
```

Se créer une commande perso

```
$ git config --global alias.last 'log -1 HEAD'
```



Branches

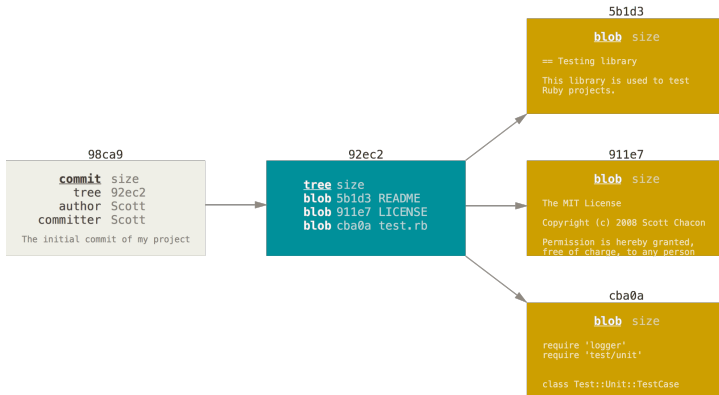


Stockage d'un commit

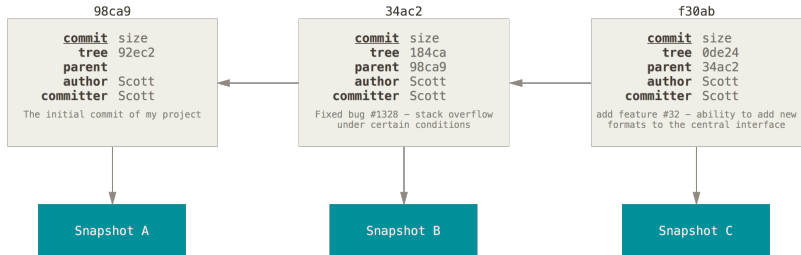
- 1 objet «commit» : métadonnées du commit
- n objets «tree» : contenu de chaque répertoire sous forme de référence vers un fichier
- m objets «blob» : contenu de chaque fichier



Un commit et son arbre



Commits et leurs parents



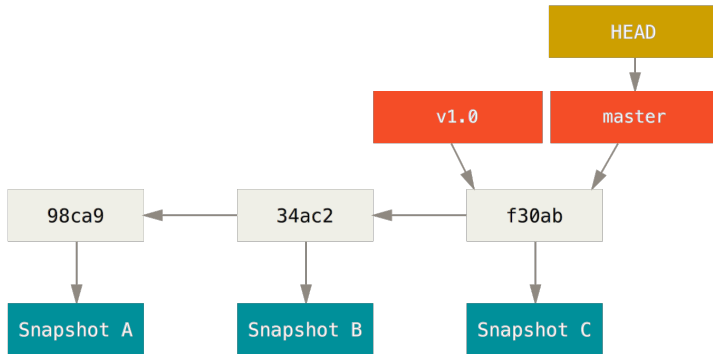
Les branches

- Une branche n'est rien d'autre qu'un pointeur léger vers un commit
- Par défaut, git init crée une branche nommée master
- À chaque commit, le pointeur de branche se déplace pour rester en permanence sur le dernier commit réalisé
- Créer une branche est similaire à `ln -s`

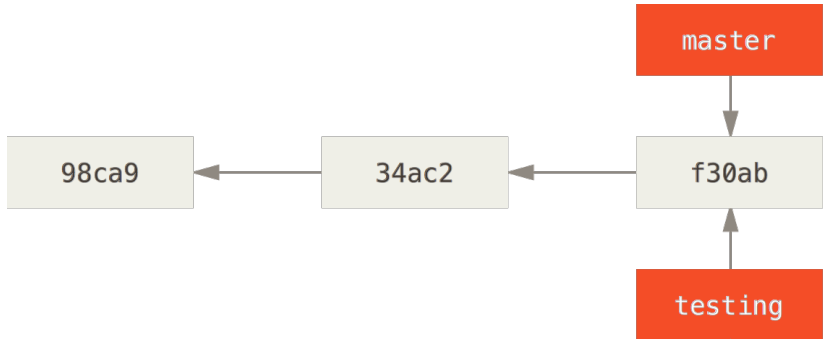
```
$ git branch testing
```



Branches et historique



Créer une nouvelle branche

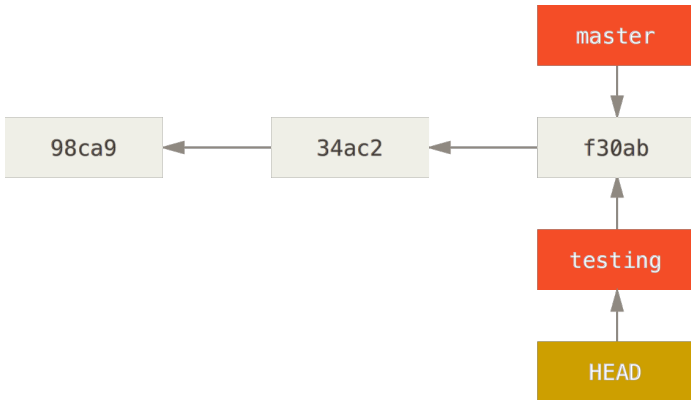


Se déplacer dans les branches

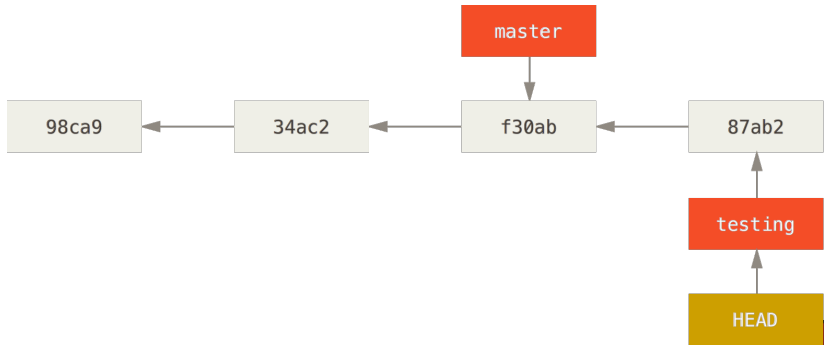
- HEAD est un pointeur supplémentaire indiquant la branche courante
- `git branch` se limite à créer un nouveau pointeur
- `git checkout` sélectionne la nouvelle branche courante
- Au prochain commit, ce sera nouvelle branche sélectionnée qui se déplacera



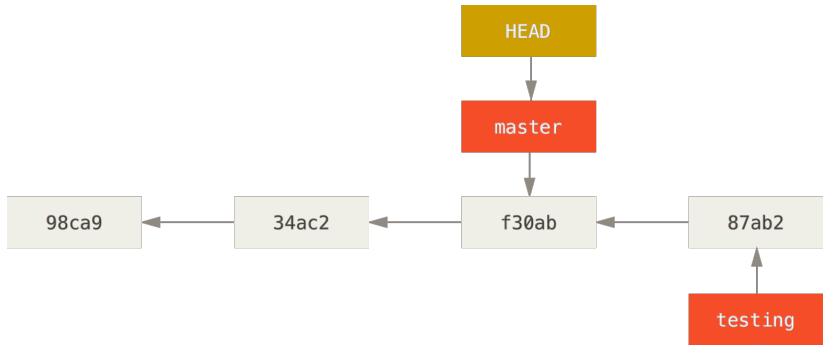
git checkout testing



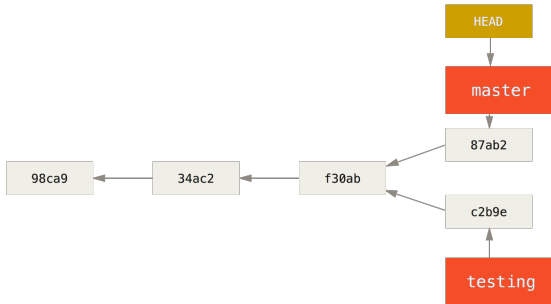
git commit -a -m 'made a change'



```
git checkout master
```



```
git commit -a -m 'made other changes'
```



Checkout

- changer de branches met à jour le répertoire de travail et l'index
- les modifications locales sont conservées

ProTip GitHub:

Branching is a core concept in Git, and the entire GitHub Flow is based upon it. There's only one rule: anything in the master branch is always deployable.

Because of this, it's extremely important that your new branch is created off of master when working on a feature or a fix. Your branch name should be descriptive (e.g., refactor-authentication, user-content-cache-key, make-retina-avatars), so that others can see what is being worked on.



Les bases

Un scénario classique

- ❶ vous devez développer une nouvelle fonctionnalité
- ❷ vous créez une branche
- ❸ vous commencez à travailler dessus



Les bases

Un scénario classique

- ❶ vous devez développer une nouvelle fonctionnalité
- ❷ vous créez une branche
- ❸ vous commencez à travailler dessus

Un problème critique est remonté, il faut réparé tout de suite



Les bases

Un scénario classique

- ❶ vous devez développer une nouvelle fonctionnalité
- ❷ vous créez une branche
- ❸ vous commencez à travailler dessus

Un problème critique est remonté, il faut réparé tout de suite

- ❶ vous basculer sur la branche de production
- ❷ vous créez une branche pour le correctif
- ❸ une fois testée, vous fusionnez et poussez en prod
- ❹ vous rebasculez sur votre branche de travail



Nouvelle fonctionnalité

Vous commencez à travailler sur la tâche 53

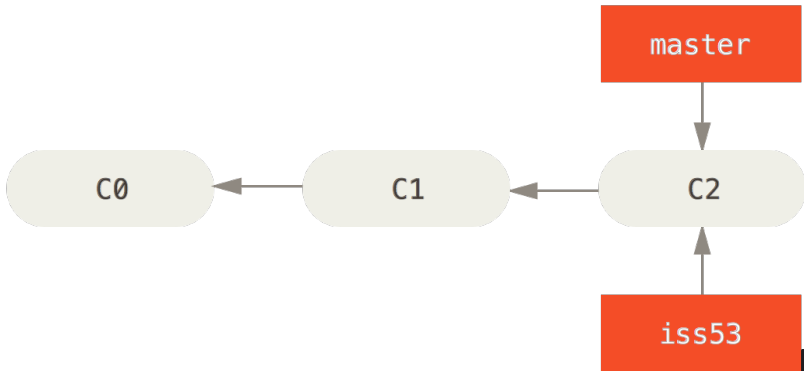
```
$ git checkout -b iss53
```

L'option -b permet de créer une branche à la volée

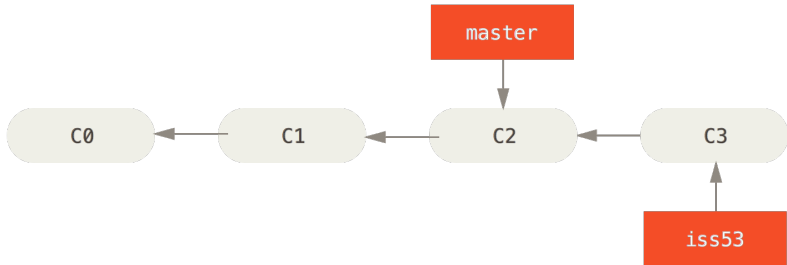
```
$ git branch iss53  
$ git checkout iss53
```



```
git checkout -b iss53
```



git commit -a -m 'add footer [#53]'



Déclaration du problème

Vous rebaseculez sur la branche master

On part du principe que tout votre travail actuel est validé
(commit)

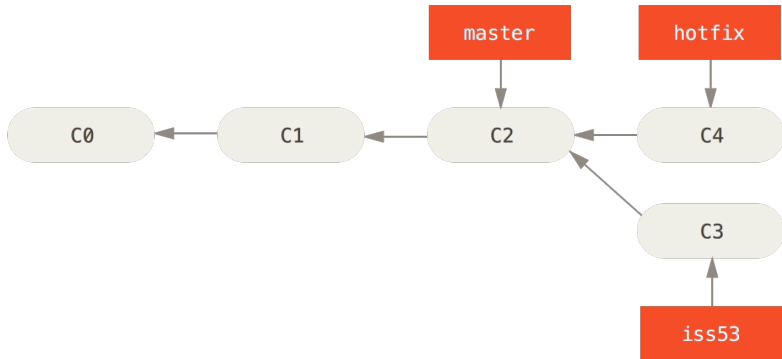
```
$ git checkout master
```

Création de la branche de correction

```
$ git checkout -b hotfix  
$ git commit -a -m 'fixed the broken email address'
```



`git commit -a -m 'fixed the broken email address'`



Problème corrigé

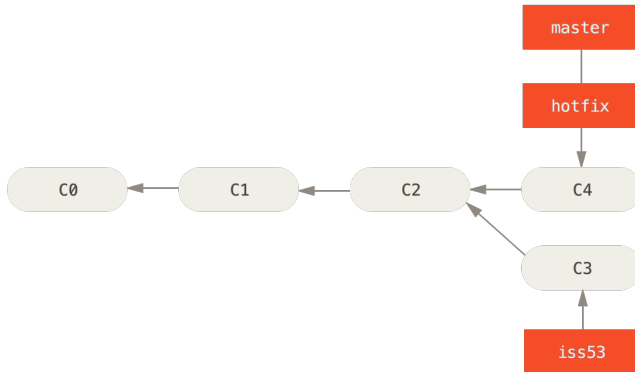
Déroulement des tests validé
vous fusionnez la branche hotfix avec la master

```
$ git checkout master  
$ git merge hotfix
```

Comme hotfix est un descendant direct de master, git s'est contenté de faire avancer le pointeur de master



git merge hotfix



Suppression de la branche hotfix

La branche hotfix n'étant plus utile, on la supprime
option -d (ou --delete)

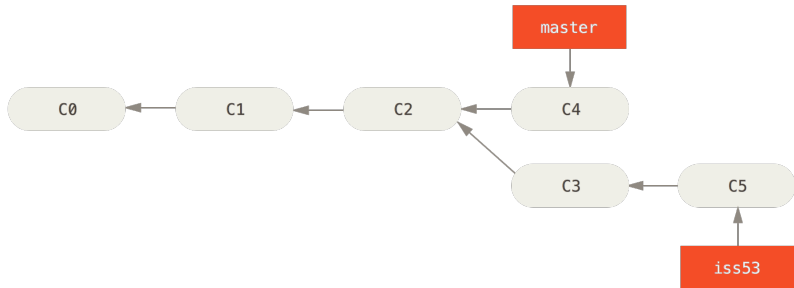
```
$ git branch -d hotfix
```

Reprise du travail sur la branche iss53

```
$ git checkout iss53  
$ git commit -a -m 'finished the new footer [issue 53]'
```



```
git commit -a -m 'finished the new footer [issue 53]'
```



Fusion de la nouvelle fonctionnalité

Le travail sur iss53 étant terminé, on le fusionne dans master

```
$ git checkout master  
$ git merge iss53
```

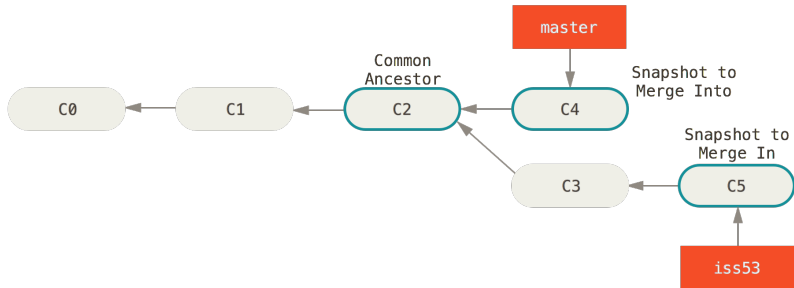
La fusion est légèrement différente. La branche actuelle n'est plus un ancêtre direct de la branche à fusionner.

La stratégie «three-way merge» s'appuie sur le plus proche ancêtre commun

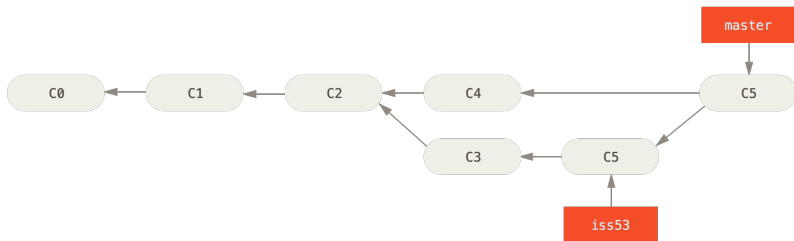
Un nouveau commit est généré automatiquement, c'est un «merge commit»



three-way merging



merge commit



Conflit de fusion

Si une même partie d'un même fichier a été modifié différemment dans les deux branches, Git ne peut effectuer la fusion automatiquement

```
$ git merge prob53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the re
```

Le commit de fusion n'est pas créé. La liste des fichiers en conflit de fusion est disponible dans le rapport:

```
$ git status
```



Conflit de fusion

Le fichier à fusionner dans votre répertoire de travail est annoté

```
<<<<<<< HEAD:index.html  
contenu de la branche master  
=====  
contenu de la branche iss53  
>>>>>>> iss53:index.html
```

Vous pouvez éditer le fichier à la main
ou utiliser un outil graphique

```
$ git mergetool
```

Quand tous les conflits sont résolus, validez avec «commit»



Gestion

Lister les branches (* indique le HEAD)

```
$ git branch
```

avec le dernier commit

```
$ git branch -v
```

—merge et —no-merge filtre les branches actuellement fusionnée ou non avec le HEAD

```
$ git branch —merge
```

Les branches non fusionnées déclenche un message d'erreur à la suppression



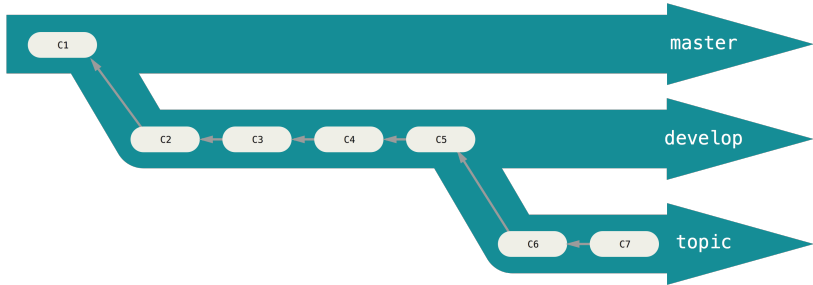
Branches longues

Il est possible de fusionner une même branche plusieurs sur une longue période

Chaque branche correspond à un niveau de stabilité différent, master étant la plus stable et la plus ancienne



Branches longues



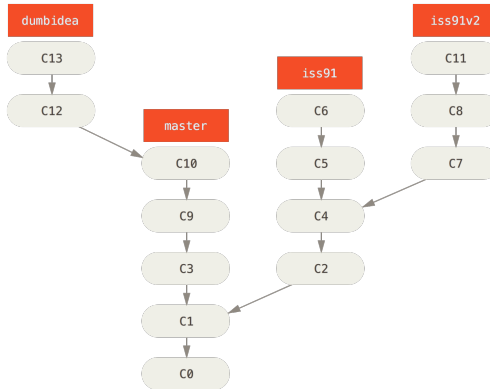
Branches thématiques

Chaque fonctionnalité possède sa branche, les créations peuvent être nombreuses

On supprime généralement la branche après fusion



Branches thématiques



Branches distantes

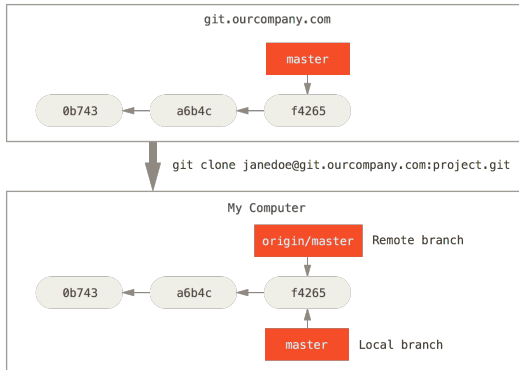
Références locales vers l'état des branches du dépôt distant

Non modifiable par l'utilisateur, uniquement par la communication avec le dépôt distant

Nommées tel que: repository-name/branch-name



Branches distantes



Mise à jour des branches distantes

Après un clone, si vous travaillez, votre branche locale va devenir un descendant de la branche distante

La branche distante évoluant certainement de son côté, vous pouvez récupérer son historique

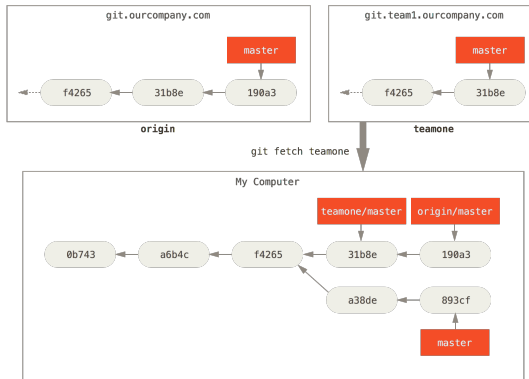
```
$ git fetch origin
```

il est possible de travailler avec plusieurs serveurs distants

```
$ git remote add a-team https://git.a-team.acme.com  
$ git fetch a-team
```



Plusieurs dépôts distants



Pousser les branches

Pas de synchronisation automatique entre les branches des différents dépôts

Vous pouvez avoir des branches privées **et** des branches partagées

```
$ git push origin <branch-name>
```

Si la branche locale et distante ne porte pas le même nom

```
$ git push origin <local-branch-name>:<remote-branch-name>
```



Travailler une nouvelle branche distante

Après un fetch, vous avez récupéré une branche distante (non éditée)

En fusionnant sur votre branche de travail

```
$ git merge origin/serverfix
```

En créant une nouvelle branche locale

```
$ git checkout -b serverfix origin/serverfix
```



Branche de suivi

Une branche locale créée à partir d'une branche distante est appelée «tracking branch» et celle qu'elle suit «upstream branch»
La synchronisation se fait via pull (ou fetch + merge) / push
La création d'une tracking branch peut se faire directement:

```
$ git checkout --track origin/serverfix  
$ git checkout -b correctionserveur origin/serverfix
```

Visualisation des branches avec indication de suivi

```
$ git branch -vv
```

Pour avoir une information à jour

```
$ git fetch --all; git branch -vv
```



Les bases

Le «rebasing» est le second moyen d'intégrer des modifications après la fusion.

Consiste à prendre le patch de modification de la branche à intégrer et de l'appliquer sur la branche courante.

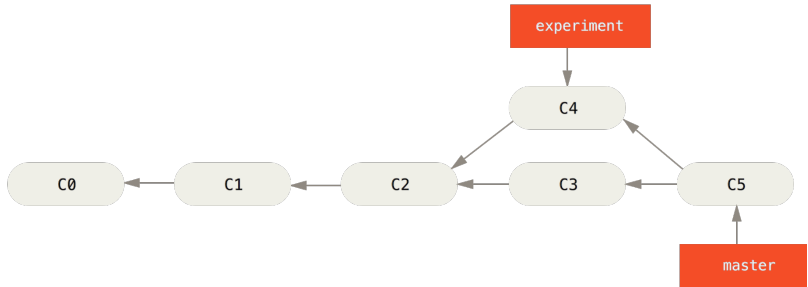
```
$ git checkout experiment  
$ git rebase master
```

Une fois le rebase effectué, vous pouvez faire un avance-rapide

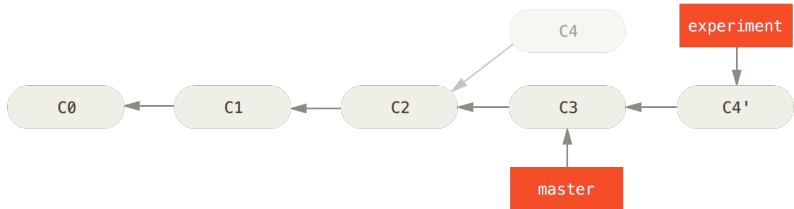
```
$ git checkout master  
$ git merge experiment
```



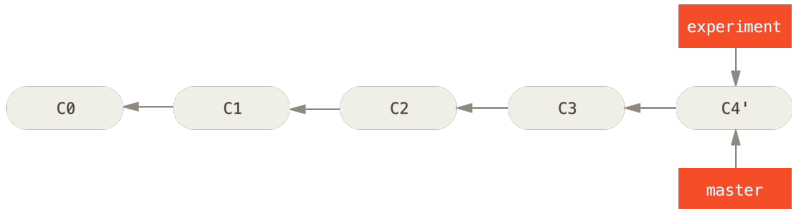
Fusions de C4 sur C3 via C2



Rebase de C4 sur C3



Rebase de C4 sur C3



Les bases

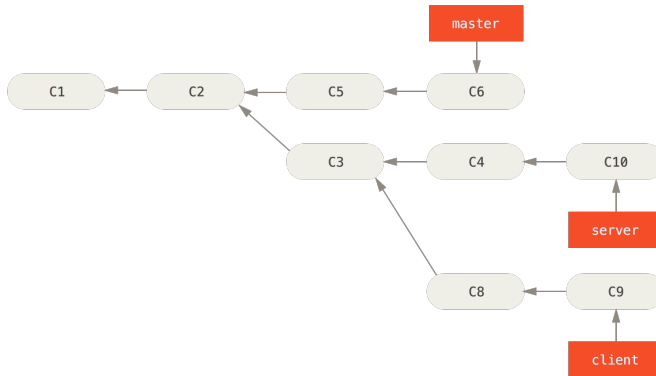
Intérêt: rend l'historique linéaire

Souvent utile pour contribuer sur un projet distant car le propriétaire du projet n'a plus qu'à faire un avance-rapide ou à appliquer le patch au lieu d'un merge complet



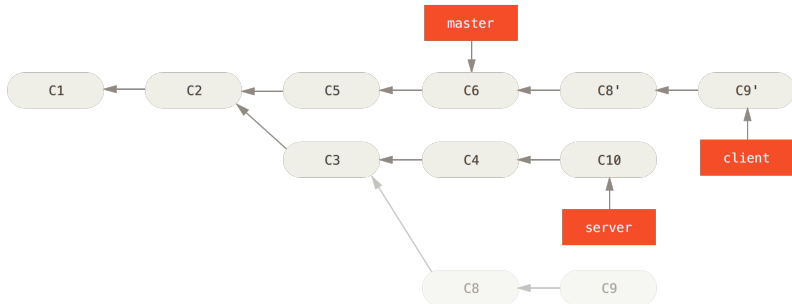
Rebase plus intéressant

Supposons un historique comme suit:



Rebase plus intéressant

Nous voulons remonter le développement de client dans master, mais sans prendre la partie liée à server



Rebase plus intéressant

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

- `--onto master` permet de préciser la nouvelle base (rattachement)
- `serveur` correspond à l'upstream (détachement)
- `client` précise la branche à rejouer (git fait un checkout avant le rebase)



Les dangers du rebasage

Quand vous faites un rebase, vous réécrivez l'histoire.
Ceux qui ont déjà tiré votre histoire précédente devront
re-fusionner sur la nouvelle.
Question: quand rebaser et quand fusionner ?



Serveur



Dépôt nu

Création d'un dépôt nu (brut, ie sans répertoire de travail)

```
$ git clone --bare mon_projet mon_projet.git
```

Copie sur le serveur

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

Et c'est tout!

Enfin presque, un administrateur décidera des protocoles à mettre en place (HTTP, Git, SSH, autre), de configurer les accès utilisateurs, etc.



GitWeb

Création d'un dépôt nu (brut, ie sans répertoire de travail)

```
$ git clone --bare mon_projet mon_projet.git
```

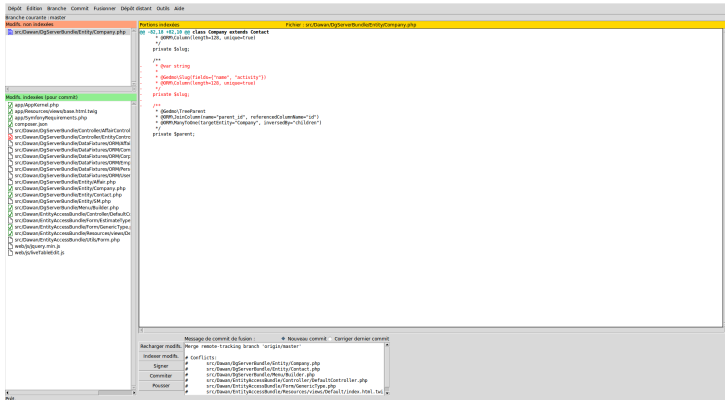
Copie sur le serveur

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



[illegible]

GitGui aperçu



Télécharger GitLab

Peut s'installer sur un serveur Ruby on Rails.
Peut se récupérer dans une VM **Bitnami**.



Lancer la VM dans VirtualBox

- 1 créer une VM Linux Ubuntu 64bits
- 2 mettez plus de 1Gio en RAM
- 3 sélectionnez le disque Bitnami
- 4 régler la conf réseau sur connexion par pont
- 5 démarrer la vm



GitLab VM

```

  ____  _
 / ___|| | | |
| |___| |_| |
 \___|_||_|_|_|

*** Welcome to the Bitnami Gitlab Stack ***
*** Built using Ubuntu 14.04 - Kernel 3.13.0-79-generic (tty2). ***

*** You can access the application at http://10.0.2.15 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to https://wiki.bitnami.com/Virtual_Machines for details. ***

*****
To access the console, please use login 'bitnami'
and password 'bitnami'
*****

ubuntu login: _

```



Bitnami console

```

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

      _--_ _--_ _--_ _--_ _--_ _--_ _--_ _--_ _--_ _--_
     |  _ \ /  _ \ /  _ \ /  _ \ /  _ \ /  _ \ /  _ \
     |_/__/_\_/__/_\_/__/_\_/__/_\_/__/_\_/__/_\_/__/_\_/

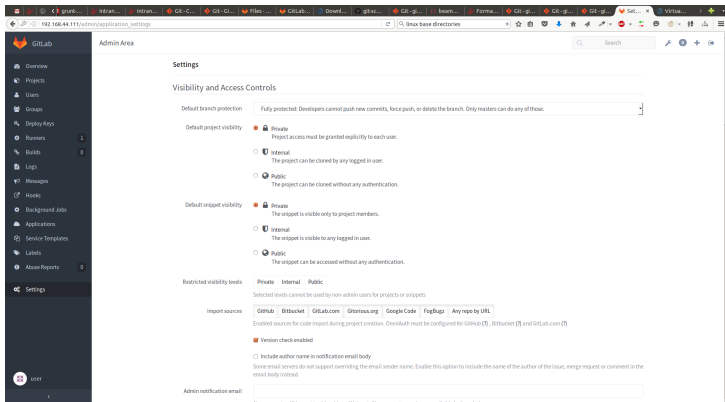
*** Welcome to the Bitnami GitLab 8.5.1-0 ***
*** Bitnami Wiki:   https://wiki.bitnami.com/ ***
*** Bitnami Forums: https://community.bitnami.com/ ***

*****
* Please insert the new user password *
* The default password is 'bitnami' *
*****
Changing password for bitnami.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
Password unchanged
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
bitnami@ubuntu:~$ _

```



GitLab admin



GitHub

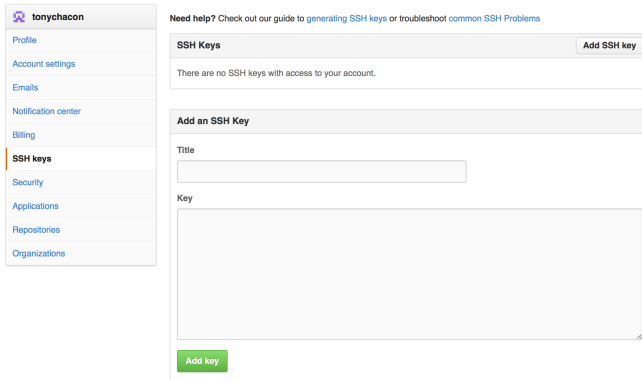


Les dangers du rebasage

Formulaire d'inscription simple sur <https://github.com/>
Renseigner votre clé publique dans les paramètres et vos
informations publiques!




Clé SSH



The screenshot shows the GitHub interface for user **tonychacon**. On the left is a sidebar with navigation links: Profile, Account settings, Emails, Notification center, Billing, **SSH keys** (highlighted with an orange bar), Security, Applications, Repositories, and Organizations. The main content area has a header with the text "Need help? Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)". Below this is a section titled "SSH Keys" with an "Add SSH key" button. A message states "There are no SSH keys with access to your account." Below that is a section titled "Add an SSH Key" containing a "Title" input field, a "Key" text area, and an "Add key" button.



2 Factors Authentication

 **tonychacon**

[Profile](#)
[Account settings](#)
[Emails](#)
[Notification center](#)
[Billing](#)
[SSH keys](#)
Security
[Applications](#)
[Repositories](#)
[Organizations](#)

Two-factor authentication



Status: **Off** ❌

[Set up two-factor authentication](#)

🔔 Two-factor authentication provides another layer of security to your account. [Learn more about two-factor auth at GitHub Help.](#)

Sessions

This is a list of devices that have logged into your account. Revoke any sessions that you do not recognize.

  **Paris 85.168.227.34**
Your current session [...](#)
Safari on OS X 10.9.4
Location:
Paris, Ile-de-France, France
Signed in:
September 30, 2014



2 Factors Authentication

«Fork» peut posséder une connotation négative



Git distribué

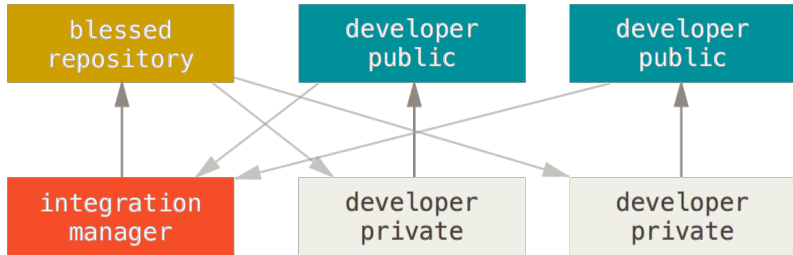


Gestionnaire d'intégration

- 1 Le mainteneur du projet pousse vers son dépôt public.
- 2 Un contributeur clone ce dépôt et introduit des modifications.
- 3 Le contributeur pousse son travail sur son dépôt public.
- 4 Le contributeur envoie au mainteneur un e-mail de demande pour tirer ses modifications depuis son dépôt.
- 5 Le mainteneur ajoute le dépôt du contributeur comme dépôt distant et fusionne les modifications localement.
- 6 Le mainteneur pousse les modifications fusionnées sur le dépôt principal.



Gestionnaire d'intégration



Gestionnaire d'intégration

- ❶ Les simples développeurs travaillent sur la branche thématique et rebasent leur travail sur master. La branche master est celle du dictateur.
- ❷ Les lieutenants fusionnent les branches thématiques des développeurs dans leur propre branche master.
- ❸ Le dictateur fusionne les branches master de ses lieutenants dans sa propre branche master.
- ❹ Le dictateur pousse sa branche master sur le dépôt de référence pour que les développeurs se rebasent dessus.



Gestionnaire d'intégration

