



ORSYS
formation

Support des Travaux pratiques Git

ahmedhosni.contact@gmail.com

Git sur le serveur

- [Protocoles](#)
 - [Protocole local](#)
 - [Avantages](#)
 - [Inconvénients](#)
 - [Protocoles sur HTTP](#)
 - [HTTP Intelligent](#)
 - [HTTP idiot](#)
 - [Avantages](#)
 - [Inconvénients](#)
 - [Protocole SSH](#)
 - [Avantages](#)
 - [Inconvénients](#)
 - [Protocole Git](#)
 - [Avantages](#)
 - [Inconvénients](#)
- [Installation de Git sur un serveur](#)
 - [Copie du dépôt nu sur un serveur](#)
 - [Petites installations](#)
 - [Accès SSH](#)
- [Génération des clés publiques SSH](#)
- [Mise en place du serveur](#)
- [Démon \(*Daemon*\) Git](#)
- [HTTP intelligent](#)
- [GitWeb](#)
- [GitLab](#)
 - [Installation](#)
 - [Administration](#)
 - [Utilisateurs](#)
 - [Groupes](#)
 - [Projets](#)
 - [Crochets \(*Hooks*\)](#)
 - [Usage de base](#)
 - [Coopérer](#)
- [Git hébergé](#)
- [Résumé](#)

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. Bien que vous puissiez techniquement tirer et pousser des modifications depuis et vers des dépôts personnels, cette pratique est déconseillée parce qu'elle introduit très facilement une confusion avec votre travail actuel. De plus, 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 canonique pour collaborer consiste à instancier un dépôt intermédiaire auquel tout le monde a accès, que ce soit pour pousser ou tirer.

Un serveur Git est simple à lancer. Premièrement, vous devez choisir quels protocoles seront supportés. La première partie de ce chapitre traite des protocoles disponibles et de leurs avantages et inconvénients. La partie suivante explique certaines configurations typiques de ces protocoles et comment les mettre en œuvre. Enfin, nous traiterons de quelques types d'hébergement, si vous souhaitez héberger votre code sur un serveur tiers, sans avoir à installer et maintenir un serveur par vous-même.

Si vous ne voyez pas d'intérêt à gérer votre propre serveur, vous pouvez sauter directement à la dernière partie de ce chapitre pour détailler les options pour mettre en place un compte hébergé, avant de continuer au chapitre suivant dans lequel les problématiques de développement distribué sont abordées.

Un dépôt distant est généralement un *dépôt nu* (*bare repository*) : un dépôt Git qui n'a pas de copie de travail. Comme ce dépôt n'est utilisé que comme centralisateur de collaboration, il n'y a aucune raison d'extraire un instantané sur le disque ; seules les données Git sont nécessaires. Pour simplifier, un dépôt nu est le contenu du répertoire `.git` sans fioriture.

Protocoles

Git peut utiliser quatre protocoles réseau majeurs pour transporter des données : local, HTTP, *Secure Shell* (SSH) et Git. Nous allons voir leur nature et dans quelles circonstances ils peuvent (ou ne peuvent pas) être utilisés.

Protocole local

Le protocole de base est le protocole *local* pour lequel le dépôt distant est un autre répertoire dans le système de fichiers. Il est souvent utilisé si tous les membres de l'équipe ont accès à un répertoire partagé via NFS par exemple ou dans le cas moins probable où tous les développeurs travaillent sur le même ordinateur. Ce dernier cas n'est pas optimum car tous les dépôts seraient hébergés de fait sur le même ordinateur, rendant ainsi toute défaillance catastrophique.

Si vous disposez d'un système de fichiers partagé, vous pouvez cloner, pousser et tirer avec un dépôt local. Pour cloner un dépôt ou pour l'utiliser comme dépôt distant d'un projet existant, utilisez le chemin vers le dépôt comme URL. Par exemple, pour cloner un dépôt local, vous pouvez lancer ceci :

```
$ git clone /opt/git/project.git
```

Ou bien cela :

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

Git opère légèrement différemment si vous spécifiez explicitement le protocole `file://` au début de l'URL. Si vous spécifiez simplement le chemin et si la destination se trouve sur le même système de fichiers, Git tente d'utiliser des liens physiques pour les fichiers communs. Si vous spécifiez le protocole `file://`, Git lance un processus d'accès à travers le réseau, ce qui est généralement moins efficace. La raison d'utiliser spécifiquement le préfixe `file://` est la volonté d'obtenir une copie propre du dépôt, sans aucune référence ou aucun objet supplémentaire qui pourraient résulter d'un import depuis un autre système de gestion de version ou d'une action similaire (voir chapitre [Les tripes de Git](#) pour les tâches de maintenance). Nous utiliserons les chemins normaux par la suite car c'est la méthode la plus efficace.

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

```
$ git remote add local_proj /opt/git/project.git
```

Ensuite, vous pouvez pousser vers et tirer depuis ce dépôt distant de la même manière que vous le feriez pour un dépôt accessible sur le réseau.

Avantages

Les avantages des dépôts accessibles sur le système de fichiers sont qu'ils sont simples et qu'ils utilisent les permissions du système de fichiers. Si vous avez déjà un montage partagé auquel toute votre équipe a accès, déployer un dépôt est extrêmement facile. Vous placez la copie du dépôt nu à un endroit accessible de tous et positionnez correctement les droits de lecture/écriture de la même manière que pour tout autre partage. Nous aborderons la méthode pour exporter une copie de dépôt nu à cette fin dans la section suivante [Installation de Git sur un serveur](#).

C'est un choix satisfaisant pour partager rapidement le travail. Si vous et votre coéquipier travaillez sur le même projet et qu'il souhaite partager son travail, lancer une commande telle que `git pull /home/john/project` est certainement plus simple que de passer par un serveur intermédiaire.

Inconvénients

Les inconvénients de cette méthode sont qu'il est généralement plus difficile de rendre disponible un partage réseau depuis de nombreux endroits que de simplement gérer des accès réseau. Si vous souhaitez pousser depuis votre portable à la maison, vous devez monter le partage distant, ce qui peut s'avérer plus difficile et plus lent que d'y accéder directement via un protocole réseau.

Il faut aussi mentionner que ce n'est pas nécessairement l'option la plus rapide à l'utilisation si un partage réseau est utilisé. Un dépôt local n'est rapide que si l'accès aux fichiers est rapide. Un dépôt accessible sur un montage NFS est souvent plus lent qu'un dépôt accessible via SSH sur le même serveur qui ferait tourner Git avec un accès aux disques locaux.

Protocoles sur HTTP

Git peut communiquer sur HTTP de deux manières. Avant Git 1.6.6, il n'existait qu'une seule manière qui était très simple et généralement en lecture seule. Depuis la version 1.6.6, il existe un nouveau protocole plus intelligent qui nécessite que Git puisse négocier les transferts de données de manière similaire à ce qu'il fait pour SSH. Ces dernières années, le nouveau protocole HTTP a gagné en popularité du fait qu'il est plus simple à utiliser et plus efficace dans ses communications. La nouvelle version est souvent appelée protocole HTTP « intelligent » et l'ancienne version protocole HTTP « idiot ». Nous allons voir tout d'abord le protocole HTTP « intelligent ».

HTTP Intelligent

Le protocole HTTP « intelligent » se comporte de manière très similaire aux protocoles SSH ou Git mais fonctionne par-dessus les ports HTTP/S et peut utiliser différents mécanismes d'authentification, ce qui le rend souvent plus facile pour l'utilisateur que SSH, puisque l'on peut utiliser des méthodes telles que l'authentification par utilisateur/mot de passe plutôt que de devoir gérer des clés SSH.

C'est devenu probablement le moyen le plus populaire d'utiliser Git, car il peut être utilisé pour du service anonyme, comme le protocole `git://` aussi bien que pour pousser avec authentification et chiffrement, comme le protocole SSH. Au lieu de devoir gérer différentes URL pour ces usages, vous pouvez maintenant utiliser une URL unique pour les deux. Si vous essayez de pousser et que le dépôt requiert une authentification (ce qui est normal), le serveur peut demander un nom d'utilisateur et un mot de passe. De même pour les accès en lecture.

En fait, pour les services tels que GitHub, l'URL que vous utilisez pour visualiser le dépôt sur le web (par exemple `https://github.com/schacon/simplegit[]`) est la même URL utilisable pour le cloner et, si vous en avez les droits, y pousser.

HTTP idiot

Si le serveur ne répond pas avec un service Git HTTP intelligent, le client Git essaiera de se rabattre sur le protocole HTTP « idiot ». Le protocole idiot consiste à servir le dépôt Git nu comme des fichiers normaux sur un serveur web. La beauté du protocole idiot réside dans sa simplicité de mise en place. Tout ce que vous avez à faire, c'est de copier les fichiers de votre dépôt nu sous la racine de documents HTTP et de positionner un crochet (**hook**) `post-update` spécifique, et c'est tout (voir [Crochets Git](#)). Dès ce moment, tous ceux qui peuvent accéder au serveur web sur lequel vous avez déposé votre dépôt peuvent le cloner. Pour permettre un accès en lecture seule à votre dépôt via HTTP, faites quelque chose comme :

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

Et voilà ! Le crochet `post-update` livré par défaut avec Git lance la commande appropriée (`git update-server-info`) pour faire fonctionner correctement le clonage et la récupération HTTP. Cette commande est lancée quand vous poussez sur ce dépôt (peut-être sur SSH). Ensuite, les autres personnes peuvent cloner via quelque chose comme :

```
$ git clone https://exemple.com/projetgit.git
```

Dans ce cas particulier, nous utilisons le chemin `/var/www/htdocs` qui est le plus commun pour une configuration Apache, mais vous pouvez utiliser n'importe quel serveur web statique – placez juste les dépôts nus dans son chemin. Les données Git sont servies comme de simples fichiers statiques (voir [Les tripes de Git](#) pour la manière exacte dont elles sont servies).

Généralement, vous choisirez soit de lancer un serveur HTTP intelligent avec des droits en lecture/écriture ou de fournir simplement les fichiers en lecture seule par le protocole idiot. Il est rare de mélanger les deux types de protocoles.

Avantages

Nous nous concentrerons sur les avantages de la version intelligente du protocole sur HTTP.

La simplicité vient de l'utilisation d'une seule URL pour tous les types d'accès et de la demande d'authentification seulement en cas de besoin. Ces deux caractéristiques rendent les choses très faciles pour l'utilisateur final. La possibilité de s'authentifier avec un nom d'utilisateur et un mot de passe apporte un gros avantage par rapport à SSH puisque les utilisateurs n'ont plus à générer localement les clés SSH et à télécharger leur clé publique sur le serveur avant de pouvoir interagir avec lui. Pour les utilisateurs débutants ou pour des utilisateurs utilisant des systèmes où SSH est moins commun, c'est un avantage d'utilisabilité majeur. C'est aussi un protocole très rapide et efficace, similaire à SSH.

Vous pouvez aussi servir vos dépôts en lecture seule sur HTTPS, ce qui signifie que vous pouvez chiffrer les communications ; ou vous pouvez pousser jusqu'à faire utiliser des certificats SSL à vos clients.

Un autre avantage est que HTTP/S sont des protocoles si souvent utilisés que les pare-feux d'entreprise sont souvent paramétrés pour les laisser passer.

Inconvénients

Configurer Git sur HTTP/S peut être un peu plus difficile que sur SSH sur certains serveurs. Mis à part cela, les autres protocoles ont peu d'avantages sur le protocole HTTP intelligent pour servir Git.

Si vous utilisez HTTP pour pousser de manière authentifiée, fournir vos information d'authentification est parfois plus compliqué qu'utiliser des clés sur SSH. Il existe cependant des outils de mise en cache d'informations d'authentification, comme Keychain sur OSX et Credential Manager sur Windows pour rendre cela indolore. Reportez-vous à [Stockage des identifiants](#) pour voir comment configurer la mise en cache des mots de passe HTTP sur votre système.

Protocole SSH

SSH est un protocole répandu de transport pour Git en auto-hébergement. 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 un protocole authentifié ; et comme il est très répandu, il est généralement facile à mettre en œuvre et à utiliser.

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

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

Vous pouvez utiliser aussi la syntaxe scp habituelle avec le protocole SSH :

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

Vous pouvez aussi ne pas spécifier de nom d'utilisateur et Git utilisera par défaut le nom de login.

Avantages

Les avantages liés à l'utilisation de SSH sont nombreux. Premièrement, SSH est relativement simple à mettre en place, les *daemons* SSH sont facilement disponibles, les administrateurs réseau sont habitués à les gérer et de nombreuses distributions de systèmes d'exploitation en disposent ou proposent des outils pour les gérer. Ensuite, l'accès distant à travers SSH est sécurisé, toutes les données sont chiffrées et authentifiées. Enfin, comme les protocoles HTTP/S, Git et local, SSH est efficace et permet de compresser autant que possible les données avant de les transférer.

Inconvénients

Le point négatif avec SSH est qu'il est impossible de proposer un accès anonyme au dépôt. Les accès sont régis par les permissions SSH, même pour un accès en lecture seule, ce qui s'oppose à une optique open source. Si vous souhaitez utiliser Git dans un environnement d'entreprise, SSH peut bien être le seul protocole nécessaire. Si vous souhaitez proposer de l'accès anonyme en lecture seule à vos projets, vous aurez besoin de SSH pour vous permettre de pousser mais un autre protocole sera nécessaire pour permettre à d'autres de tirer.

Protocole Git

Vient ensuite le protocole Git. Celui-ci est géré par un *daemon* spécial livré avec Git. Ce *daemon* (démon, processus en arrière-plan) écoute sur un port dédié (9418) et propose un service similaire au protocole SSH, mais sans aucune sécurisation. Pour qu'un dépôt soit publié via le protocole Git, le fichier `git-daemon-export-ok` 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 pas de pousser des modifications. Vous pouvez activer la capacité à pousser mais étant donné l'absence d'authentification, n'importe qui sur Internet ayant trouvé l'URL du projet peut pousser sur le dépôt. Autant dire que ce mode est rarement recherché.

Avantages

Le protocole Git est souvent le protocole avec la vitesse de transfert la plus rapide. Si vous devez servir un gros trafic pour un projet public ou un très gros projet qui ne nécessite pas d'authentification en lecture, il est très probable que vous devriez installer un *daemon* Git. Il utilise le même mécanisme de transfert de données que SSH, la surcharge du chiffrement et de l'authentification en moins.

Inconvénients

Le défaut du protocole Git est le manque d'authentification. N'utiliser que le protocole Git pour accéder à un projet n'est généralement pas suffisant. Il faut le coupler avec un accès SSH ou HTTPS pour quelques développeurs qui auront le droit de pousser (écrire) et le garder en accès `git://` pour la lecture seule. C'est aussi le protocole le plus difficile à mettre en place. Il doit être géré par son propre *daemon* qui est spécifique. Il nécessite la configuration d'un *daemon* `xinetd` ou apparenté, ce qui est loin d'être simple. Il nécessite aussi un accès à travers le pare-feu au port 9418 qui n'est pas un port ouvert en standard dans les pare-feux professionnels. Derrière les gros pare-feux professionnels, ce port obscur est tout simplement bloqué.

Installation de Git sur un serveur

Nous allons à présent traiter de la configuration d'un service Git gérant ces protocoles sur votre propre serveur.

Les commandes et étapes décrites ci-après s'appliquent à des installations simplifiées sur un serveur à base de Linux, bien qu'il soit aussi possible de faire fonctionner ces services sur des serveurs Mac ou Windows. La mise en place effective d'un serveur en production au sein d'une infrastructure englobera vraisemblablement des

différences dans les mesures de sécurité et les outils système, mais ceci devrait permettre de se faire une idée générale des besoins.

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_projet mon_projet.git
Clonage dans le dépôt nu 'mon_projet.git'...
fait.
```

Vous devriez maintenant avoir une copie des données de Git dans votre répertoire `mon_projet.git`.

C'est grossièrement équivalent à :

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

Il y a quelques légères différences dans le fichier de configuration mais pour l'utilisation envisagée, c'est très proche. La commande extrait le répertoire Git sans répertoire de travail et crée un répertoire spécifique pour l'accueillir.

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 `/srv/git`. En supposant que `/srv/git` existe, vous pouvez mettre en place votre dépôt en copiant le dépôt nu :

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

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

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

Si un utilisateur se connecte via SSH au serveur et a accès en écriture au répertoire `/srv/git/mon_projet.git`, il aura automatiquement accès pour pousser.

Git ajoutera automatiquement les droits de groupe en écriture à un dépôt si vous lancez la commande `git init` avec l'option `--shared`. Notez qu'en lançant cette commande, vous ne détruisez pas les *commits*, références, etc. en cours de route.

```
$ ssh utilisateur@git.exemple.com
$ cd /srv/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.

Il faut noter que c'est littéralement tout ce dont vous avez besoin pour démarrer un serveur Git utile auquel plusieurs personnes ont accès : ajoutez simplement des comptes SSH sur un serveur, et collez un dépôt nu quelque part où tous les utilisateurs ont accès en lecture et écriture. Vous êtes prêts à travailler, vous n'avez besoin de rien d'autre.

Dans les chapitres à venir, nous traiterons de mises en place plus sophistiquées. Ces sujets incluront l'élimination du besoin de créer un compte système pour chaque utilisateur, l'accès public aux dépôts, la mise en place d'interfaces utilisateur web, etc. Néanmoins, gardez à l'esprit que pour collaborer avec quelques personnes sur un projet privé, tout ce qu'il faut, c'est un

serveur SSH et un dépôt nu.

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.

Accès SSH

Si vous disposez déjà d'un serveur auquel tous vos développeurs ont un accès SSH, il est généralement plus facile d'y mettre en place votre premier dépôt car vous n'aurez quasiment aucun réglage supplémentaire à faire (comme nous l'avons expliqué dans le chapitre précédent). Si vous souhaitez des permissions d'accès plus complexes, vous pouvez les mettre en place par le jeu des permissions standards sur le système de fichiers du système d'exploitation de votre serveur.

Si vous souhaitez placer vos dépôts sur un serveur qui ne dispose pas déjà de comptes pour chacun des membres de votre équipe qui aurait accès en écriture, alors vous devrez mettre en place un accès SSH pour eux. En supposant que pour vos dépôts, vous disposiez déjà d'un serveur SSH installé et auquel vous avez accès.

Il y a quelques moyens de donner un accès à tout le monde dans l'équipe. Le premier est de créer des comptes pour tout le monde, ce qui est logique mais peut s'avérer lourd. Vous ne souhaiteriez sûrement pas lancer `adduser` et entrer un mot de passe temporaire pour chaque utilisateur.

Une seconde méthode consiste à créer un seul utilisateur Git sur la machine, demander à chaque développeur nécessitant un accès en écriture de vous envoyer une clé publique SSH et d'ajouter la-dite clé au fichier `~/.ssh/authorized_keys` de votre utilisateur Git. À partir de là, tout le monde sera capable d'accéder à la machine via l'utilisateur Git. Cela n'affecte en rien les données de *commit* — les informations de l'utilisateur SSH par lequel on se connecte n'affectent pas les données de *commit* enregistrées.

Une dernière méthode consiste à faire une authentification SSH auprès d'un serveur LDAP ou tout autre système d'authentification centralisé que vous utiliseriez déjà. Tant que chaque utilisateur peut accéder à un shell sur la machine, n'importe quel schéma d'authentification SSH devrait fonctionner.

Génération des clés publiques SSH

Cela dit, de nombreux serveurs Git utilisent une authentification par clés publiques SSH. Pour fournir une clé 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 clés SSH d'un utilisateur sont stockées dans le répertoire `~/.ssh` du compte. Vous pouvez facilement vérifier si vous avez déjà une clé en listant le contenu de ce répertoire :

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

Recherchez une paire de fichiers appelés *quelquechose* et *quelquechose*.pub où le *quelquechose* en question est généralement `id_dsa` ou `id_rsa`. Le fichier en `.pub` est la clé publique tandis que l'autre est la clé privée. Si vous ne voyez pas ces fichiers (ou n'avez même pas de répertoire `.ssh`), vous pouvez les créer en lançant un programme appelé `ssh-keygen` fourni par le paquet SSH sur les systèmes Linux/Mac et MSysGit pour Windows :

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
```



```
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Premièrement, le programme demande confirmation de l'endroit où vous souhaitez sauvegarder la clé (`.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 le taper quand vous utilisez la clé.

Maintenant, chaque utilisateur ayant suivi ces indications doit envoyer la clé 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 clés publiques). Ils doivent copier le contenu du fichier `.pub` et l'envoyer par courriel. Les clés publiques ressemblent à ceci :

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAKLOUpkDHrfHY17SbrmTIpNLTKG9Tjom/BWDSU
GPl+naFzLHDTYw7hdI4yZ5ew18JH4Jw9jbhUFRviQzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1wXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYnby6vw/Pb0rwert/En
mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Pour un tutoriel plus approfondi sur la création de clé SSH sur différents systèmes d'exploitation, référez-vous au guide GitHub sur les clés SSH à <https://help.github.com/articles/generating-ssh-keys>.

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 supposons 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 && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Ensuite, vous devez ajouter la clé publique d'un développeur au fichier `authorized_keys` de l'utilisateur Git. Supposons que vous avez reçu quelques clés par courriel et les avez sauveées dans des fichiers temporaires. Pour rappel, une clé publique ressemble à ceci :

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRswj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NysnEAZuXz0jTTyAUFrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGw1GYEIGS9Ez
Sdfd8ACCIicTDWbQLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
07TCSBdLq1gmV0Fq1I2uPwQ0K0WQAhuKE0mfjy2jctxSDBQ220ymjaNsHT4kgTzg2AYYgPq
dAv8JggJICUVax2T9va5 gsg-keypair
```

Il suffit de les ajouter au fichier `authorized_keys` de l'utilisateur `git` dans son répertoire `.ssh` :

```
$ 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 project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

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 par shell 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 `gitserveur` sur ce serveur, alors vous pouvez utiliser les commandes suivantes telles quelles (en supposant que `monprojet` est un projet existant et comprenant des fichiers) :

```
# Sur l'ordinateur de John
$ cd monprojet
$ git init
$ git add .
$ git commit -m 'première 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 :

```
$ git clone git@gitserveur:/opt/git/projet.git
$ cd projet
$ vim LISEZMOI
$ git commit -am 'correction du 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.

Il faut aussi noter que pour l'instant tous ces utilisateurs peuvent aussi se connecter au serveur et obtenir un shell en tant qu'utilisateur « `git` ». Si vous souhaitez restreindre ces droits, il faudra changer le shell pour quelque chose d'autre dans le fichier `passwd`.

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 configurez 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. Pour faire cela, vous devez d'abord ajouter `git-shell` à `/etc/shells` s'il n'y est pas déjà :

```
$ cat /etc/shells # voir si `git-shell` est déjà déclaré. Sinon...
$ which git-shell # s'assurer que git-shell est installé sur le système
$ sudo vim /etc/shells # et ajouter le chemin complet vers git-shell
```

Maintenant, vous pouvez éditer le shell de l'utilisateur en utilisant `chsh <utilisateur> -s <shell>` :

```
$ sudo chsh git -s `which git-shell`
```

À présent, l'utilisateur `git` ne peut plus utiliser la connexion SSH que pour pousser et tirer sur des dépôts Git, il ne peut plus ouvrir un shell. Si vous essayez, vous verrez un rejet de login :

```
$ ssh git@gitserveur
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserveur closed.
```

Maintenant, les commandes réseau Git continueront de fonctionner correctement mais les utilisateurs ne pourront plus obtenir de shell. Comme la sortie l'indique, vous pouvez aussi configurer un répertoire dans le répertoire personnel de l'utilisateur « git » qui va personnaliser légèrement le `git-shell`. Par exemple, vous pouvez restreindre les commandes Git que le serveur accepte ou vous pouvez personnaliser le message que les utilisateurs verront s'ils essaient de se connecter en SSH comme ci-dessus. Lancer `git help shell` pour plus d'informations sur la personnalisation du shell.

Démon (*Daemon*) Git

Dans la suite, nous allons configurer un *daemon* qui servira des dépôts sur le protocole « Git ». C'est un choix répandu pour permettre un accès rapide sans authentification à vos données Git. Souvenez-vous que du fait de l'absence d'authentification, tout ce qui est servi sur ce protocole est public au sein de son réseau.

Mis en place sur un serveur à l'extérieur de votre pare-feu, il ne devrait être utilisé que pour des projets qui sont destinés à être visibles publiquement par le monde entier. Si le serveur est derrière le pare-feu, il peut être utilisé pour des projets avec accès en lecture seule pour un grand nombre d'utilisateurs ou des ordinateurs (intégration continue ou serveur de compilation) pour lesquels vous ne souhaitez pas avoir à gérer des clés SSH.

En tout cas, le protocole Git est relativement facile à mettre en place. Grossièrement, il suffit de lancer la commande suivante en tant que *daemon* :

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` autorise le serveur à redémarrer sans devoir attendre que les anciennes connexions expirent, l'option `--base-path` autorise les utilisateurs à cloner des projets sans devoir spécifier le chemin complet, et le chemin en fin de ligne indique au *daemon* Git l'endroit où chercher des dépôts à exporter. Si vous utilisez un pare-feu, il sera nécessaire de rediriger le port 9418 sur la machine hébergeant le serveur.

Transformer ce processus en *daemon* peut s'effectuer de différentes manières qui dépendent du système d'exploitation sur lequel il est lancé. Sur une machine Ubuntu, c'est un script Upstart. Donc dans le fichier :

```
/etc/event.d/local-git-daemon
```

mettez le script suivant :

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
  --user=git --group=git \
  --reuseaddr \
  --base-path=/opt/git/ \
  /opt/git/
respawn
```

Par sécurité, ce *daemon* devrait être lancé par un utilisateur n'ayant que des droits de lecture seule sur les dépôts — simplement en créant un nouvel utilisateur « git-ro » qui servira à lancer le *daemon*. Par simplicité, nous le lancerons avec le même utilisateur « git » qui est utilisé par `git-shell`.

Au redémarrage de la machine, votre *daemon* Git démarrera automatiquement et redémarrera s'il meurt. Pour le lancer sans avoir à redémarrer, vous pouvez lancer ceci :

```
initctl start local-git-daemon
```

Sur d'autres systèmes, le choix reste large, allant de `xinetd` à un script de système `sysvinit` ou à tout autre moyen — tant que le programme est démonisé et surveillé.

Ensuite, il faut spécifier à Git quels dépôts sont autorisés en accès non authentifié au moyen du serveur. Dans chaque dépôt concerné, il suffit de créer un fichier appelé `git-daemon-export-ok`.

```
$ cd /chemin/au/projet.git
$ touch git-daemon-export-ok
```

La présence de ce fichier indique à Git que ce projet peut être servi sans authentification.

HTTP intelligent

Nous avons à présent un accès authentifié par SSH et un accès non authentifié par `git://`, mais il existe aussi un protocole qui peut faire les deux à la fois. La configuration d'un HTTP intelligent revient simplement à activer sur le serveur un script CGI livré avec Git qui s'appelle `git-http-backend`. Ce CGI va lire le chemin et les entêtes envoyés par un `git fetch` ou un `git push` à une URL donnée et déterminer si le client peut communiquer sur HTTP (ce qui est vrai pour tout client depuis la version 1.6.6). Si le CGI détecte que le client est intelligent, il va commencer à communiquer par protocole intelligent, sinon il repassera au comportement du protocole idiot (ce qui le rend de ce fait compatible avec les vieux clients).

Détaillons une installation de base. Nous la réaliserons sur un serveur web Apache comme serveur CGI. Si Apache n'est pas installé sur votre PC, vous pouvez y remédier avec une commande :

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

Cela a aussi pour effet d'activer les modules `mod_cgi`, `mod_alias`, et `mod_env` qui sont nécessaires au fonctionnement du serveur.

Ensuite, nous devons ajouter quelques lignes à la configuration d'Apache pour qu'il lance `git-http-backend` comme gestionnaire de tous les chemins du serveur web sous `/git`.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Si vous ne définissez pas la variable d'environnement `GIT_HTTP_EXPORT_ALL`, Git ne servira aux utilisateurs non authentifiés que les dépôts comprenant le fichier `git-daemon-export-ok`, de la même manière que le *daemon* Git.

Puis, nous allons indiquer à Apache qu'il doit accepter les requêtes sur ce chemin avec quelque chose comme :

```
<Directory "/usr/lib/git-core*">
    Options ExecCGI Indexes
    Order allow,deny
    Allow from all
    Require all granted
</Directory>
```

Enfin, il faut forcer l'authentification pour l'écriture, probablement avec un bloc `Auth` comme celui-ci :

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
</LocationMatch>
```

Il faudra donc créer un fichier `.htaccess` contenant les mots de passe de tous les utilisateurs valides. Voici un exemple d'ajout d'un utilisateur `schacon` au fichier :

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

Il existe des milliers de façons d'authentifier des utilisateurs avec Apache, il suffira d'en choisir une et de la mettre en place. L'exemple ci-dessus n'est que le plus simple. Vous désirerez sûrement gérer tout ceci sous SSL pour que vos données soient chiffrées.

Nous ne souhaitons pas nous appesantir spécifiquement sur la configuration d'Apache, car on peut utiliser un serveur différent ou avoir besoin d'une authentification différente. L'idée générale reste que Git est livré avec un CGI appelé `git-http-backend` qui, après authentification, va gérer toute la négociation pour envoyer et recevoir les données sur HTTP. Il ne gère pas l'authentification par lui-même, mais peut être facilement contrôlé à la couche serveur web qui l'invoque. Cela peut être réalisé avec n'importe quel serveur web gérant le CGI, donc celui que vous connaissez le mieux.

Pour plus d'informations sur la configuration de l'authentification dans Apache, référez-vous à la documentation d'Apache : <http://httpd.apache.org/docs/current/howto/auth.html>

GitWeb

Après avoir réglé les accès de base en lecture/écriture et en lecture seule pour vos projets, vous souhaitez 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.

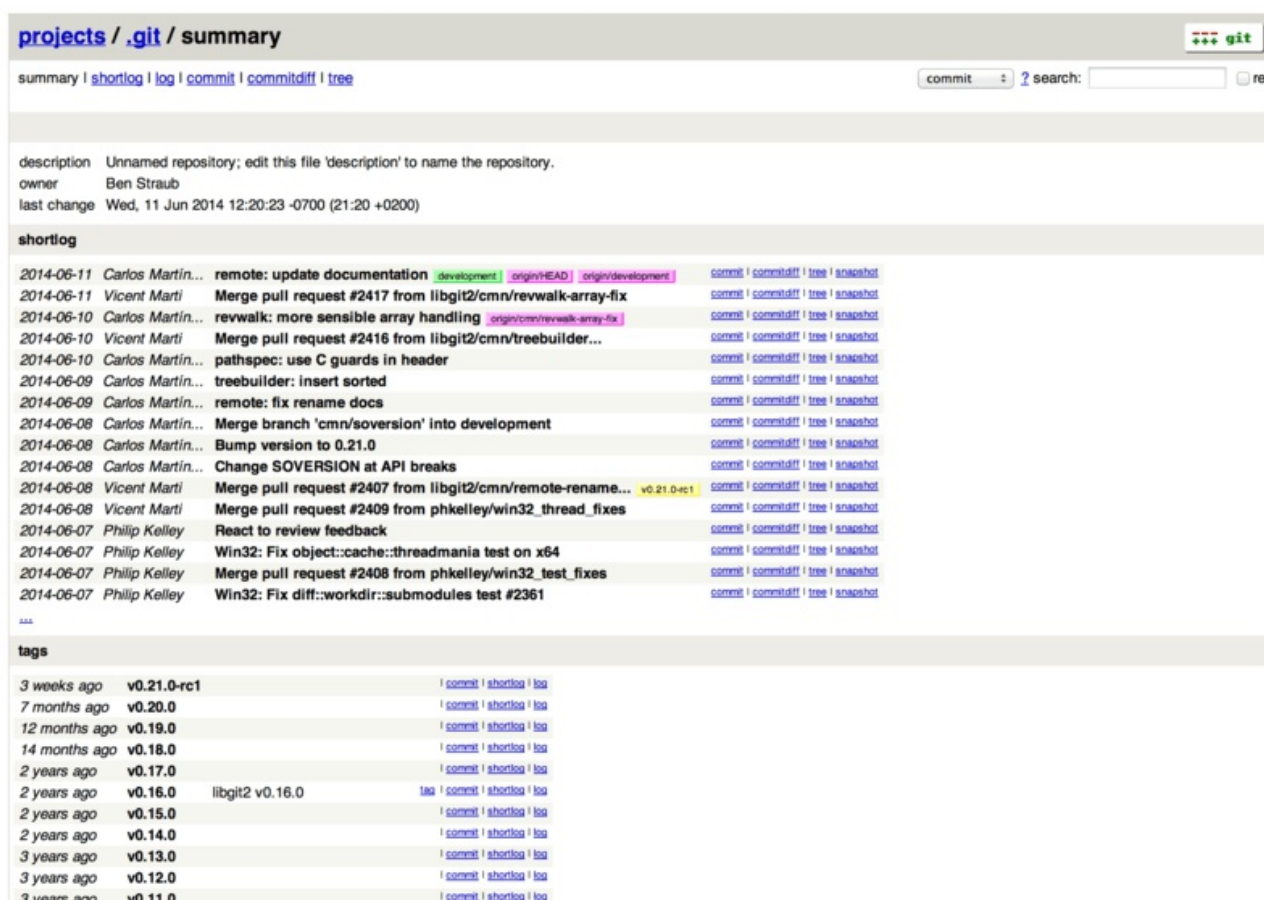


Figure 50 : L'interface web de visualisation Gitweb.

Figure 49. L'interface web de visualisation 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 pré-installé 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]
```

Cette commande démarre un serveur HTTP sur le port 1234 et lance automatiquement un navigateur Internet qui ouvre la page d'accueil. C'est vraiment très simple. Pour arrêter le serveur, il suffit de lancer la même commande, mais avec l'option `--stop` :

```
$ git instaweb --httpd=webrick --stop
```

Si vous souhaitez fournir l'interface web en permanence sur le serveur pour votre équipe ou pour un projet opensource que vous hébergez, il sera nécessaire d'installer le script CGI pour qu'il soit appelé par votre serveur web. Quelques distributions Linux ont un package `gitweb` qu'il suffira d'installer via `apt` ou `dnf`, ce qui est une possibilité. Nous détaillerons tout de même rapidement l'installation manuelle de GitWeb. Premièrement, le code source de Git qui fournit GitWeb est nécessaire pour pouvoir générer un script CGI personnalisé :

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
  SUBDIR gitweb
  SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
  GEN gitweb.cgi
  GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Notez que vous devez indiquer où trouver les dépôts Git au moyen de la variable `GITWEB_PROJECTROOT`. Maintenant, il faut paramétrer dans Apache l'utilisation de CGI pour ce script, en spécifiant un nouveau VirtualHost :

```
<VirtualHost *:80>
  ServerName gitserver
  DocumentRoot /var/www/gitweb
  <Directory /var/www/gitweb>
    Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
    AllowOverride All
    order allow,deny
    Allow from all
    AddHandler cgi-script cgi
    DirectoryIndex gitweb.cgi
  </Directory>
</VirtualHost>
```

Une fois de plus, GitWeb peut être géré par tout serveur web capable de prendre en charge CGI ou Perl. La mise en place ne devrait pas être plus difficile avec un autre serveur. Après redémarrage du serveur, vous devriez être capable de visiter `http://gitserver/` pour visualiser vos dépôts en ligne.

GitLab

GitWeb reste tout de même simpliste. Si vous cherchez un serveur Git plus moderne et complet, il existe quelques solutions libres pertinentes. Comme GitLab est un des plus populaires, nous allons prendre son installation et son utilisation comme exemple. Cette solution est plus complexe que l'option GitWeb et demandera indubitablement plus de maintenance, mais

elle est aussi plus complète.

Installation

GitLab est une application web reposant sur une base de données, ce qui rend son installation un peu plus lourde que certains autres serveurs Git. Celle-ci est heureusement très bien documentée et supportée.

GitLab peut s'installer de différentes manières. Pour obtenir rapidement quelque chose qui tourne, vous pouvez télécharger une image de machine virtuelle ou un installateur rapide depuis <https://bitnami.com/stack/gitlab>, puis configurer plus finement selon vos besoins. Une touche particulière incluse par Bitnami concerne l'écran d'identification (accessible via `alt + -`) qui vous indique l'adresse IP, l'utilisateur et le mot de passe par défaut de l'instance GitLab installée.



Figure 51 : L'écran d'identification de la machine virtuelle du GitLab de Bitnami.

Figure 50. L'écran d'identification de la machine virtuelle du GitLab de Bitnami.

Pour toute autre méthode, suivez les instructions du readme du *GitLab Community Edition*, qui est consultable à <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Vous y trouverez une aide pour installer GitLab en utilisant une recette Chef, une machine virtuelle sur Digital Ocean, ou encore via RPM ou DEB (qui, au moment de la rédaction du présent livre sont en bêta). Il existe aussi des guides « non-officiels » pour faire fonctionner GitLab avec des systèmes d'exploitation ou de base données non standards, un script d'installation totalement manuel et d'autres guides couvrant d'autres sujets.

Administration

L'interface d'administration de GitLab passe par le web. Pointez simplement votre navigateur sur le nom d'hôte ou l'adresse IP où GitLab est hébergé et identifiez-vous comme administrateur. L'utilisateur par défaut est `admin@local.host` et le mot de passe par défaut est `5iveL!fe` (qu'il vous sera demandé de changer dès la première connexion). Une fois identifié, cliquez sur l'icône « Admin area » dans le menu en haut à droite.

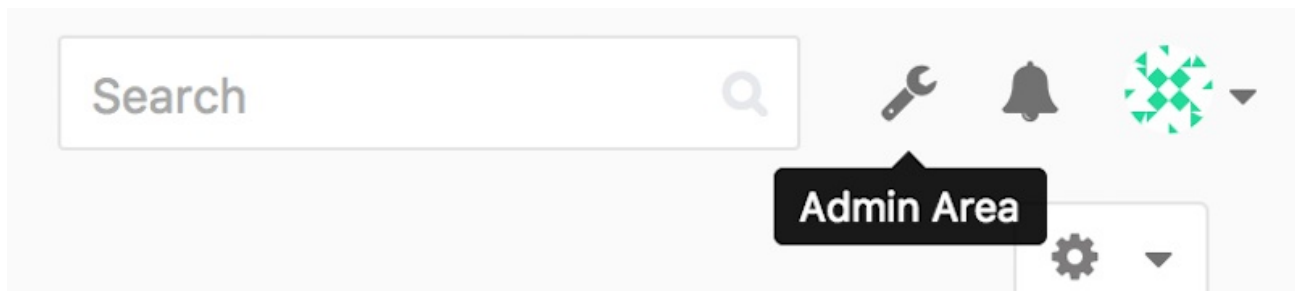


Figure 52 : L'entrée « Admin area » dans le menu GitLab.

Figure 51. L'entrée « Admin area » dans le menu GitLab.

Utilisateurs

Les utilisateurs dans GitLab sont des comptes qui correspondent à des personnes. Les comptes utilisateurs ne sont pas très complexes ; ce sont principalement des collections d'informations personnelles rattachées à chaque information d'identification. Chaque compte utilisateur fournit un **espace de nommage**, qui est un rassemblement logique des projets appartenant à cet utilisateur. Si l'utilisateur `jane` a un projet appelé `projet`, l'URL du projet est `http://serveur/jane/projet`.

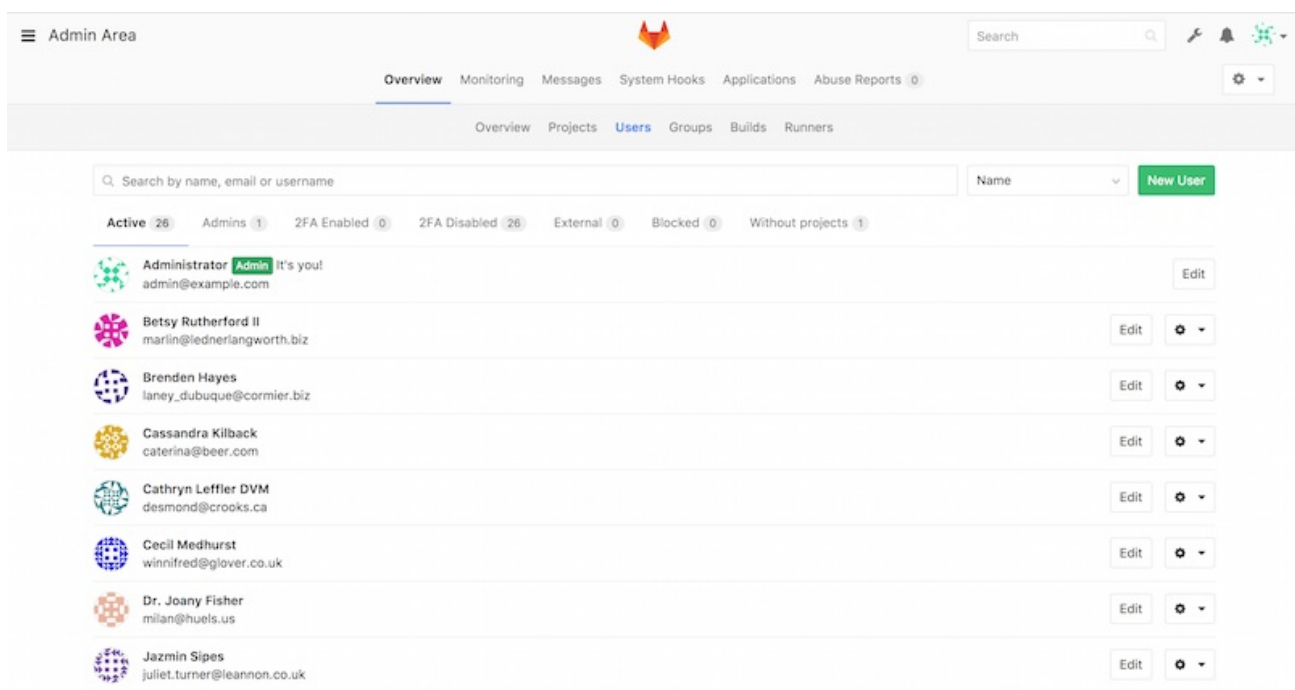


Figure 53 : L'écran d'administration des utilisateurs GitLab.

Figure 52. L'écran d'administration des utilisateurs GitLab.

Il existe deux manières de supprimer un utilisateur. Bloquer (`blocking`) un utilisateur l'empêche de s'identifier sur l'instance Gitlab, mais toutes les données sous l'espace de nom de cet utilisateur sont préservées, et les commits signés avec l'adresse courriel de cet utilisateur renverront à son profil.

Détruire (`destroying`) un utilisateur, par contre, l'efface complètement de la base de données et du système de fichiers. Tous les projets et les données situées dans son espace de nom sont effacés et tous les groupes qui lui appartiennent sont aussi effacés. Il s'agit clairement d'une action plus destructive et permanente, et son usage est assez rare.

Groupes

Un groupe GitLab est un assemblage de projets, accompagné des informations de droits d'accès à ces projets. Chaque groupe a un espace de nom de projet (de la même manière que les utilisateurs), donc si le groupe `formation` a un projet `matériel`, son URL sera `http://serveur/formation/matériel`.

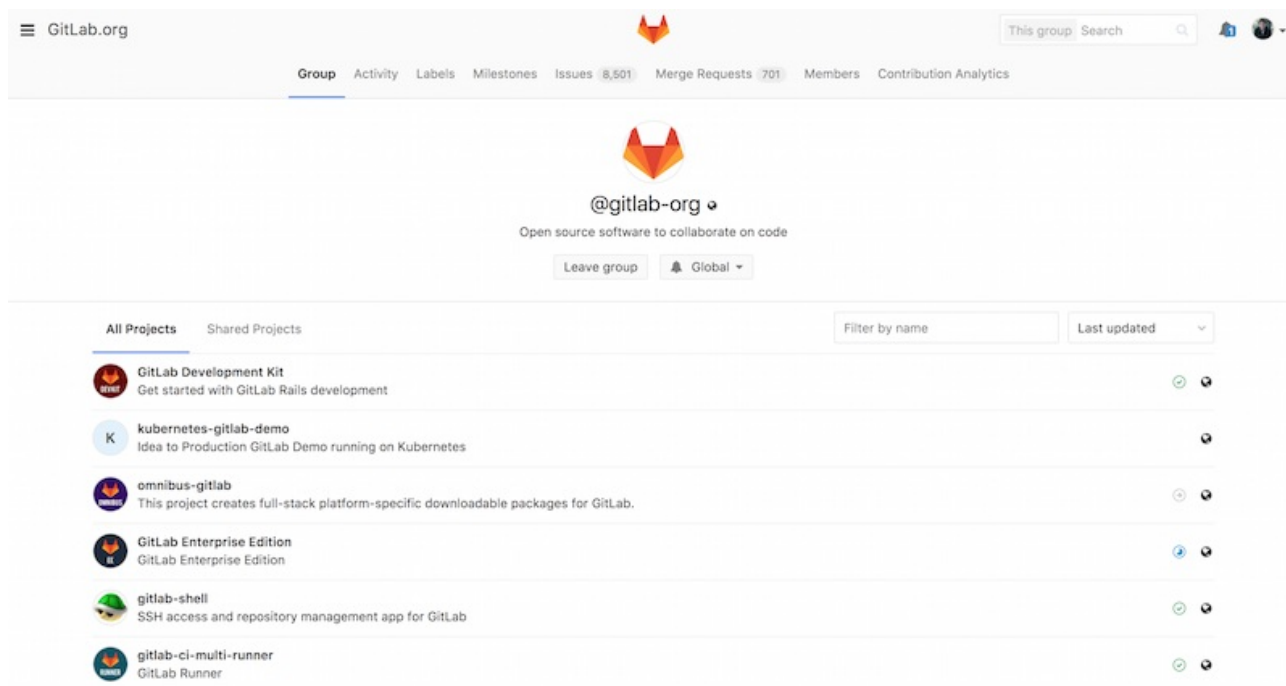


Figure 54 : L'écran d'administration des groupes GitLab.

Figure 53. L'écran d'administration des groupes GitLab.

Chaque groupe est associé à des utilisateurs, dont chacun dispose d'un niveau de permissions sur les projets du groupe et sur le groupe lui-même. Ces niveaux s'échelonnent de *invité* : `Guest` (tickets et discussions seulement) à *propriétaire* : `Owner` (contrôle complet du groupe, ses membres et ses projets). Les types de permissions sont trop nombreux pour être énumérés ici, mais GitLab fournit un lien très utile sur son écran d'administration.

Projets

Un projet GitLab correspond grossièrement à un dépôt Git unique. Tous les projets appartiennent à un espace de nom unique, que ce soit un utilisateur ou un groupe. Si le projet appartient à un utilisateur, le propriétaire du projet contrôle directement les droits d'accès au projet ; si le projet appartient à un groupe, le niveau de permission de l'utilisateur pour le groupe est aussi pris en compte.

Tous les projets ont un niveau de visibilité qui permet de contrôler qui a accès en lecture aux pages et au dépôt de ce projet. Si un projet est privé (*Private*), l'accès au projet doit être explicitement accordé par le propriétaire du projet à chaque utilisateur. Un projet interne (*Internal*) est visible par tout utilisateur identifié, et un projet public (*Public*) est un projet visible par tout le monde. Notez que ces droits contrôlent aussi bien les accès pour git `fetch` que les accès à l'interface utilisateur web du projet.

Crochets (Hooks)

GitLab inclut le support pour les crochets, tant au niveau projet que système. Pour ces deux niveaux, le serveur GitLab lance des requêtes HTTP POST contenant un JSON de description lorsque certains événements précis arrivent. C'est un excellent moyen de connecter vos dépôts Git et votre instance GitLab avec le reste de vos automatisations de développement, telles que serveurs d'intégration continue, forum de discussion et outils de déploiement.

Usage de base

La première chose à faire avec GitLab est de créer un nouveau projet. Pour cela, il suffit de cliquer sur l'icône **+** sur la barre d'outils. On vous demande le nom du projet, à quel espace de nom il appartient, et son niveau de visibilité. La plupart des configurations demandées ici ne sont pas permanentes et peuvent être réajustées plus tard grâce à l'interface de paramétrage. Cliquez sur **Create Project** pour achever la création.

Une fois le projet créé, on peut le connecter à un dépôt Git local. Chaque projet est accessible sur HTTPS ou SSH, qui peuvent donc être utilisés pour un dépôt distant. Les URLs sont visibles en haut de la page du projet. Pour un dépôt local existant, cette commande crée un dépôt distant nommé `gitlab` pointant vers l'hébergement distant :

```
$ git remote add gitlab https://serveur/espace_de_nom/projet.git
```

Si vous n'avez pas de copie locale du dépôt, vous pouvez simplement taper ceci :

```
$ git clone https://serveur/espace_de_nom/projet.git
```

L'interface utilisateur web donne accès à différentes vues utiles du dépôt lui-même. La page d'accueil de chaque projet montre l'activité récente et des liens alignés en haut vous mènent aux fichiers du projet et au journal des *commits*.

Coopérer

Le moyen le plus simple de coopérer sur un projet GitLab consiste à donner à un autre utilisateur un accès direct en écriture sur le dépôt Git. Vous pouvez ajouter un utilisateur à un projet en sélectionnant la section **Members** des paramètres du projet et en associant le nouvel utilisateur à un niveau d'accès (les différents niveaux d'accès sont abordés dans [Groupes](#)). En donnant un niveau d'accès **Developer** ou plus à un utilisateur, cet utilisateur peut pousser des *commits* et des branches directement sur le dépôt sans restriction.

Un autre moyen plus découplé de collaborer est d'utiliser des requêtes de tirage (*pull request*). Cette fonction permet à n'importe quel utilisateur qui peut voir le projet d'y contribuer de manière contrôlée. Les utilisateurs avec un accès direct peuvent simplement créer une branche, pousser des *commits* dessus et ouvrir une requête de tirage depuis leur branche vers `master` ou toute autre branche. Les utilisateurs qui n'ont pas la permission de pousser sur un dépôt peuvent en faire un *fork* (créer leur propre copie), pousser des *commits* sur cette copie et ouvrir une requête de tirage depuis leur *fork* vers le projet principal. Ce modèle permet au propriétaire de garder le contrôle total sur ce qui entre dans le dépôt et quand, tout en autorisant les contributions des utilisateurs non fiables.

Les requêtes de fusion (*merge requests*) et les problèmes (*issues*) sont les principaux moyens pour mener des discussions au long cours dans GitLab. Chaque requête de fusion permet une discussion ligne par ligne sur les modifications proposées (qui permettent un sorte de revue de code légère), ainsi qu'un fil de discussion général. Requêtes de fusion et problèmes peuvent être assignés à des utilisateurs ou assemblés en jalons (*milestones*).

Cette section se concentre principalement sur les parties de GitLab dédiées à Git, mais c'est un système assez mature qui fournit beaucoup d'autres fonctions qui peuvent aider votre équipe à coopérer. Parmi celles-ci figurent les wikis, les murs de discussion et des outils de maintenance du système. Un des bénéfices de GitLab est que, une fois le serveur paramétré et en marche, vous n'aurez pas besoin de bricoler un fichier de configuration ou d'accéder au serveur via SSH ; la plupart des tâches générales ou d'administration peuvent se réaliser à travers l'interface web.

Git hébergé

Si vous ne vous ne voulez pas vous investir dans la mise en place de votre propre serveur Git, il reste quelques options pour héberger vos projets Git sur un site externe dédié à l'hébergement. Cette méthode offre de nombreux avantages : un site en hébergement est généralement rapide à créer et facilite le démarrage de projets, et n'implique pas de maintenance et de

surveillance de serveur. Même si vous montez et faites fonctionner votre serveur en interne, vous souhaiterez sûrement utiliser un site d'hébergement public pour votre code open source — cela rend généralement plus facile l'accès et l'aide par la communauté.

Aujourd'hui, vous avez à disposition un nombre impressionnant d'options d'hébergement, chacune avec différents avantages et inconvénients. Pour une liste à jour, référez-vous à la page GitHosting sur le wiki principal de Git :

<https://git.wiki.kernel.org/index.php/GitHosting>.

Nous traiterons de l'utilisation de GitHub en détail dans [GitHub](#) du fait que c'est le plus gros hébergement de Git sur Internet et que vous pourriez avoir besoin d'y interagir pour des projets hébergés à un moment, mais il existe aussi d'autres plateformes d'hébergement si vous ne souhaitez pas mettre en place votre propre serveur Git.

Résumé

Vous disposez de plusieurs moyens de mettre en place un dépôt Git distant pour pouvoir collaborer avec d'autres et partager votre travail.

Gérer votre propre serveur vous donne une grande maîtrise et vous permet de l'installer derrière un pare-feu, mais un tel serveur nécessite généralement une certaine quantité de travail pour l'installation et la maintenance. Si vous placez vos données sur un serveur hébergé, c'est très simple à installer et maintenir. Cependant vous devez pouvoir héberger votre code sur des serveurs tiers et certaines politiques d'organisation ne le permettent pas.

Choisir la meilleure solution ou combinaison de solutions pour votre cas ou celui de votre société ne devrait pas poser de problème.

Git distribué

- [Développements distribués](#)
 - [Gestion Centralisée](#)
 - [Mode du gestionnaire d'intégration](#)
 - [Mode dictateur et ses lieutenants](#)
 - [Résumé](#)
- [Contribution à un projet](#)
 - [Guides pour une validation](#)
 - [Cas d'une petite équipe privée](#)
 - [Équipe privée importante](#)
 - [Projet public dupliqué](#)
 - [Projet public via courriel](#)
 - [Résumé](#)
- [Maintenance d'un projet](#)
 - [Travail dans des branches thématiques](#)
 - [Application des patches à partir de courriel](#)
 - [Application d'un patch avec `apply`](#)
 - [Application d'un patch avec `am`](#)
 - [Vérification des branches distantes](#)
 - [Déterminer les modifications introduites](#)
 - [Intégration des contributions](#)
 - [Modes de fusion](#)
 - [Gestions avec nombreuses fusions](#)
 - [Gestion par rebasage et sélection de `commit`](#)
 - [Rerere](#)
 - [Étiquetage de vos publications](#)
 - [Génération d'un nom de révision](#)
 - [Préparation d'une publication](#)
 - [Shortlog](#)
- [Résumé](#)

Avec un dépôt distant Git mis en place pour permettre à tous les développeurs de partager leur code, et la connaissance des commandes de base de Git pour une gestion locale, abordons les méthodes de gestion distribuée que Git nous offre.

Dans ce chapitre, vous découvrirez comment travailler dans un environnement distribué avec Git en tant que contributeur ou comme intégrateur. Cela recouvre la manière de contribuer efficacement à un projet et de rendre la vie plus facile au mainteneur du projet ainsi qu'à vous-même, mais aussi en tant que mainteneur, de gérer un projet avec de nombreux contributeurs.

Développements distribués

À la différence des systèmes de gestion de version centralisés (CVCS), la nature distribuée de Git permet une bien plus grande flexibilité dans la manière dont les développeurs collaborent sur un projet. Dans les systèmes centralisés, tout développeur est un nœud travaillant de manière plus ou moins égale sur un concentrateur central. Dans Git par contre, tout développeur est potentiellement un nœud et un concentrateur, c'est-à-dire que chaque développeur peut à la fois contribuer du code vers les autres dépôts et maintenir un dépôt public sur lequel d'autres vont baser leur travail et auquel ils vont contribuer. Cette capacité ouvre une perspective de modes de développement pour votre projet ou votre équipe dont certains archétypes tirant parti de cette flexibilité seront traités dans les sections qui suivent. Les avantages et inconvénients éventuels de chaque mode seront traités. Vous pouvez choisir d'en utiliser un seul ou de mélanger les fonctions de chacun.

Gestion Centralisée

Dans les systèmes centralisés, il n'y a généralement qu'un seul modèle de collaboration, la gestion centralisée. Un concentrateur ou dépôt central accepte le code et tout le monde doit synchroniser son travail avec. Les développeurs sont des nœuds, des consommateurs du concentrateur, seul endroit où ils se synchronisent.

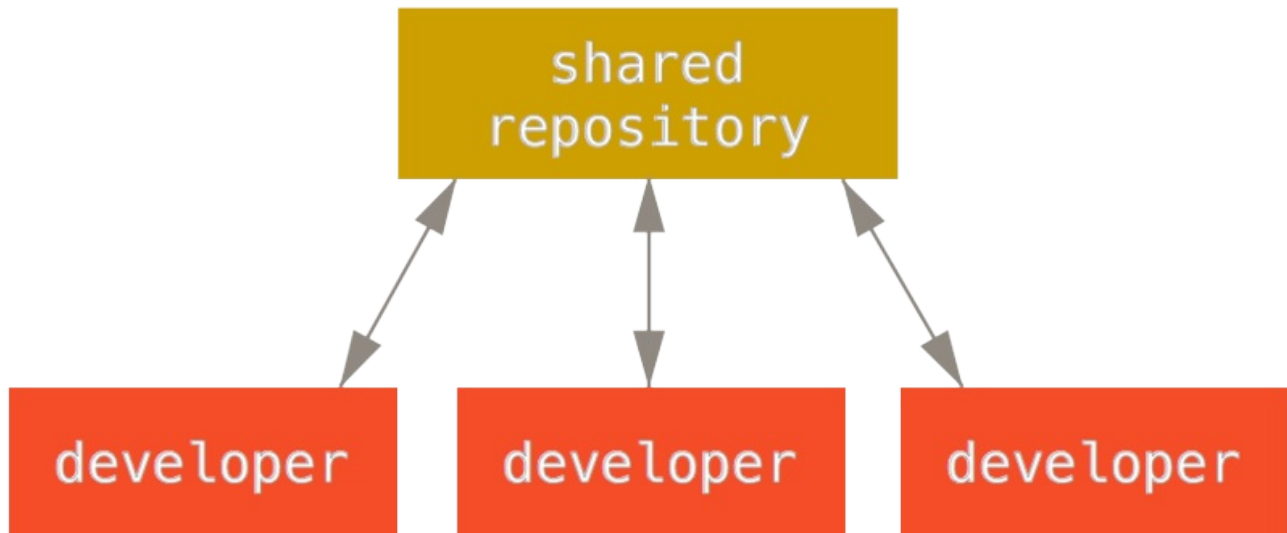


Figure 55 : Gestion centralisée.

Figure 54. Gestion centralisée.

Cela signifie que si deux développeurs clonent depuis le concentrateur et qu'ils introduisent tous les deux des modifications, le premier à pousser ses modifications le fera sans encombre. Le second développeur doit fusionner les modifications du premier dans son dépôt local avant de pousser ses modifications pour ne pas écraser les modifications du premier. Ce concept reste aussi vrai avec Git qu'il l'est avec Subversion (ou tout autre CVCS) et le modèle fonctionne parfaitement dans Git.

Si vous êtes déjà habitué à une gestion centralisée dans votre société ou votre équipe, vous pouvez simplement continuer à utiliser cette méthode avec Git. Mettez en place un dépôt unique et donnez à tous l'accès en poussée. Git empêchera les utilisateurs d'écraser le travail des autres. Supposons que John et Jessica commencent en même temps une tâche. John la termine et pousse ses modifications sur le serveur. Puis Jessica essaie de pousser ses modifications, mais le serveur les rejette. Il lui indique qu'elle tente de pousser des modifications sans avance rapide et qu'elle ne pourra le faire tant qu'elle n'aura pas récupéré et fusionné les nouvelles modifications depuis le serveur. Cette méthode est très intéressante pour de nombreuses personnes car c'est un paradigme avec lequel beaucoup sont familiarisés et à l'aise.

Ce modèle n'est pas limité aux petites équipes. Avec le modèle de branchement de Git, des centaines de développeurs peuvent travailler harmonieusement sur un unique projet au travers de dizaines de branches simultanées.

Mode du gestionnaire d'intégration

Comme Git permet une multiplicité de dépôts distants, il est possible d'envisager un mode de fonctionnement où chaque développeur a un accès en écriture à son propre dépôt public et en lecture à tous ceux des autres. Ce scénario inclut souvent un dépôt canonique qui représente le projet « officiel ». Pour commencer à contribuer au projet, vous créez votre propre clone public du projet et poussez vos modifications dessus. Après, il suffit d'envoyer une demande au mainteneur de projet pour qu'il tire vos modifications dans le dépôt canonique. Il peut ajouter votre dépôt comme dépôt distant, tester vos modifications localement, les fusionner dans sa branche et les pousser vers le dépôt public. Le processus se passe comme ceci (voir [Le mode du 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.

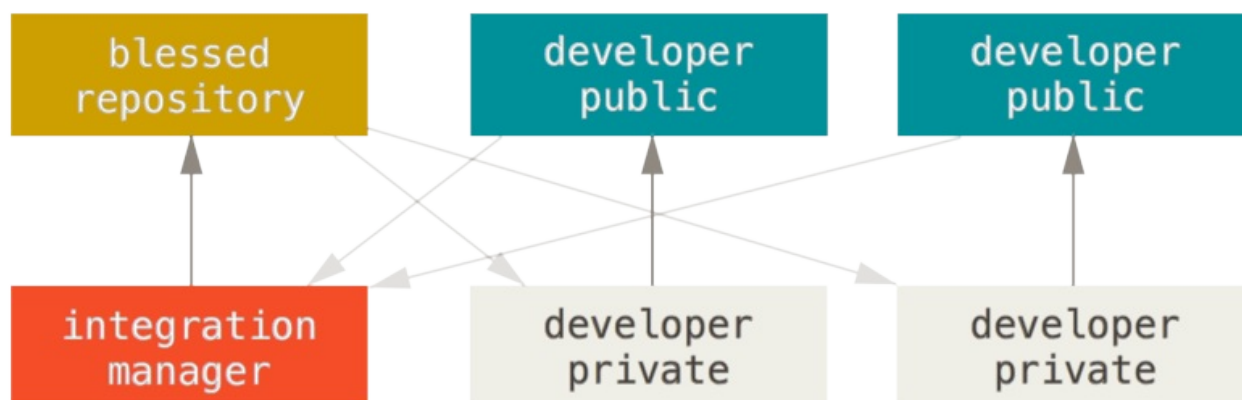


Figure 56 : Le mode du gestionnaire d'intégration.

Figure 55. Le mode du gestionnaire d'intégration.

C'est une gestion très commune sur des sites « échangeurs » tels que GitHub ou GitLab où il est aisé de dupliquer un projet et de pousser ses modifications pour les rendre publiques. Un avantage distinctif de cette approche est qu'il devient possible de continuer à travailler et que le mainteneur du dépôt principal peut tirer les modifications à tout moment. Les contributeurs n'ont pas à attendre le bon vouloir du mainteneur pour incorporer leurs modifications. Chaque acteur peut travailler à son rythme.

Mode dictateur et ses lieutenants

C'est une variante de la gestion multi-dépôt. En général, ce mode est utilisé sur des projets immenses comprenant des centaines de collaborateurs. Un exemple célèbre est le noyau Linux. Des gestionnaires d'intégration gèrent certaines parties du projet. Ce sont les lieutenants. Tous les lieutenants ont un unique gestionnaire d'intégration, le dictateur bienveillant. Le dépôt du dictateur sert de dépôt de référence à partir duquel tous les collaborateurs doivent tirer. Le processus se déroule comme suit (voir [Le processus du dictateur bienveillant](#)) :

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

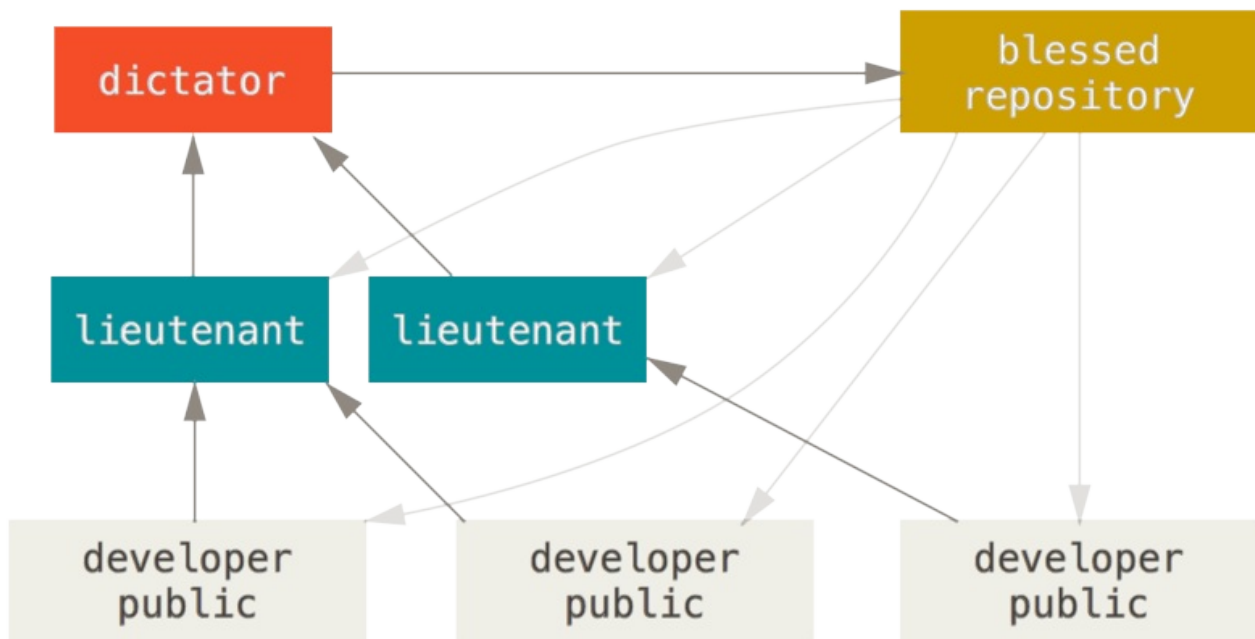


Figure 57 : Le processus du dictateur bienveillant.

Figure 56. Le processus du dictateur bienveillant.

Ce schéma de processus n'est pas très utilisé mais s'avère utile dans des projets très gros ou pour lesquels un ordre hiérarchique existe, car il permet au chef de projet (le dictateur) de déléguer une grande partie du travail et de collecter de grands sous-ensembles de codes à différents points avant de les intégrer.

Résumé

Voilà donc quelques-uns des flux de travail les plus utilisés avec un système distribué tel que Git, mais on voit que de nombreuses variations sont possibles pour mieux correspondre à un mode de gestion réel. À présent que vous avez pu déterminer le mode de gestion qui s'adapte à votre cas, nous allons traiter des exemples spécifiques détaillant comment remplir les rôles principaux constituant chaque mode. Dans le chapitre suivant, nous traiterons de quelques modèles d'activité pour la contribution à un projet.

Contribution à un projet

La principale difficulté à décrire ce processus réside dans l'extraordinaire quantité de variations dans sa réalisation. Comme Git est très flexible, les gens peuvent collaborer de différentes façons et ils le font, et il devient problématique de décrire de manière unique comment devrait se réaliser la contribution à un projet. Chaque projet est légèrement différent. Les variables incluent la taille du corps des contributeurs, le choix du flux de gestion, les accès en validation et la méthode de contribution externe.

La première variable est la taille du corps de contributeurs. Combien de personnes contribuent activement du code sur ce projet et à quelle vitesse ? Dans de nombreux cas, vous aurez deux à trois développeurs avec quelques validations par jour, voire moins pour des projets endormis. Pour des sociétés ou des projets particulièrement grands, le nombre de développeurs peut être de plusieurs milliers, avec des centaines ou des milliers de patches ajoutés chaque jour. Ce cas est important car avec de plus en plus de développeurs, les problèmes de fusion et d'application de patch deviennent de plus en plus courants. Les modifications soumises par un développeur peuvent être obsolètes ou impossibles à appliquer à cause de changements qui ont eu lieu dans l'intervalle de leur développement, de leur approbation ou de leur application. Comment dans ces conditions conserver son code en permanence synchronisé et ses patches valides ?

La variable suivante est le mode de gestion utilisé pour le projet. Est-il centralisé avec chaque développeur ayant un accès égal en écriture sur la ligne de développement principale ? Le projet présente-t-il un mainteneur ou un gestionnaire d'intégration qui vérifie tous les patches ? Tous les patches doivent-ils subir une revue de pair et une approbation ? Faites-vous partie du processus ? Un système à lieutenants est-il en place et doit-on leur soumettre les modifications en premier ?

La variable suivante est la gestion des accès en écriture. Le mode de gestion nécessaire à la contribution au projet est très différent selon que vous avez ou non accès au dépôt en écriture. Si vous n'avez pas accès en écriture, quelle est la méthode préférée pour la soumission de modifications ? Y a-t-il seulement une politique en place ? Quelle est la quantité de modifications fournie à chaque fois ? Quelle est la périodicité de contribution ?

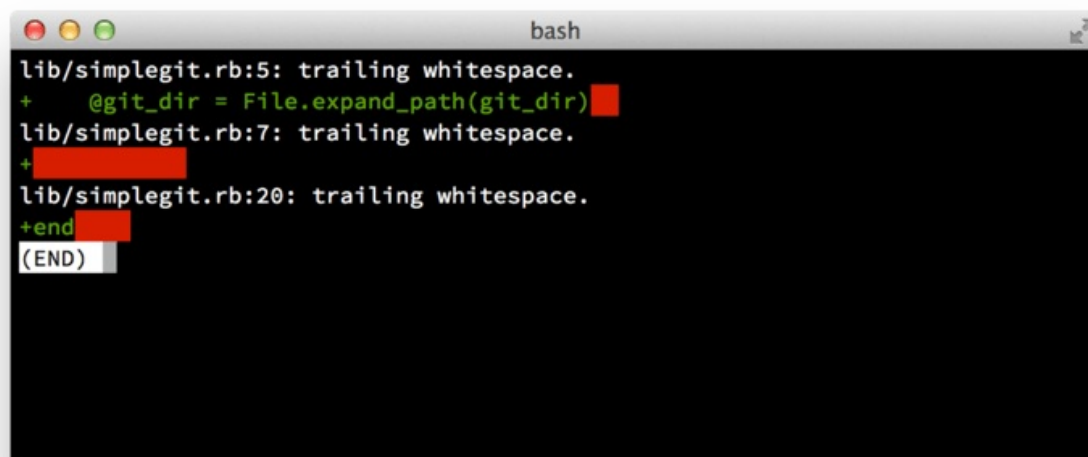
Toutes ces questions affectent la manière de contribuer efficacement à un projet et les modes de gestion disponibles ou préférables. Je vais traiter ces sujets dans une série de cas d'utilisation allant des plus simples aux plus complexes. Vous devriez pouvoir construire vos propres modes de gestion à partir de ces exemples.

Guides pour une validation

Avant de passer en revue les cas d'utilisation spécifiques, voici un point rapide sur les messages de validation. La définition et l'utilisation d'une bonne ligne de conduite sur les messages de validation facilitent grandement l'utilisation de Git et la collaboration entre développeurs. Le projet Git fournit un document qui décrit un certain nombre de bonnes pratiques pour créer des *commits* qui serviront à fournir des patches — le document est accessible dans les sources de Git, dans le fichier

`Documentation/SubmittingPatches`.

Premièrement, il ne faut pas soumettre de patches comportant des erreurs d'espace (caractères espace inutiles en fin de ligne ou entrelacement d'espaces et de tabulations). Git fournit un moyen simple de le vérifier — avant de valider, lancez la commande `git diff --check` qui identifiera et listera les erreurs d'espace.



```
bash
lib/simplegit.rb:5: trailing whitespace.
+   @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Figure 58 : Sortie de `\`git diff --check\``.

Figure 57. Sortie de `git diff --check`.

En lançant cette commande avant chaque validation, vous pouvez vérifier que vous ne commettez pas d'erreurs d'espace qui pourraient ennuyer les autres développeurs.

Ensuite, assurez-vous de faire de chaque validation une modification logiquement atomique. Si possible, rendez chaque modification digeste — ne codez pas pendant un week-end entier sur cinq sujets différents pour enfin les soumettre tous dans une énorme validation le lundi suivant. Même si vous ne validez pas du week-end, utilisez la zone d'index le lundi pour

découper votre travail en au moins une validation par problème, avec un message utile par validation. Si certaines modifications touchent au même fichier, essayez d'utiliser `git add --patch` pour indexer partiellement des fichiers (cette fonctionnalité est traitée au chapitre [Indexation interactive](#)). L'instantané final sera identique, que vous utilisiez une validation unique ou cinq petites validations, à condition que toutes les modifications soient intégrées à un moment, donc n'hésitez pas à rendre la vie plus simple à vos compagnons développeurs lorsqu'ils auront à vérifier vos modifications. Cette approche simplifie aussi le retrait ou l'inversion ultérieurs d'une modification en cas de besoin. Le chapitre [Réécrire l'historique](#) décrit justement quelques trucs et astuces de Git pour réécrire l'historique et indexer interactivement les fichiers — utilisez ces outils pour fabriquer un historique propre et compréhensible.

Le dernier point à soigner est le message de validation. S'habituer à écrire des messages de validation de qualité facilite grandement l'emploi et la collaboration avec Git. En règle générale, les messages doivent débuter par une ligne unique d'au plus 50 caractères décrivant concisément la modification, suivie d'une ligne vide, suivie d'une explication plus détaillée. Le projet Git exige que l'explication détaillée inclue la motivation de la modification en contrastant le nouveau comportement par rapport à l'ancien — c'est une bonne règle de rédaction. Une bonne règle consiste aussi à utiliser le présent de l'impératif ou des verbes substantivés dans le message. En d'autres termes, utilisez des ordres. Au lieu d'écrire « J'ai ajouté des tests pour » ou « En train d'ajouter des tests pour », utilisez juste « Ajoute des tests pour » ou « Ajout de tests pour ».

Voici ci-dessous un modèle écrit par Tim Pope :

```
Court résumé des modifications (50 caractères ou moins)

Explication plus détaillée, si nécessaire. Retour à la ligne vers 72
caractères. Dans certains contextes, la première ligne est traitée
comme le sujet d'un courriel et le reste comme le corps. La ligne
vide qui sépare le titre du corps est importante (à moins d'omettre
totalement le corps). Des outils tels que rebase peuvent être gênés
si vous les laissez collés.

Paragraphes supplémentaires après des lignes vides.

- Les listes à puce sont aussi acceptées

- Typiquement, un tiret ou un astérisque précédés d'un espace unique
  séparés par des lignes vides mais les conventions peuvent varier
```

Si tous vos messages de validation ressemblent à ceci, les choses seront beaucoup plus simples pour vous et les développeurs avec qui vous travaillez. Le projet Git montre des messages de *commit* bien formatés — lancez donc `git log --no-merges` dessus pour voir à quoi ressemble un historique de *commits* avec des messages bien formatés.

Dans les exemples suivants et à travers tout ce livre, par souci de simplification, je ne formaterai pas les messages aussi proprement. J'utiliserai plutôt l'option `-m` de `git commit`. Faites ce que je dis, pas ce que je fais.

Cas d'une petite équipe privée

Le cas le plus probable que vous rencontrerez est celui du projet privé avec un ou deux autres développeurs. Par privé, j'entends code source fermé non accessible au public en lecture. Vous et les autres développeurs aurez accès en poussée au dépôt.

Dans cet environnement, vous pouvez suivre une méthode similaire à ce que vous feriez en utilisant Subversion ou tout autre système centralisé. Vous bénéficiez toujours d'avantages tels que la validation hors-ligne et la gestion de branche et de fusion grandement simplifiée mais les étapes restent similaires. La différence principale reste que les fusions ont lieu du côté client plutôt que sur le serveur au moment de valider. Voyons à quoi pourrait ressembler la collaboration de deux développeurs sur un dépôt partagé. Le premier développeur, John, clone le dépôt, fait une modification et valide localement. Dans les exemples qui suivent, les messages de protocole sont remplacés par `...` pour les raccourcir.

```
# Ordinateur de John
$ git clone john@githost:simplegit.git
Clonage dans 'simplegit'...
...
$ cd simplegit/
```

```
$ vim lib/simplegit.rb
$ git commit -am 'Eliminer une valeur par défaut invalide'
[master 738ee87] Eliminer une valeur par défaut invalide
1 files changed, 1 insertions(+), 1 deletions(-)
```

La deuxième développeuse, Jessica, fait la même chose. Elle clone le dépôt et valide une modification :

```
# Ordinateur de Jessica
$ git clone jessica@github:simplegit.git
Clonage dans 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Ajouter une tâche reset'
[master fbff5bc] Ajouter une tâche reset
1 files changed, 1 insertions(+), 0 deletions(-)
```

À présent, Jessica pousse son travail sur le serveur :

```
# Ordinateur de Jessica
$ git push origin master
...
To jessica@github:simplegit.git
1edee6b..fbff5bc master -> master
```

John tente aussi de pousser ses modifications :

```
# Ordinateur de John
$ git push origin master
To john@github:simplegit.git
! [rejected]        master -> master (non-fast forward)
error: impossible de pousser des références vers 'john@github:simplegit.git'
astuce: Les mises à jour ont été rejetées car la pointe de la branche courante est derrière
astuce: son homologue distant. Intégrez les changements distants (par exemple 'git pull ...')
astuce: avant de pousser à nouveau.
astuce: Voir la 'Note à propos des avances rapides' dans 'git push --help' pour plus d'information.
```

John n'a pas le droit de pousser parce que Jessica a déjà poussé dans l'intervalle. Il est très important de comprendre ceci si vous avez déjà utilisé Subversion, parce qu'il faut remarquer que les deux développeurs n'ont pas modifié le même fichier. Quand des fichiers différents ont été modifiés, Subversion réalise cette fusion automatiquement sur le serveur alors que Git nécessite une fusion des modifications locale. John doit récupérer les modifications de Jessica et les fusionner avant d'être autorisé à pousser :

```
$ git fetch origin
...
From john@github:simplegit
+ 049d078...fbff5bc master -> origin/master
```

À présent, le dépôt local de John ressemble à la figure 5-4.

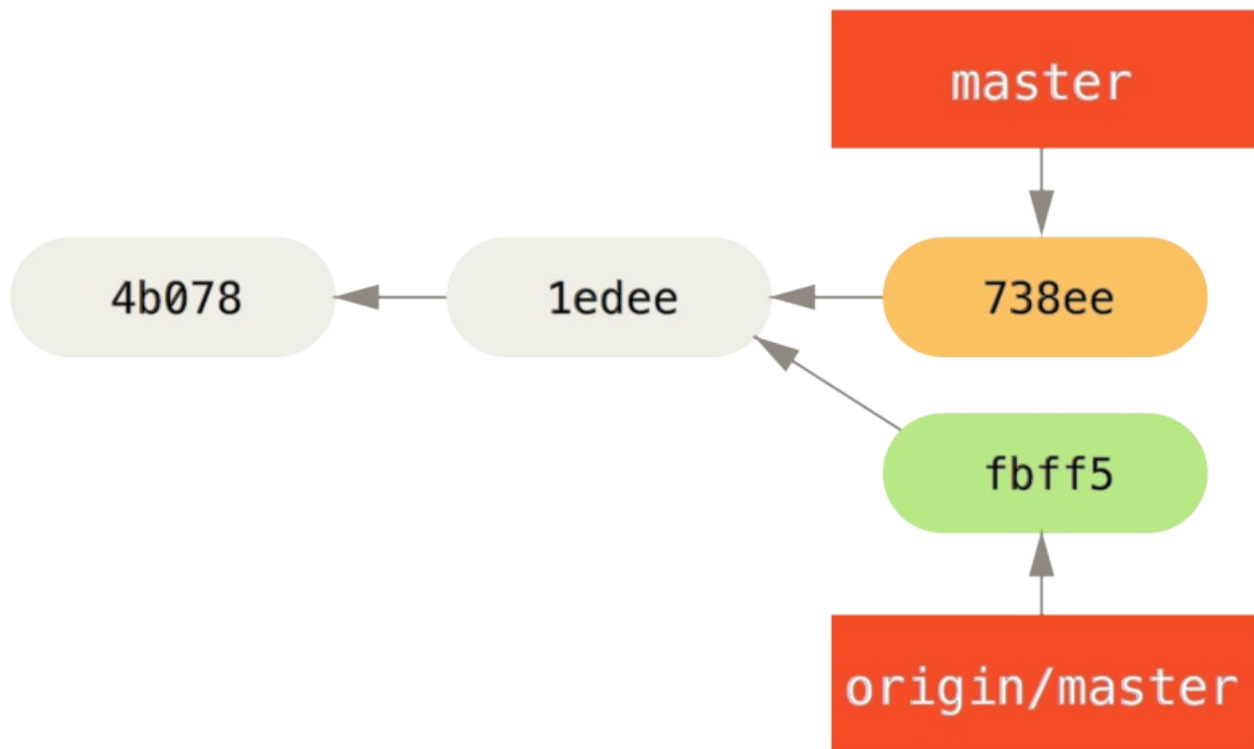


Figure 59 : Historique divergent de John.

Figure 58. Historique divergent de John.

John a une référence aux modifications que Jessica a poussées, mais il doit les fusionner dans sa propre branche avant d'être autorisé à pousser :

```

$ git merge origin/master
Merge made by recursive.
TODO | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
  
```

Cette fusion se passe sans problème — l'historique de *commits* de John ressemble à présent à ceci :

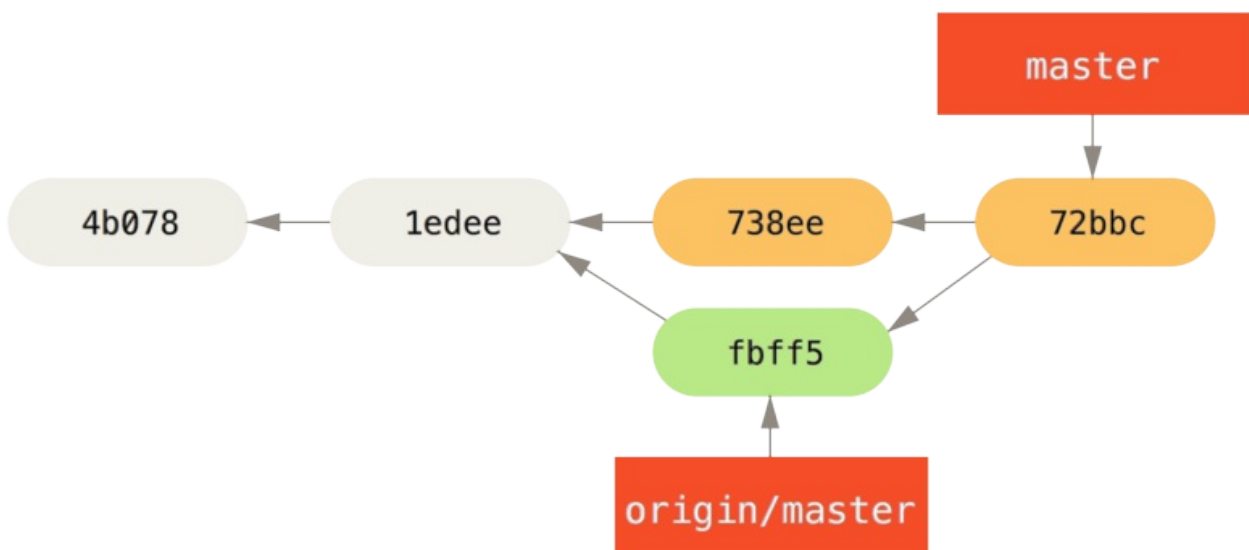


Figure 60 : Le dépôt de John après la fusion d'`origin/master`.

Figure 59. Le dépôt de John après la fusion d' `origin/master` .

Maintenant, John peut tester son code pour s'assurer qu'il fonctionne encore correctement et peut pousser son travail nouvellement fusionné sur le serveur :

```
$ git push origin master
...
To john@github:simplegit.git
fbff5bc..72bbc59  master -> master
```

À la fin, l'historique des *commits* de John ressemble à ceci :

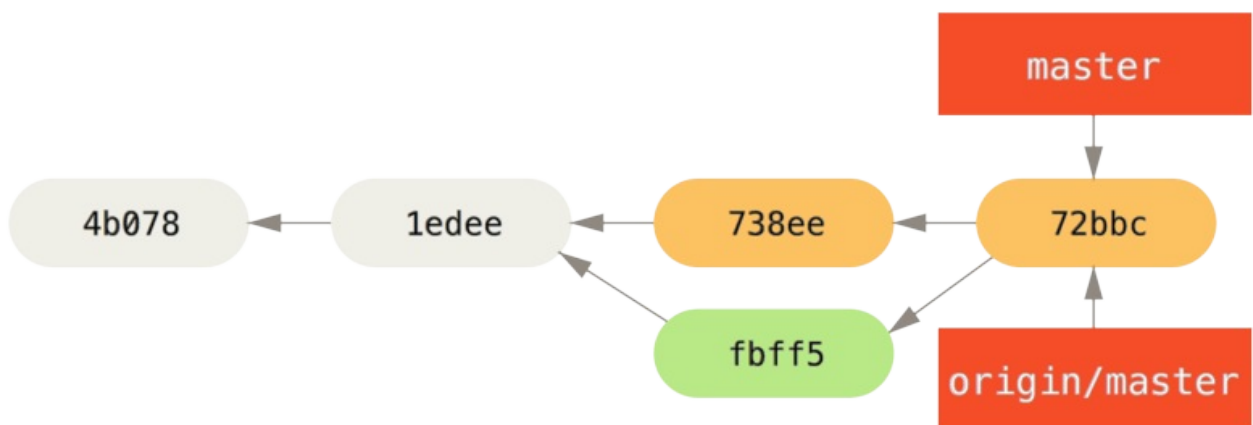


Figure 61 : L'historique de John après avoir poussé sur le serveur origin.

Figure 60. L'historique de John après avoir poussé sur le serveur origin.

Dans l'intervalle, Jessica a travaillé sur une branche thématique. Elle a créé une branche thématique nommée `prob54` et réalisé trois validations sur cette branche. Elle n'a pas encore récupéré les modifications de John, ce qui donne un historique semblable à ceci :

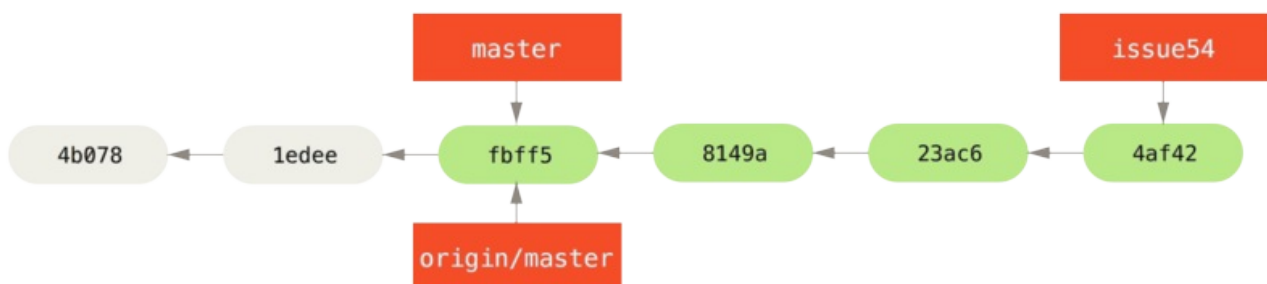


Figure 62 : La branche thématique de Jessica.

Figure 61. La branche thématique de Jessica.

Jessica souhaite se synchroniser sur le travail de John. Elle récupère donc ses modifications :

```
# Ordinateur de Jessica
$ git fetch origin
```

```
...
From jessica@github:simplegit
fbff5bc..72bbc59 master -> origin/master
```

Cette commande tire le travail que John avait poussé dans l'intervalle. L'historique de Jessica ressemble maintenant à ceci :

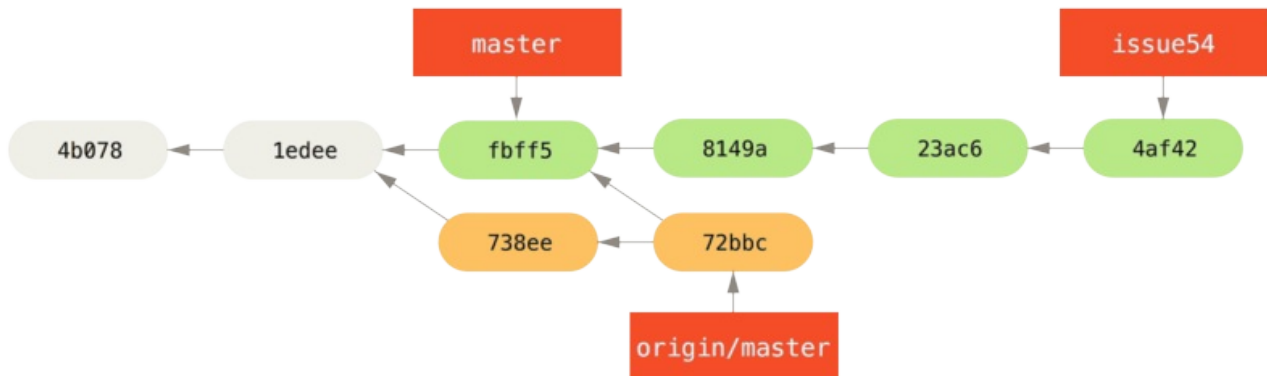


Figure 63 : L'historique de Jessica après avoir récupéré les modifications de John.

Figure 62. L'historique de Jessica après avoir récupéré les modifications de John.

Jessica pense que sa branche thématique est prête mais elle souhaite savoir si elle doit fusionner son travail avant de pouvoir pousser. Elle lance `git log` pour s'en assurer :

```
$ git log --no-merges issue54..origin/master
commit 738ee872852d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date: Fri May 29 16:01:27 2009 -0700
```

Eliminer une valeur par défaut invalide

La syntaxe `prob54..origin/master` est un filtre du journal qui ordonne à Git de ne montrer que la liste des *commits* qui sont sur la seconde branche (dans ce cas `origin/master`) et qui ne sont pas sur la première (dans ce cas `prob54`). Nous aborderons cette syntaxe en détail dans [Plages de commits](#).

Pour l'instant, nous pouvons voir dans le résultat qu'il n'y a qu'un seul *commit* créé par John que Jessica n'a pas fusionné. Si elle fusionne `origin/master`, ce sera le seul commit qui modifiera son travail local.

Maintenant, Jessica peut fusionner sa branche thématique dans sa branche `master`, fusionner le travail de John (`origin/master`) dans sa branche `master`, puis pousser le résultat sur le serveur. Premièrement, elle rebasecule sur sa branche `master` pour intégrer son travail :

```
$ git checkout master
Basculement sur la branche 'master'
Votre branche est en retard sur 'origin/master' de 2 commits, et peut être mise à jour en avance rapide.
```

Elle peut fusionner soit `origin/master` soit `prob54` en premier — les deux sont en avance, mais l'ordre n'importe pas. L'instantané final devrait être identique quel que soit l'ordre de fusion qu'elle choisit. Seul l'historique sera légèrement différent. Elle choisit de fusionner en premier `prob54` :

```
$ git merge issue54
Mise à jour fbff5bc..4af4298
Avance rapide
LISEZMOI      | 1 +
lib/simplegit.rb | 6 +++++
2 files changed, 6 insertions(+), 1 deletions(-)
```

Aucun problème n'apparaît. Comme vous pouvez le voir, c'est une simple avance rapide. Maintenant, Jessica fusionne le travail de John (`origin/master`) :

```
$ git merge origin/master
Fusion automatique de lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb | 2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Tout a fusionné proprement et l'historique de Jessica ressemble à ceci :

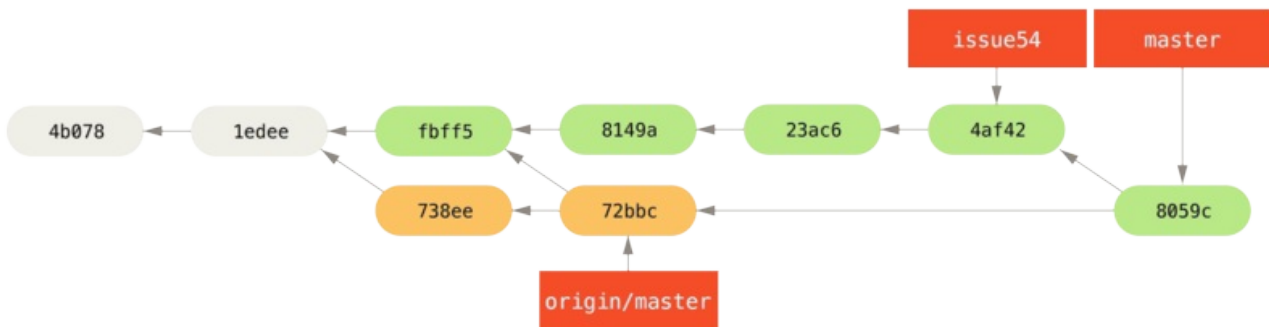


Figure 64 : L'historique de Jessica après avoir fusionné les modifications de John.

Figure 63. L'historique de Jessica après avoir fusionné les modifications de John.

Maintenant `origin/master` est accessible depuis la branche `master` de Jessica, donc elle devrait être capable de pousser (en considérant que John n'a pas encore poussé dans l'intervalle) :

```
$ git push origin master
...
To jessica@github:simplegit.git
 72bbc59..8059c15 master -> master
```

Chaque développeur a validé quelques fois et fusionné les travaux de l'autre avec succès.

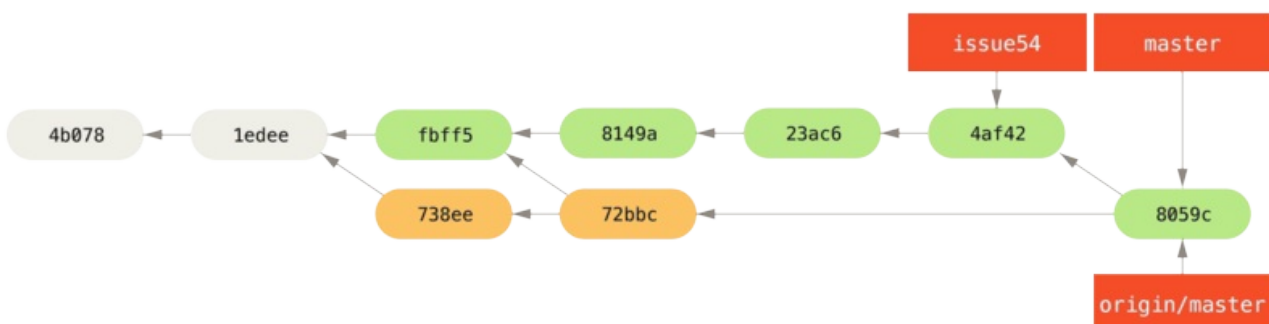


Figure 65 : L'historique de Jessica après avoir poussé toutes ses modifications sur le serveur.

Figure 64. L'historique de Jessica après avoir poussé toutes ses modifications sur le serveur.

C'est un des schémas les plus simples. Vous travaillez pendant quelque temps, généralement sur une branche thématique, et fusionnez dans votre branche `master` quand elle est prête à être intégrée. Quand vous souhaitez partager votre travail, vous récupérez `origin/master` et la fusionnez si elle a changé, puis finalement vous poussez le résultat sur la branche `master` du serveur. La séquence correspond à ceci :

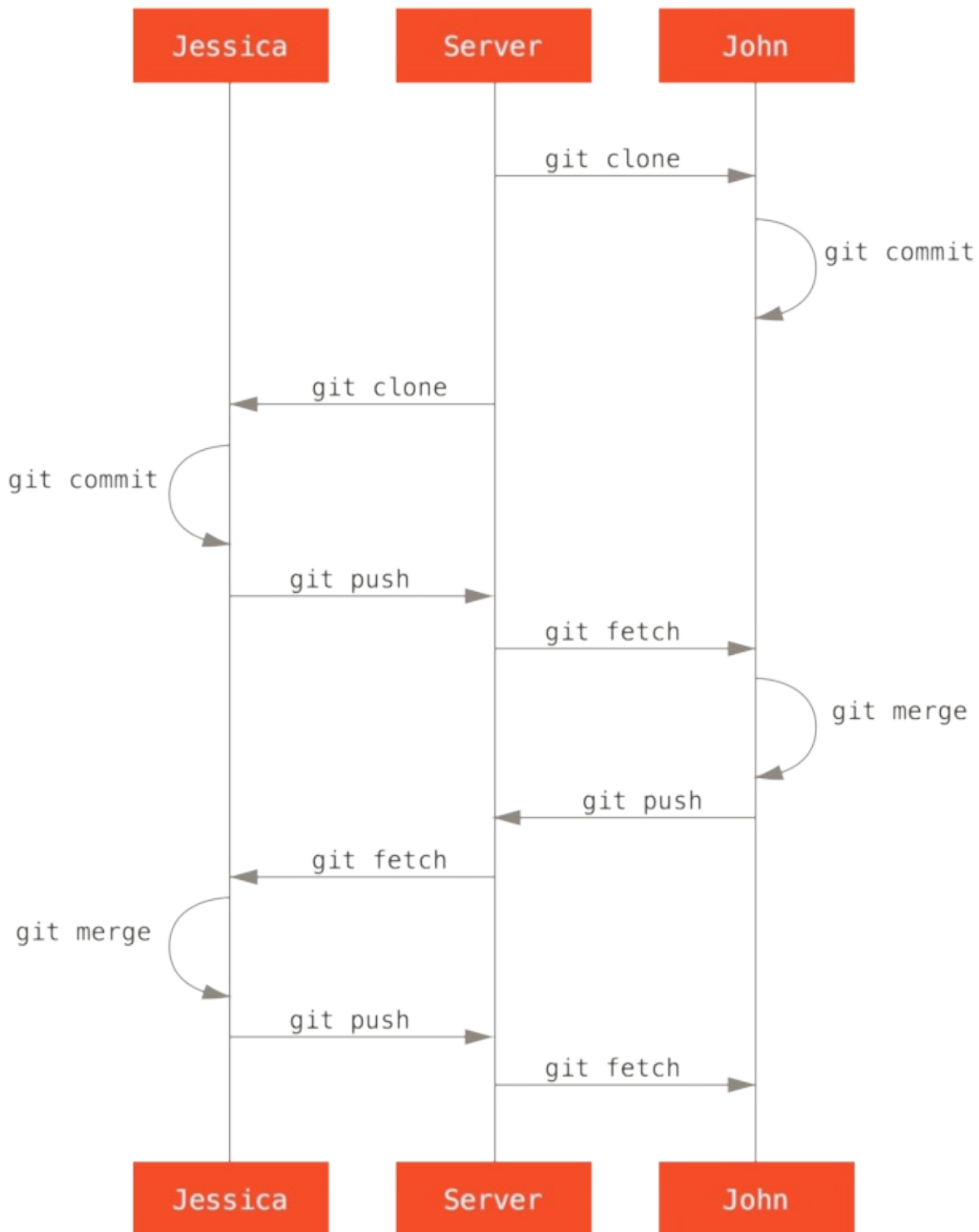


Figure 66 : Séquence générale des événements pour une utilisation simple multi-développeur de Git.

Figure 65. Séquence générale des événements pour une utilisation simple multi-développeur de Git.

Équipe privée importante

Dans le scénario suivant, nous aborderons les rôles de contributeur dans un groupe privé plus grand. Vous apprendrez comment travailler dans un environnement où des petits groupes collaborent sur des fonctionnalités, puis les contributions de chaque équipe sont intégrées par une autre entité.

Supposons que John et Jessica travaillent ensemble sur une première fonctionnalité, tandis que Jessica et Josie travaillent sur une autre. Dans ce cas, l'entreprise utilise un mode d'opération de type « gestionnaire d'intégration » où le travail des groupes est intégré par certains ingénieurs, et la branche `master` du dépôt principal ne peut être mise à jour que par ces ingénieurs. Dans ce scénario, tout le travail est validé dans des branches orientées équipe, et tiré plus tard par les intégrateurs.

Suivons le cheminement de Jessica tandis qu'elle travaille sur les deux nouvelles fonctionnalités, collaborant en parallèle avec deux développeurs différents dans cet environnement. En supposant qu'elle ait cloné son dépôt, elle décide de travailler sur la `fonctionA` en premier. Elle crée une nouvelle branche pour cette fonction et travaille un peu dessus :

```
# Ordinateur de Jessica
$ git checkout -b fonctionA
Basculement sur la nouvelle branche 'fonctionA'
$ vim lib/simplegit.rb
$ git commit -am 'Ajouter une limite à la fonction de log'
[fonctionA 3300904] Ajouter une limite à la fonction de log
1 files changed, 1 insertions(+), 1 deletions(-)
```

À ce moment, elle a besoin de partager son travail avec John, donc elle pousse les *commits* de sa branche `fonctionA` sur le serveur. Jessica n'a pas le droit de pousser sur la branche `master` — seuls les intégrateurs l'ont — et elle doit donc pousser sur une autre branche pour collaborer avec John :

```
$ git push -u origin fonctionA
...
To jessica@github:simplegit.git
* [nouvelle branche]    fonctionA -> fonctionA
```

Jessica envoie un courriel à John pour lui indiquer qu'elle a poussé son travail dans la branche appelée `fonctionA` et qu'il peut l'inspecter. Pendant qu'elle attend le retour de John, Jessica décide de commencer à travailler sur la `fonctionB` avec Josie. Pour commencer, elle crée une nouvelle branche thématique, à partir de la base `master` du serveur :

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b fonctionB origin/master
Basculement sur la nouvelle branche 'fonctionB'
```

À présent, Jessica réalise quelques validations sur la branche `fonctionB` :

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
1 files changed, 5 insertions(+), 0 deletions(-)
```

Le dépôt de Jessica ressemble à la figure suivante :

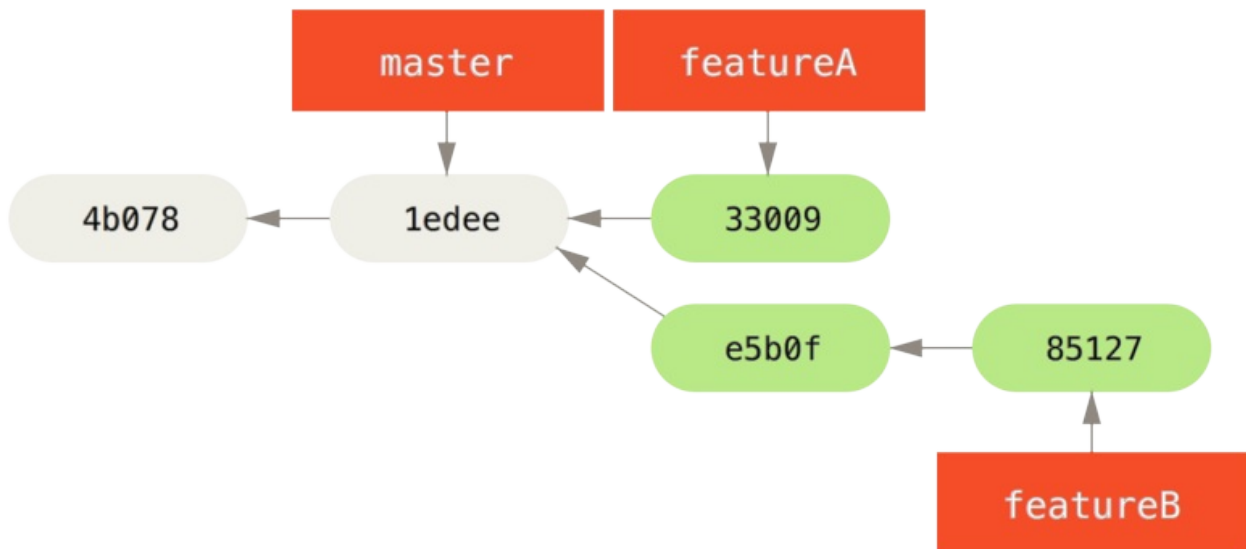


Figure 67 : L'historique initial de Jessica.

Figure 66. L'historique initial de Jessica.

Elle est prête à pousser son travail, mais elle reçoit un mail de Josie indiquant qu'une branche avec un premier travail a déjà été poussé sur le serveur en tant que `fonctionBee`. Jessica doit d'abord fusionner ces modifications avec les siennes avant de pouvoir pousser sur le serveur. Elle peut récupérer les modifications de Josie avec `git fetch` :

```
$ git fetch origin
...
From jessica@github:simplegit
* [nouvelle branche]    fonctionBee -> origin/fonctionBee
```

Jessica peut à présent fusionner ceci dans le travail qu'elle a réalisé grâce à `git merge` :

```
$ git merge origin/fonctionBee
Fusion automatique de lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb | 4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Mais il y a un petit problème — elle doit pousser son travail fusionné dans sa branche `fonctionB` sur la branche `fonctionBee` du serveur. Elle peut le faire en spécifiant la branche locale suivie de deux points (:) suivi de la branche distante à la commande `git push` :

```
$ git push -u origin fonctionB:fonctionBee
...
To jessica@github:simplegit.git
 fba9af8..cd685d1 fonctionB -> fonctionBee
```

Cela s'appelle une *refspec*. Référez-vous à [La refspec](#) pour une explication plus détaillée des refsspecs Git et des possibilités qu'elles offrent. Notez l'option `-u`. C'est un raccourci pour `--set-upstream`, qui configure les branches pour faciliter les poussées et les tirages plus tard.

Ensuite, John envoie un courriel à Jessica pour lui indiquer qu'il a poussé des modifications sur la branche `fonctionA` et lui demander de les vérifier. Elle lance `git fetch` pour tirer toutes ces modifications :

```
$ git fetch origin
...
```

```
From jessica@github:simplegit
3300904..aad881d fonctionA -> origin/fonctionA
```

Elle peut voir ce qui a été modifié avec `git log` :

```
$ git log fonctionA..origin/fonctionA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date: Fri May 29 19:57:33 2009 -0700
```

largeur du log passée de 25 à 30

Finalement, elle fusionne le travail de John dans sa propre branche `fonctionA` :

```
$ git checkout fonctionA
Basculement sur la branche 'fonctionA'
$ git merge origin/fonctionA
Updating 3300904..aad881d
Avance rapide
lib/simplegit.rb | 10 +++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica veut régler quelques détails. Elle valide donc encore et pousse ses changements sur le serveur :

```
$ git commit -am 'details regles'
[fonctionA ed774b3] details regles
1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@github:simplegit.git
3300904..ed774b3 fonctionA -> fonctionA
```

L'historique des *commits* de Jessica ressemble à présent à ceci :

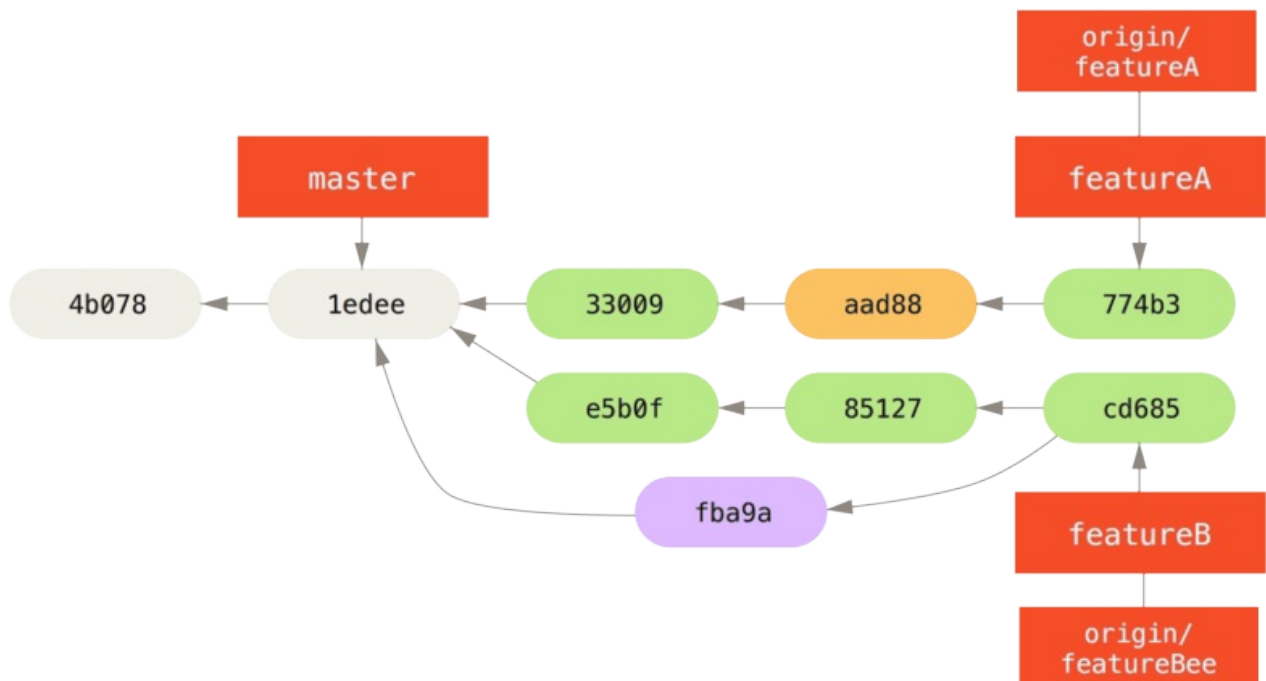


Figure 68 : L'historique de Jessica après la validation dans la branche thématique.

Figure 67. L'historique de Jessica après la validation dans la branche thématique.

Jessica, Josie et John informent les intégrateurs que les branches `fonctionA` et `fonctionB` du serveur sont prêtes pour une intégration dans la branche principale. Après cette intégration dans la branche principale, une synchronisation apportera les *commits* de fusion, ce qui donnera un historique comme celui-ci :

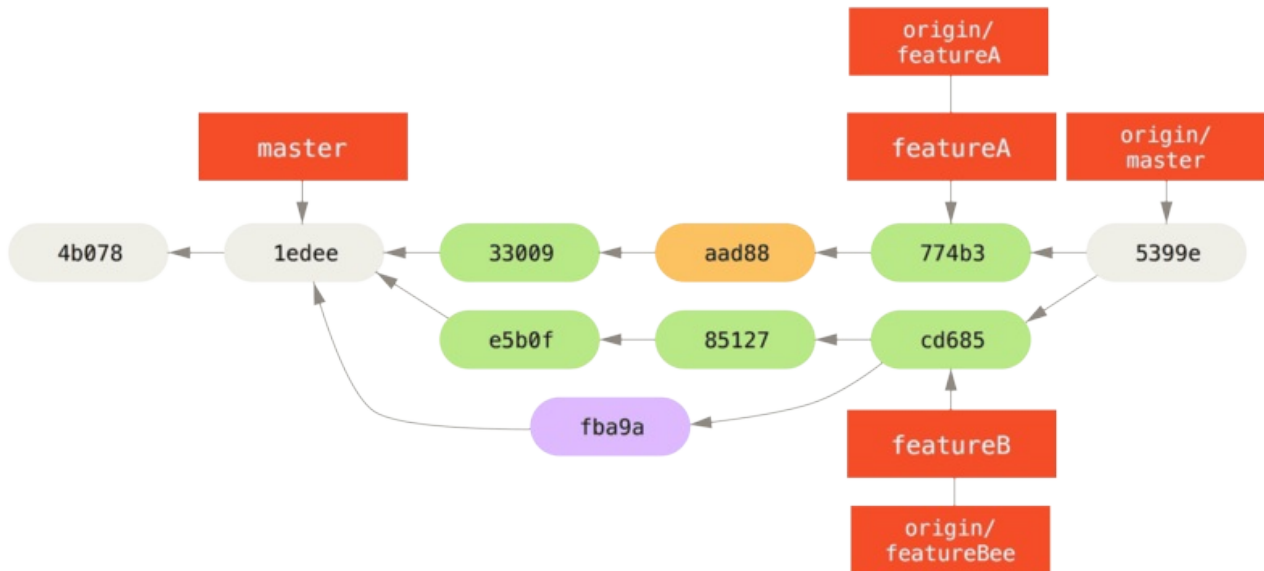


Figure 69 : L'historique de Jessica après la fusion de ses deux branches thématiques.

Figure 68. L'historique de Jessica après la fusion de ses deux branches thématiques.

De nombreux groupes basculent vers Git du fait de cette capacité à gérer plusieurs équipes travaillant en parallèle, fusionnant plusieurs lignes de développement très tard dans le processus de livraison. La capacité donnée à plusieurs sous-groupes d'équipes de collaborer au moyen de branches distantes sans nécessairement impacter le reste de l'équipe est un grand bénéfice apporté par Git. La séquence de travail qui vous a été décrite ressemble à la figure suivante :

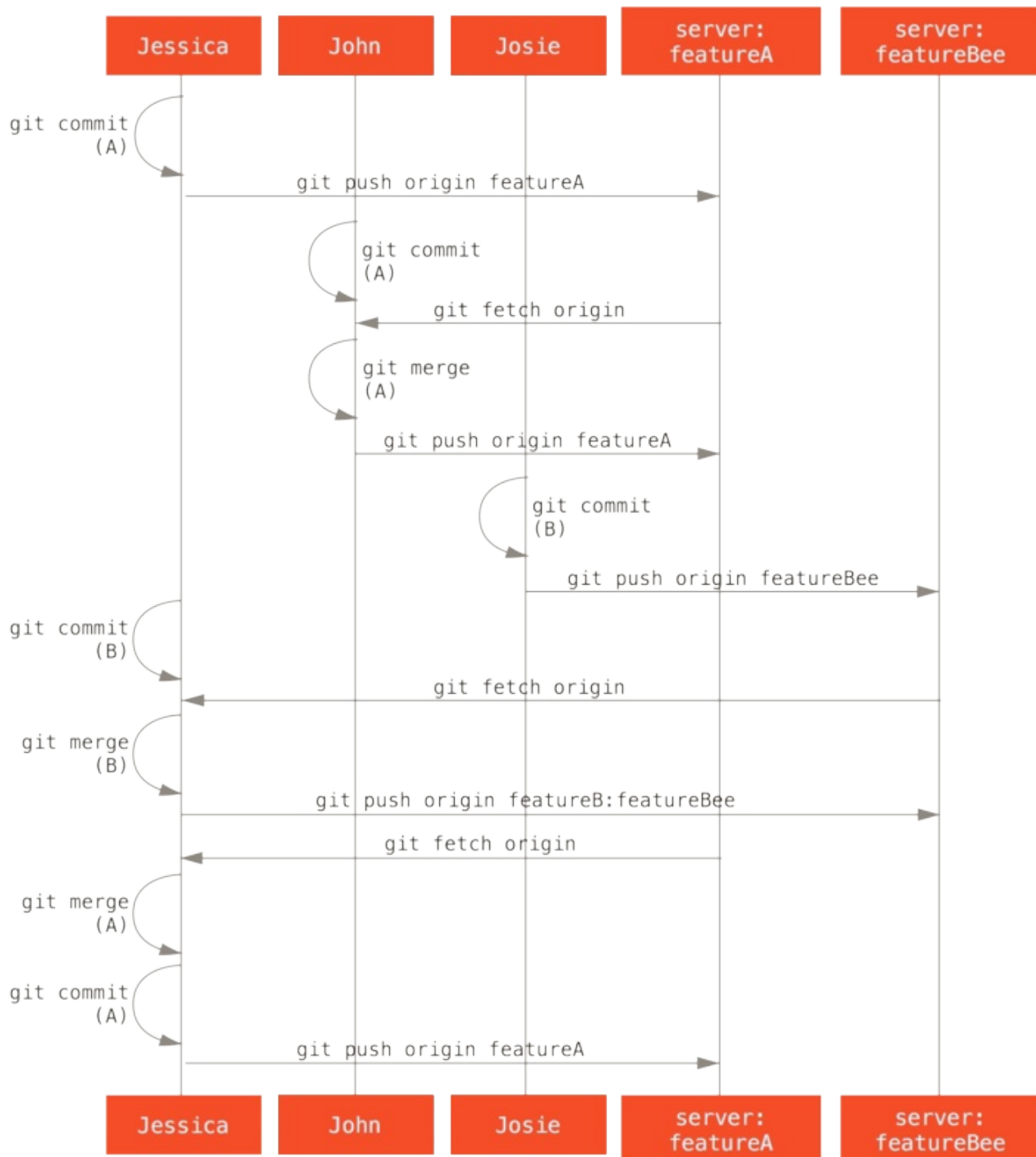


Figure 70 : Une séquence simple de gestion orientée équipe.

Figure 69. Une séquence simple de gestion orientée équipe.

Projet public dupliqué

Contribuer à un projet public est assez différent. Il faut présenter le travail au mainteneur d'une autre manière parce que vous n'avez pas la possibilité de mettre à jour directement des branches du projet. Ce premier exemple décrit un mode de contribution via des serveurs Git qui proposent facilement la duplication de dépôt. De nombreux sites proposent cette méthode (dont GitHub, BitBucket, Google Code, repo.or.cz), et de nombreux mainteneurs s'attendent à ce style de contribution. Le chapitre suivant traite des projets qui préfèrent accepter les contributions sous forme de patch via courriel.

Premièrement, vous souhaitez probablement cloner le dépôt principal, créer une nouvelle branche thématique pour le patch ou la série de patches que seront votre contribution, et commencer à travailler. La séquence ressemble globalement à ceci :

```
$ git clone (url)
$ cd projet
$ git checkout -b fonctionA
# (travail)
$ git commit
# (travail)
$ git commit
```

Vous pouvez utiliser `rebase -i` pour réduire votre travail à une seule validation ou pour réarranger les modifications dans des *commits* qui rendront les patches plus faciles à relire pour le mainteneur — référez-vous à [Réécrire l'historique](#) pour plus d'information sur comment rebaser de manière interactive.

Lorsque votre branche de travail est prête et que vous êtes prêt à la fournir au mainteneur, rendez-vous sur la page du projet et cliquez sur le bouton « Fork » pour créer votre propre projet dupliqué sur lequel vous aurez les droits en écriture. Vous devez alors ajouter l'URL de ce nouveau dépôt en tant que second dépôt distant, dans notre cas nommé `macopie` :

```
$ git remote add macopie (url)
```

Vous devez pousser votre travail sur ce dépôt distant. C'est beaucoup plus facile de pousser la branche sur laquelle vous travaillez sur une branche distante que de fusionner et de pousser le résultat sur le serveur. La raison principale en est que si le travail n'est pas accepté ou s'il est picoré, vous n'aurez pas à faire marche arrière sur votre branche `master`. Si le mainteneur fusionne, rebase ou picore votre travail, vous le saurez en tirant depuis son dépôt :

```
$ git push -u macopie fonctionA
```

Une fois votre travail poussé sur votre dépôt copie, vous devez notifier le mainteneur. Ce processus est souvent appelé une demande de tirage (*pull request*) et vous pouvez la générer soit via le site web — GitHub propose son propre mécanisme qui sera traité au chapitre [GitHub](#) — soit lancer la commande `git request-pull` et envoyer manuellement par courriel le résultat au mainteneur de projet.

La commande `request-pull` prend en paramètres la branche de base dans laquelle vous souhaitez que votre branche thématique soit fusionnée et l'URL du dépôt Git depuis lequel vous souhaitez qu'elle soit tirée, et génère un résumé des modifications que vous demandez à faire tirer. Par exemple, si Jessica envoie à John une demande de tirage et qu'elle a fait deux validations dans la branche thématique qu'elle vient de pousser, elle peut lancer ceci :

```
$ git request-pull origin/master macopie
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    ajout d'une nouvelle fonction

are available in the git repository at:

  git://github.com/simblegit.git fonctionA

Jessica Smith (2):
  Ajout d'une limite à la fonction de log
  change la largeur du log de 25 à 30

lib/simblegit.rb | 10 ++++++
1 files changed, 9 insertions(+), 1 deletions(-)
```

Le résultat peut être envoyé au mainteneur – cela lui indique d'où la modification a été branchée, le résumé des validations et d'où tirer ce travail.

Pour un projet dont vous n'êtes pas le mainteneur, il est généralement plus aisé de toujours laisser la branche `master` suivre `origin/master` et de réaliser vos travaux sur des branches thématiques que vous pourrez facilement effacer si elles sont rejetées. Garder les thèmes de travaux isolés sur des branches thématiques facilite aussi leur rebasage si le sommet du dépôt principal a avancé dans l'intervalle et que vos modifications ne s'appliquent plus proprement. Par exemple, si vous souhaitez soumettre un second sujet de travail au projet, ne continuez pas à travailler sur la branche thématique que vous venez de pousser mais démarrez-en plutôt une depuis la branche `master` du dépôt principal :

```
$ git checkout -b fonctionB origin/master
# (travail)
$ git commit
$ git push macopie fonctionB
# (email au maintainer)
$ git fetch origin
```

À présent, chaque sujet est contenu dans son propre silo – similaire à une file de patches – que vous pouvez réécrire, rebaser et modifier sans que les sujets n'interfèrent ou ne dépendent les uns des autres, comme ceci :

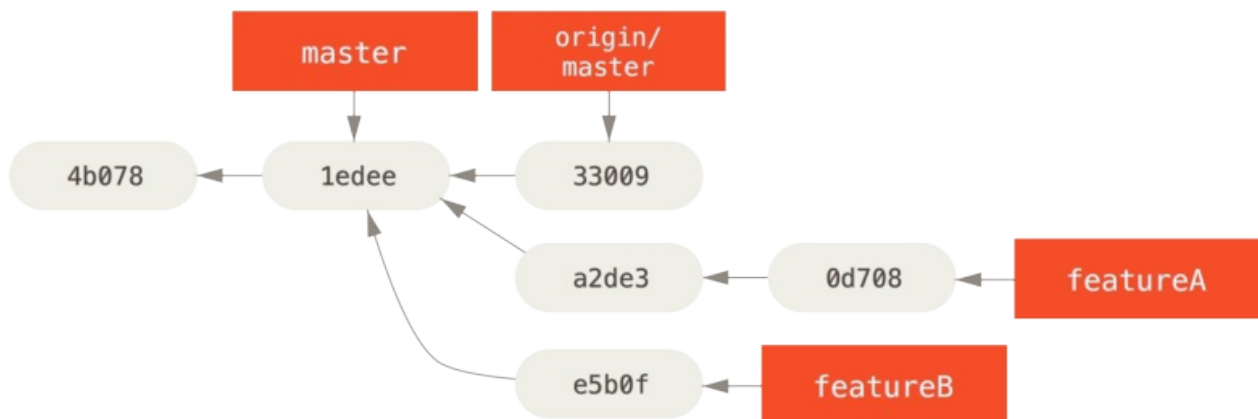


Figure 71 : Historique initial des `\commits\` avec les modifications de fonctionB.

Figure 70. Historique initial des `commits` avec les modifications de fonctionB.

Supposons que le mainteneur du projet a tiré une poignée d'autres patches et essayé par la suite votre première branche, mais celle-ci ne s'applique plus proprement. Dans ce cas, vous pouvez rebaser cette branche au sommet de `origin/master`, résoudre les conflits pour le mainteneur et soumettre de nouveau vos modifications :

```
$ git checkout fonctionA
$ git rebase origin/master
$ git push -f macopie fonctionA
```

Cette action réécrit votre historique pour qu'il ressemble à [Historique des validations après le travail sur fonctionA..](#)

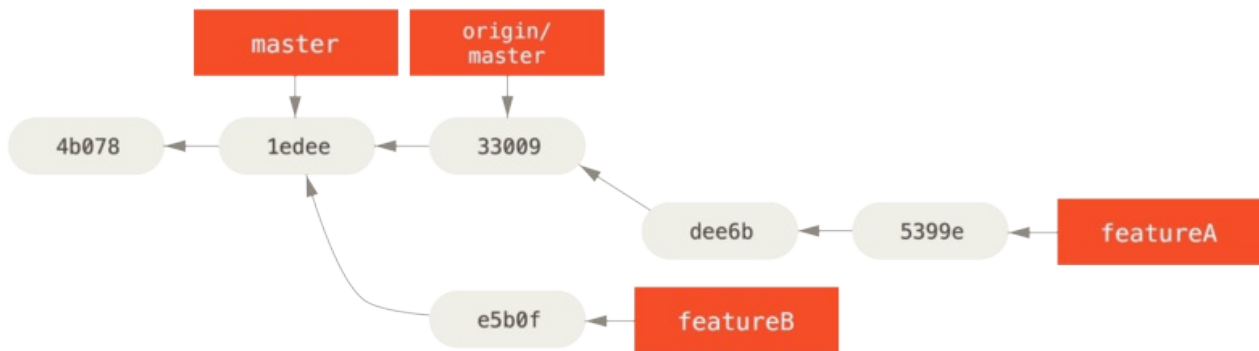


Figure 72 : Historique des validations après le travail sur fonctionA.

Figure 71. Historique des validations après le travail sur fonctionA.

Comme vous avez rebasé votre branche, vous devez spécifier l'option `-f` à votre commande pour pousser, pour forcer le remplacement de la branche `fonctionA` sur le serveur par la suite de `commits` qui n'en est pas descendante. Une solution alternative serait de pousser ce nouveau travail dans une branche différente du serveur (appelée par exemple `fonctionAv2`).

Examinons un autre scénario possible : le mainteneur a revu les modifications dans votre seconde branche et apprécie le concept, mais il souhaiterait que vous changiez des détails d'implémentation. Vous en profiterez pour rebaser ce travail sur le sommet actuel de la branche `master` du projet. Vous démarrez une nouvelle branche à partir de la branche `origin/master` courante, y collez les modifications de `fonctionB` en résolvant les conflits, changez l'implémentation et poussez le tout en tant que nouvelle branche :

```

$ git checkout -b fonctionBv2 origin/master
$ git merge --no-commit --squash fonctionB
# (changement d'implémentation)
$ git commit
$ git push macopie fonctionBv2

```

L'option `--squash` prend tout le travail de la branche à fusionner et le colle dans un `commit` sans fusion au sommet de la branche extraite. L'option `--no-commit` indique à Git de ne pas enregistrer automatiquement une validation. Cela permet de reporter toutes les modifications d'une autre branche, puis de réaliser d'autres modifications avant de réaliser une nouvelle validation.

À présent, vous pouvez envoyer au mainteneur un message indiquant que vous avez réalisé les modifications demandées et qu'il peut trouver cette nouvelle mouture sur votre branche `fonctionBv2`.

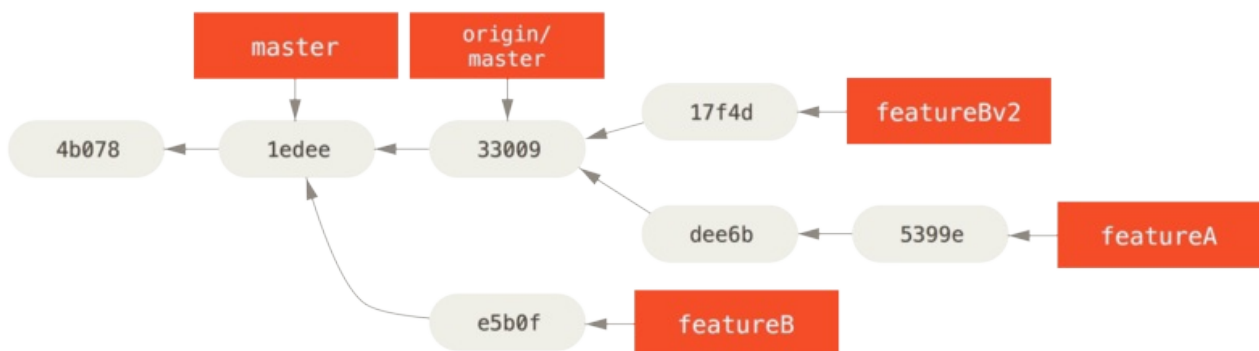


Figure 73 : Historique des validations après le travail sur fonctionBv2.

Figure 72. Historique des validations après le travail sur fonctionBv2.

Projet public via courriel

De nombreux grands projets ont des procédures établies pour accepter des patches — il faut vérifier les règles spécifiques à chaque projet qui peuvent varier. Comme il existe quelques gros projets établis qui acceptent les patches via une liste de diffusion de développement, nous allons éclairer cette méthode d'un exemple.

La méthode est similaire au cas précédent — vous créez une branche thématique par série de patches sur laquelle vous travaillez. La différence réside dans la manière de les soumettre au projet. Au lieu de dupliquer le projet et de pousser vos soumissions sur votre dépôt, il faut générer des versions courriel de chaque série de *commits* et les envoyer à la liste de diffusion de développement.

```
$ git checkout -b topicA
# (travail)
$ git commit
# (travail)
$ git commit
```

Vous avez à présent deux *commits* que vous souhaitez envoyer à la liste de diffusion. Vous utilisez `git format-patch` pour générer des fichiers au format mbox que vous pourrez envoyer à la liste. Cette commande transforme chaque *commit* en un message courriel dont le sujet est la première ligne du message de validation et le corps contient le reste du message plus le patch correspondant. Un point intéressant de cette commande est qu'appliquer le patch à partir d'un courriel formaté avec `format-patch` préserve toute l'information de validation.

```
$ git format-patch -M origin/master
0001-Ajout-d-une-limite-la-fonction-de-log.patch
0002-change-la-largeur-du-log-de-25-a-30.patch
```

La commande `format-patch` affiche les noms de fichiers de patch créés. L'option `-M` indique à Git de suivre les renommages. Le contenu des fichiers ressemble à ceci :

```
$ cat 0001-Ajout-d-une-limite-la-fonction-de-log.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] Ajout d'un limite à la fonction de log

Limite la fonctionnalité de log aux 20 premières lignes

---
 lib/simplegit.rb | 2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```


Vous pouvez maintenant éditer ces fichiers de patch pour ajouter plus d'informations à destination de la liste de diffusion mais que vous ne souhaitez pas voir apparaître dans le message de validation. Si vous ajoutez du texte entre la ligne `---` et le début du patch (la ligne `diff --git`), les développeurs peuvent le lire mais l'application du patch ne le prend pas en compte.

Pour envoyer par courriel ces fichiers, vous pouvez soit copier leur contenu dans votre application de courriel, soit l'envoyer via une ligne de commande. Le copier-coller cause souvent des problèmes de formatage, spécialement avec les applications « intelligentes » qui ne préservent pas les retours à la ligne et les types d'espace. Heureusement, Git fournit un outil pour envoyer correctement les patchs formatés via IMAP, la méthode la plus facile. Je démontrerai comment envoyer un patch via Gmail qui s'avère être la boîte mail que j'utilise ; vous pourrez trouver des instructions détaillées pour de nombreuses applications de mail à la fin du fichier susmentionné [Documentation/SubsubmittingPatches](#) du code source de Git.

Premièrement, il est nécessaire de paramétrer la section `imap` de votre fichier `~/.gitconfig`. Vous pouvez positionner ces valeurs séparément avec une série de commandes `git config`, ou vous pouvez les ajouter manuellement. À la fin, le fichier de configuration doit ressembler à ceci :

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = utilisateur@gmail.com
  pass = x67Nrs,/V:Xt84N
  port = 993
  sslverify = false
```

Si votre serveur IMAP n'utilise pas SSL, les deux dernières lignes ne sont probablement pas nécessaires et le paramètre `host` commencera par `imap://` au lieu de `imaps://`. Quand c'est fait, vous pouvez utiliser la commande `git imap-send` pour placer la série de patchs dans le répertoire *Drafts* du serveur IMAP spécifié :

```
$ cat *.patch | git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

À ce stade, vous devriez être capable d'aller dans votre dossier « Brouillons », remplacer le champ « Destinataire » par la liste de diffusion à laquelle vous envoyez le patch, peut-être mettre en copie le mainteneur ou la personne responsable de cette section, et l'envoyer.

Vous pouvez aussi envoyer les patchs via un serveur SMTP. Comme précédemment, vous pouvez définir chaque valeur séparément avec une série de commandes `git config`, ou vous pouvez les ajouter manuellement dans la section « `sendmail` » dans votre fichier `~/.gitconfig` :

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

Après que ceci soit fait, vous pouvez utiliser `git send-email` pour envoyer vos patchs :

```
$ git send-email *.patch
0001-Ajout-d-une-limite-la-fonction-de-log.patch
0002-change-la-largeur-du-log-de-25-a-30.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Ensuite, Git crache un certain nombre d'informations qui ressemblent à ceci pour chaque patch à envoyer :

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

À présent, vous devriez pouvoir vous rendre dans le répertoire *Drafts*, changer le champ destinataire pour celui de la liste de diffusion, y ajouter optionnellement en copie le mainteneur du projet ou le responsable et l'envoyer.

Résumé

Ce chapitre a traité quelques-unes des méthodes communes de gestion de types différents de projets Git que vous pourrez rencontrer et a introduit un certain nombre de nouveaux outils pour vous aider à gérer ces processus. Dans la section suivante, nous allons voir comment travailler de l'autre côté de la barrière : en tant que mainteneur de projet Git. Vous apprendrez comment travailler comme dictateur bienveillant ou gestionnaire d'intégration.

Maintenance d'un projet

En plus de savoir comment contribuer efficacement à un projet, vous aurez probablement besoin de savoir comment en maintenir un. Cela peut consister à accepter et appliquer les patchs générés via `format-patch` et envoyés par courriel, ou à intégrer des modifications dans des branches distantes de dépôts distants. Que vous mainteniez le dépôt de référence ou que vous souhaitiez aider en vérifiant et approuvant les patchs, vous devez savoir comment accepter les contributions d'une manière limpide pour vos contributeurs et soutenable à long terme pour vous.

Travail dans des branches thématiques

Quand vous vous apprêtez à intégrer des contributions, une bonne idée consiste à les essayer d'abord dans une branche thématique, une branche temporaire spécifiquement créée pour essayer cette nouveauté. De cette manière, il est plus facile de rectifier un patch à part et de le laisser s'il ne fonctionne pas jusqu'à ce que vous disposiez de temps pour y travailler. Si vous créez une simple branche nommée d'après le thème de la modification que vous allez essayer, telle que `ruby_client` ou quelque chose d'aussi descriptif, vous pouvez vous en souvenir simplement plus tard. Le mainteneur du projet Git a l'habitude d'utiliser des espaces de nommage pour ses branches, tels que `sc/ruby_client`, où `sc` représente les initiales de la personne qui a fourni le travail. Comme vous devez vous en souvenir, on crée une branche à partir de `master` de la manière suivante :

```
$ git branch sc/ruby_client master
```

Ou bien, si vous voulez aussi basculer immédiatement dessus, vous pouvez utiliser l'option `checkout -b` :

```
$ git checkout -b sc/ruby_client master
```

Vous voilà maintenant prêt à ajouter les modifications sur cette branche thématique et à déterminer si c'est prêt à être fusionné dans les branches au long cours.

Application des patchs à partir de courriel

Si vous recevez un patch par courriel et que vous devez l'intégrer dans votre projet, vous devez l'appliquer dans une branche thématique pour l'évaluer. Il existe deux moyens d'appliquer un patch reçu par courriel : `git apply` et `git am`.

Application d'un patch avec `apply`

Si vous avez reçu le patch de quelqu'un qui l'a généré avec la commande `git diff` ou `diff` Unix, vous pouvez l'appliquer avec la commande `git apply`. Si le patch a été sauvé comme fichier `/tmp/patch-ruby-client.patch`, vous pouvez l'appliquer comme ceci :

```
$ git apply /tmp/patch-ruby-client.patch
```

Les fichiers dans votre copie de travail sont modifiés. C'est quasiment identique à la commande `patch -p1` qui applique directement les patches mais en plus paranoïaque et moins tolérant sur les concordances approximatives. Les ajouts, effacements et renommages de fichiers sont aussi gérés s'ils sont décrits dans le format `git diff`, ce que `patch` ne supporte pas. Enfin, `git apply` fonctionne en mode « applique tout ou refuse tout » dans lequel toutes les modifications proposées sont appliquées si elles le peuvent, sinon rien n'est modifié, là où `patch` peut n'appliquer que partiellement les patches, laissant le répertoire de travail dans un état intermédiaire. `git apply` est par-dessus tout plus paranoïaque que `patch`. Il ne créera pas une validation à votre place : après l'avoir lancé, vous devrez indexer et valider les modifications manuellement.

Vous pouvez aussi utiliser `git apply` pour voir si un patch s'applique proprement avant de réellement l'appliquer — vous pouvez lancer `git apply --check` avec le patch :

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

S'il n'y pas de message, le patch devrait s'appliquer proprement. Cette commande se termine avec un statut non-nul si la vérification échoue et vous pouvez donc l'utiliser dans des scripts.

Application d'un patch avec `am`

Si le contributeur est un utilisateur de Git qui a été assez gentil d'utiliser la commande `format-patch` pour générer ses patches, votre travail sera facilité car le patch contient alors déjà l'information d'auteur et le message de validation. Si possible, encouragez vos contributeurs à utiliser `format-patch` au lieu de `patch` pour générer les patches qu'ils vous adressent. Vous ne devriez avoir à n'utiliser `git apply` que pour les vrais patches.

Pour appliquer un patch généré par `format-patch`, vous utilisez `git am`. Techniquement, `git am` s'attend à lire un fichier au format mbox, qui est un format texte simple permettant de stocker un ou plusieurs courriels dans un unique fichier texte. Il ressemble à ceci :

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

C'est le début de ce que la commande `format-patch` affiche, comme vous avez vu dans la section précédente. C'est aussi un format courriel mbox parfaitement valide. Si quelqu'un vous a envoyé par courriel un patch correctement formaté en utilisant `git send-mail` et que vous le téléchargez en format mbox, vous pouvez pointer `git am` sur ce fichier mbox et il commencera à appliquer tous les patches contenus. Si vous utilisez un client courriel qui sait sauver plusieurs messages au format mbox, vous pouvez sauver la totalité de la série de patches dans un fichier et utiliser `git am` pour les appliquer tous en une fois.

Néanmoins, si quelqu'un a déposé un fichier de patch généré via `format-patch` sur un système de suivi de faits techniques ou quelque chose de similaire, vous pouvez toujours sauvegarder le fichier localement et le passer à `git am` pour l'appliquer :

```
$ git am 0001-limit-log-function.patch
Application : add limit to log function
```

Vous remarquez qu'il s'est appliqué proprement et a créé une nouvelle validation pour vous. L'information d'auteur est extraite des en-têtes `From` et `Date` tandis que le message de validation est repris du champ `Subject` et du corps (avant le patch) du message. Par exemple, si le patch est appliqué depuis le fichier mbox ci-dessus, la validation générée ressemblerait à ceci :

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author: Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit: Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

    add limit to log function

    Limit log functionality to the first 20
```

L'information `Commit` indique la personne qui a appliqué le patch et la date d'application. L'information `Author` indique la personne qui a créé le patch et la date de création.

Il reste la possibilité que le patch ne s'applique pas proprement. Peut-être votre branche principale a-t-elle déjà trop divergé de la branche sur laquelle le patch a été construit, ou peut-être que le patch dépend d'un autre patch qui n'a pas encore été appliqué. Dans ce cas, le processus de `git am` échouera et vous demandera ce que vous souhaitez faire :

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Application : seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Le patch a échoué à 0001.
Lorsque vous aurez résolu ce problème, lancez "git am --continue".
Si vous préférez sauter ce patch, lancez "git am --skip" à la place.
Pour restaurer la branche d'origine et stopper le patchage, lancez
"git am --abort".
```

Cette commande introduit des marqueurs de conflit dans tous les fichiers qui ont généré un problème, de la même manière qu'un conflit de fusion ou de rebasage. Vous pouvez résoudre les problèmes de manière identique — éditez le fichier pour résoudre les conflits, indexez le nouveau fichier, puis lancez `git am --resolved` ou `git am --continue` pour continuer avec le patch suivant :

```
$ (correction du fichier)
$ git add ticgit.gemspec
$ git am --continue
Applying: seeing if this helps the gem
```

Si vous souhaitez que Git essaie de résoudre les conflits avec plus d'intelligence, vous pouvez passer l'option `-3` qui demande à Git de tenter une fusion à trois sources. Cette option n'est pas active par défaut parce qu'elle ne fonctionne pas si le `commit` sur lequel le patch indique être basé n'existe pas dans votre dépôt. Si par contre, le patch est basé sur un `commit` public, l'option `-3` est généralement beaucoup plus fine pour appliquer des patches conflictuels :

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
```

```
No changes -- Patch already applied.
```

Dans ce cas, je cherchais à appliquer un patch qui avait déjà été intégré. Sans l'option `-3`, cela aurait ressemblé à un conflit.

Si vous appliquez des patches à partir d'un fichier mbox, vous pouvez aussi lancer la commande `am` en mode interactif qui s'arrête à chaque patch trouvé et vous demande si vous souhaitez l'appliquer :

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

C'est agréable si vous avez un certain nombre de patches sauvegardés parce que vous pouvez voir les patches pour vous rafraîchir la mémoire et ne pas les appliquer s'ils ont déjà été intégrés.

Quand tous les patches pour votre sujet ont été appliqués et validés dans votre branche, vous pouvez choisir si et comment vous souhaitez les intégrer dans une branche au long cours.

Vérification des branches distantes

Si votre contribution a été fournie par un utilisateur de Git qui a mis en place son propre dépôt public sur lequel il a poussé ses modifications et vous a envoyé l'URL du dépôt et le nom de la branche distante, vous pouvez les ajouter en tant que dépôt distant et réaliser les fusions localement.

Par exemple, si Jessica vous envoie un courriel indiquant qu'elle a une nouvelle fonctionnalité géniale dans la branche `ruby-client` de son dépôt, vous pouvez la tester en ajoutant le dépôt distant et en tirant la branche localement :

```
$ git remote add jessica git://github.com/jessica/monproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Si elle vous envoie un autre mail indiquant une autre branche contenant une autre fonctionnalité géniale, vous pouvez la récupérer et la tester simplement à partir de votre référence distante.

C'est d'autant plus utile si vous travaillez en continu avec une personne. Si quelqu'un n'a qu'un seul patch à contribuer de temps en temps, l'accepter via courriel peut s'avérer moins consommateur en temps de préparation du serveur public, d'ajout et retrait de branches distantes juste pour tirer quelques patches. Vous ne souhaiteriez sûrement pas devoir gérer des centaines de dépôts distants pour intégrer à chaque fois un ou deux patches. Néanmoins, des scripts et des services hébergés peuvent rendre cette tâche moins ardue. Cela dépend largement de votre manière de développer et de celle de vos contributeurs.

Cette approche a aussi l'avantage de vous fournir l'historique des validations. Même si vous pouvez rencontrer des problèmes de fusion légitimes, vous avez l'information dans votre historique de la base ayant servi pour les modifications contribuées. La fusion à trois sources est choisie par défaut plutôt que d'avoir à spécifier l'option `-3` en espérant que le patch a été généré à partir d'un instantané public auquel vous auriez accès.

Si vous ne travaillez pas en continu avec une personne mais souhaitez tout de même tirer les modifications de cette manière, vous pouvez fournir l'URL du dépôt distant à la commande `git pull`. Cela permet de réaliser un tirage unique sans sauver l'URL comme référence distante :

```
$ git pull https://github.com/pourunefois/projet
From https://github.com/onetimeguy/project
* branch          HEAD      -> FETCH_HEAD
Merge made by recursive.
```

Déterminer les modifications introduites

Vous avez maintenant une branche thématique qui contient les contributions. À partir de là, vous pouvez déterminer ce que vous souhaitez en faire. Cette section revisite quelques commandes qui vont vous permettre de faire une revue de ce que vous allez exactement introduire si vous fusionnez dans la branche principale.

Faire une revue de tous les *commits* dans cette branche s'avère souvent d'une grande aide. Vous pouvez exclure les *commits* de la branche `master` en ajoutant l'option `--not` devant le nom de la branche. C'est équivalent au format `master..contrib` utilisé plus haut. Par exemple, si votre contributeur vous envoie deux patches et que vous créez une branche appelée `contrib` et y appliquez ces patches, vous pouvez lancer ceci :

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

Pour visualiser les modifications que chaque **commit** introduit, souvenez-vous que vous pouvez passer l'option `-p` à `git log` et elle ajoutera le diff introduit à chaque *commit*.

Pour visualiser un diff complet de ce qui arriverait si vous fusionniez cette branche thématique avec une autre branche, vous pouvez utiliser un truc bizarre pour obtenir les résultats corrects. Vous pourriez penser à lancer ceci :

```
$ git diff master
```

Cette commande affiche un diff mais elle peut être trompeuse. Si votre branche `master` a avancé depuis que vous avez créé la branche thématique, vous obtiendrez des résultats apparemment étranges. Cela arrive parce que Git compare directement l'instantané de la dernière validation sur la branche thématique et celui de la dernière validation sur la branche `master`. Par exemple, si vous avez ajouté une ligne dans un fichier sur la branche `master`, une comparaison directe donnera l'impression que la branche thématique va retirer cette ligne.

Si `master` est un ancêtre direct de la branche thématique, ce n'est pas un problème. Si les deux historiques ont divergé, le diff donnera l'impression que vous ajoutez toutes les nouveautés de la branche thématique et retirez tout ce qui a été fait depuis dans la branche `master`.

Ce que vous souhaitez voir en fait, ce sont les modifications ajoutées sur la branche thématique — le travail que vous introduirez si vous fusionnez cette branche dans `master`. Vous obtenez ce résultat en demandant à Git de comparer le dernier instantané de la branche thématique avec son ancêtre commun à la branche `master` le plus récent.

Techniquement, c'est réalisable en déterminant exactement l'ancêtre commun et en lançant la commande `diff` dessus :

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Néanmoins, comme ce n'est pas très commode, Git fournit un raccourci pour réaliser la même chose : la syntaxe à trois points. Dans le contexte de la commande `diff`, vous pouvez placer trois points après une autre branche pour réaliser un `diff` entre le dernier instantané de la branche sur laquelle vous vous trouvez et son ancêtre commun avec une autre branche :

```
$ git diff master...contrib
```

Cette commande ne vous montre que les modifications que votre branche thématique a introduites depuis son ancêtre commun avec `master`. C'est une syntaxe très simple à retenir.

Intégration des contributions

Lorsque tout le travail de votre branche thématique est prêt à être intégré dans la branche principale, il reste à savoir comment le faire. De plus, il faut connaître le mode de gestion que vous souhaitez pour votre projet. Vous avez de nombreux choix et je vais en traiter quelques-uns.

Modes de fusion

Un mode simple fusionne votre travail dans la branche `master`. Dans ce scénario, vous avez une branche `master` qui contient le code stable. Quand vous avez des modifications prêtes dans une branche thématique, vous la fusionnez dans votre branche `master` puis effacez la branche thématique, et ainsi de suite. Si vous avez un dépôt contenant deux branches nommées `ruby_client` et `php_client` qui ressemble à [Historique avec quelques branches thématiques](#), et que vous fusionnez `ruby_client` en premier, suivi de `php_client`, alors votre historique ressemblera à la fin à [Après fusion des branches thématiques..](#)

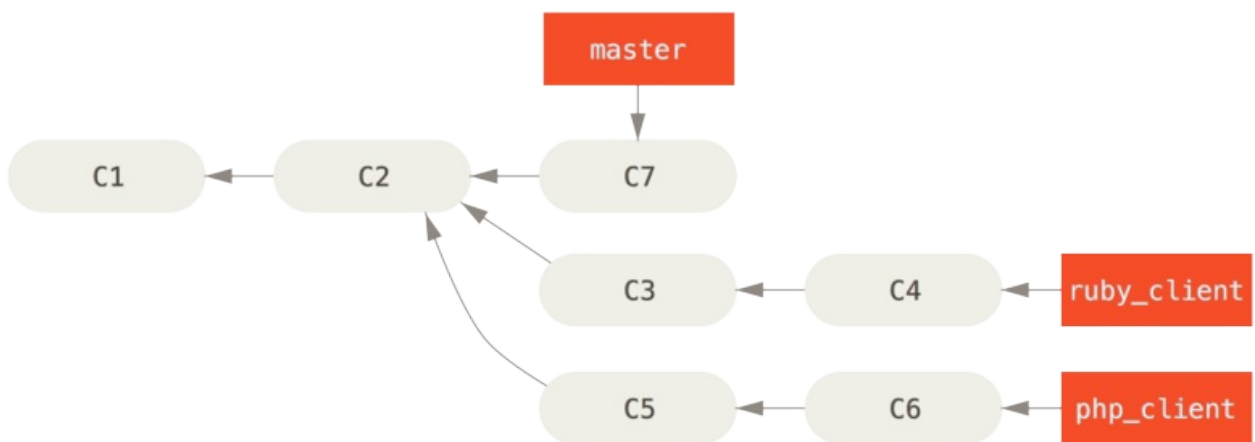


Figure 74 : Historique avec quelques branches thématiques.

Figure 73. Historique avec quelques branches thématiques.

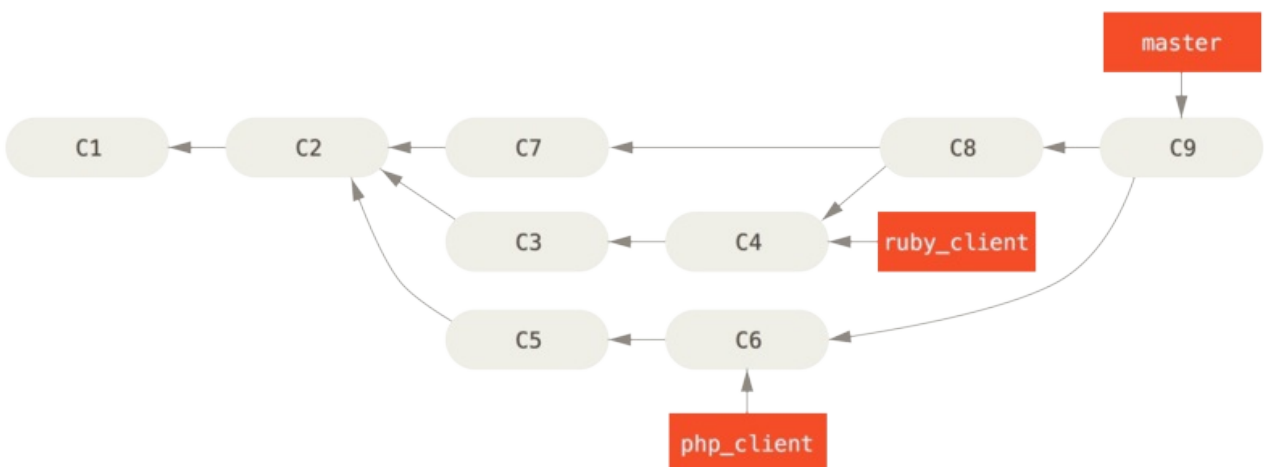


Figure 75 : Après fusion des branches thématiques.

Figure 74. Après fusion des branches thématiques.

C'est probablement le mode le plus simple mais cela peut s'avérer problématique si vous avez à gérer des dépôts ou des projets plus gros pour lesquels vous devez être circonspect sur ce que vous acceptez.

Si vous avez plus de développeurs ou un projet plus important, vous souhaitez probablement utiliser un cycle de fusion à deux étapes. Dans ce scénario, vous avez deux branches au long cours, `master` et `develop`, dans lequel vous déterminez que `master` est mis à jour seulement lors d'une version vraiment stable et tout le nouveau code est intégré dans la branche `develop`. Vous poussez régulièrement ces deux branches sur le dépôt public. Chaque fois que vous avez une nouvelle branche thématique à fusionner ([Avant la fusion d'une branche thématique.](#)), vous la fusionnez dans `develop` ([Après la fusion d'une branche thématique.](#)). Puis, lorsque vous étiquetez une version majeure, vous mettez `master` à niveau avec l'état stable de `develop` en avance rapide ([Après une publication d'une branche thématique.](#)).

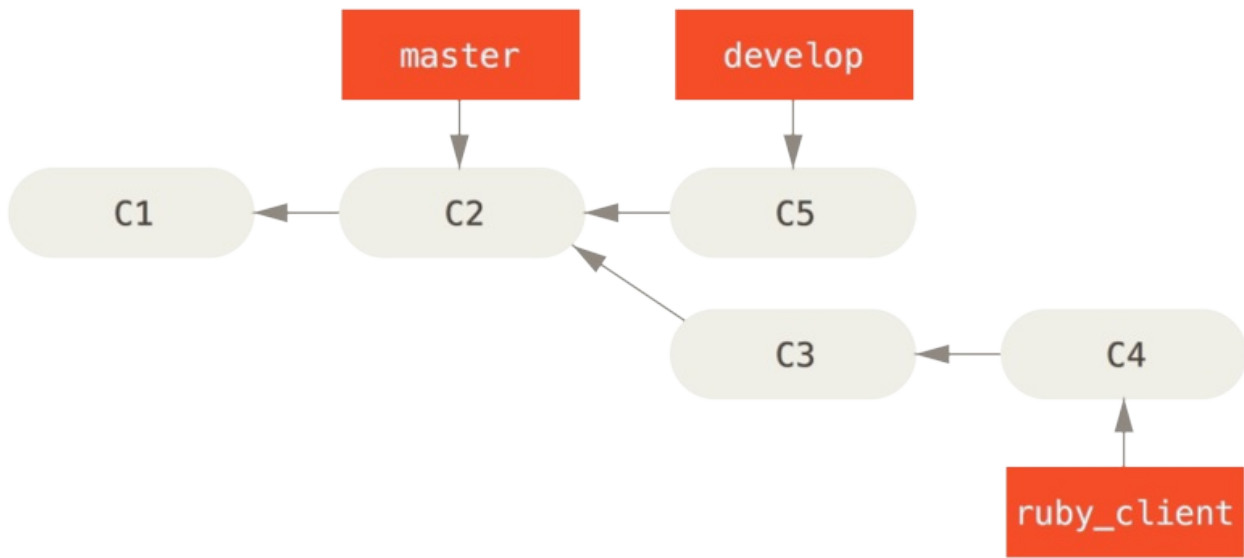


Figure 76 : Avant la fusion d'une branche thématique.

Figure 75. Avant la fusion d'une branche thématique.

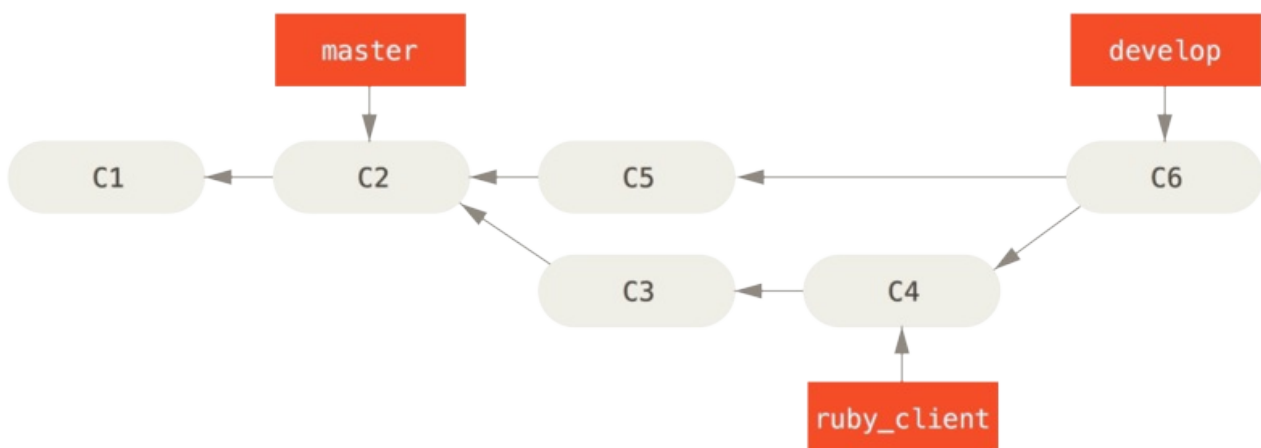


Figure 77 : Après la fusion d'une branche thématique.

Figure 76. Après la fusion d'une branche thématique.

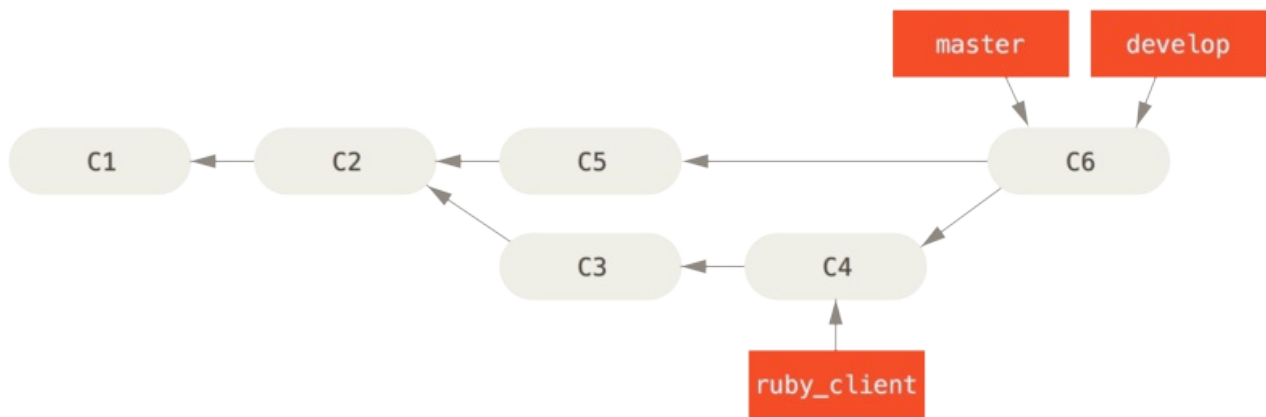


Figure 78 : Après une publication d'une branche thématique.

Figure 77. Après une publication d'une branche thématique.

Ainsi, lorsque l'on clone le dépôt de votre projet, on peut soit extraire la branche `master` pour construire la dernière version stable et mettre à jour facilement ou on peut extraire la branche `develop` qui représente le nec plus ultra du développement.

Vous pouvez aussi continuer ce concept avec une branche d'intégration où tout le travail est fusionné. Alors, quand la base de code sur cette branche est stable et que les tests passent, vous la fusionnez dans la branche `develop`. Quand cela s'est avéré stable pendant un certain temps, vous mettez à jour la branche `master` en avance rapide.

Gestions avec nombreuses fusions

Le projet Git dispose de quatre branches au long cours : `master`, `next`, `pu` (*proposed updates* : propositions) pour les nouveaux travaux et `maint` pour les backports de maintenance. Quand une nouvelle contribution est proposée, elle est collectée dans des branches thématiques dans le dépôt du mainteneur d'une manière similaire à ce que j'ai décrit ([Série complexe de branches thématiques contribuées en parallèle](#)). À ce point, les fonctionnalités sont évaluées pour déterminer si elles sont stables et prêtes à être consommées ou si elles nécessitent un peaufinage. Si elles sont stables, elles sont fusionnées dans `next` et cette branche est poussée sur le serveur public pour que tout le monde puisse essayer les fonctionnalités intégrées ensemble.

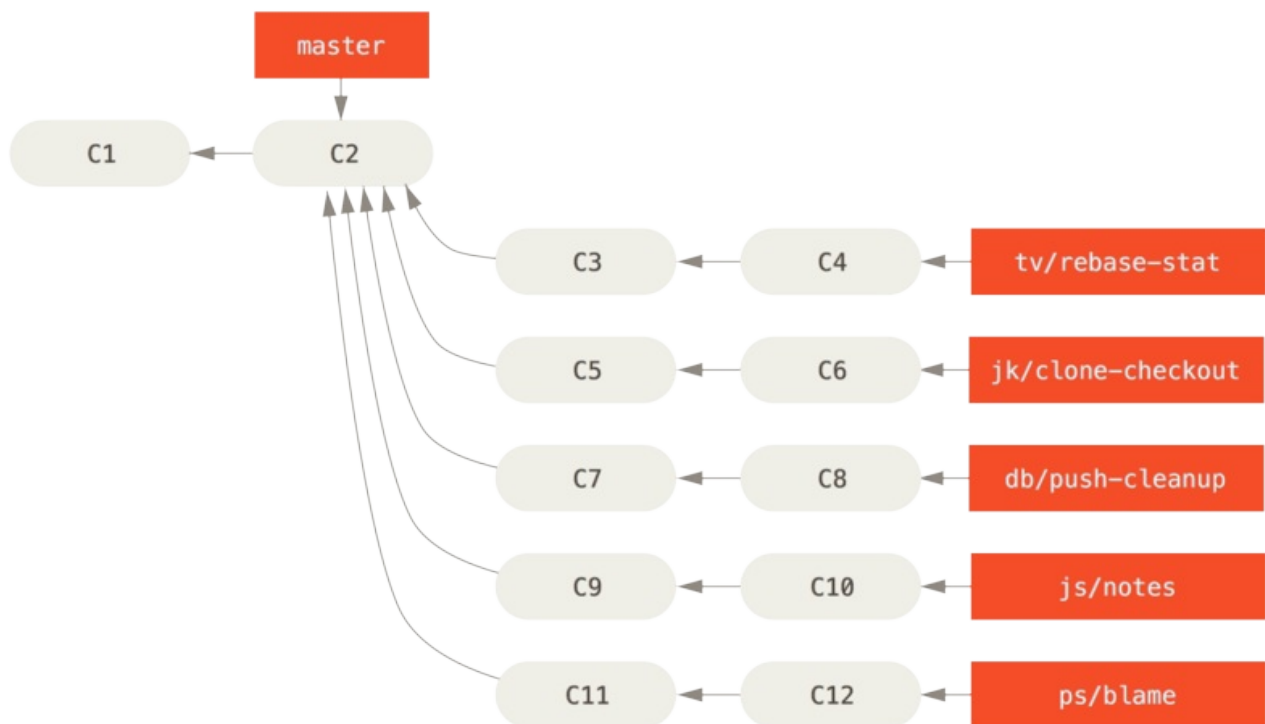


Figure 79 : Série complexe de branches thématiques contribuant en parallèle.

Figure 78. Série complexe de branches thématiques contribuant en parallèle.

Si les fonctionnalités nécessitent encore du travail, elles sont fusionnées plutôt dans `pu`. Quand elles sont considérées comme totalement stables, elles sont re-fusionnées dans `master` et sont alors reconstruites à partir des fonctionnalités qui résidaient dans `next` mais n'ont pu intégrer `master`. Cela signifie que `master` évolue quasiment toujours en mode avance rapide, tandis que `next` est rebasé assez souvent et `pu` est rebasé encore plus souvent :

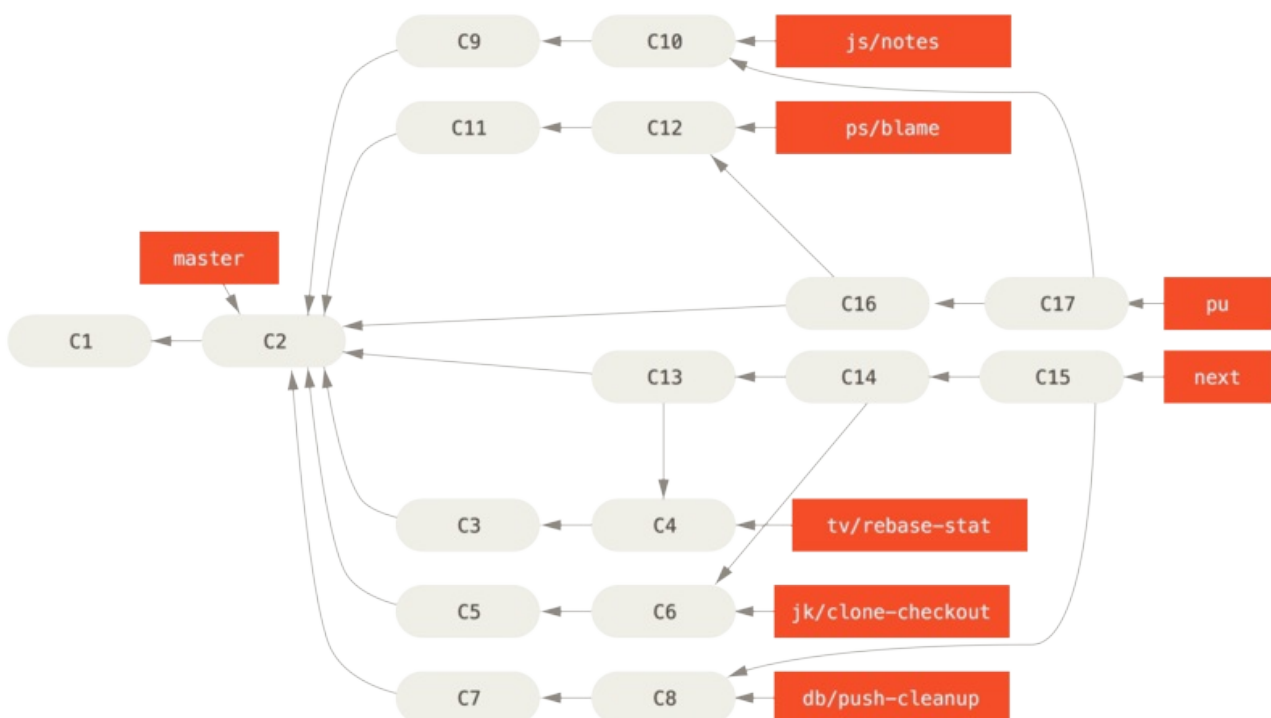


Figure 80 : Fusion des branches thématiques dans les branches à long terme.

Figure 79. Fusion des branches thématiques dans les branches à long terme.

Quand une branche thématique a finalement été fusionnée dans `master`, elle est effacée du dépôt. Le projet Git a aussi une branche `maint` qui est créée à partir de la dernière version pour fournir des patches correctifs en cas de besoin de version de maintenance. Ainsi, quand vous clonez le dépôt de Git, vous avez quatre branches disponibles pour évaluer le projet à différentes étapes de développement, selon le niveau de développement que vous souhaitez utiliser ou pour lequel vous souhaitez contribuer. Le mainteneur a une gestion structurée qui lui permet d'évaluer et sélectionner les nouvelles contributions.

Gestion par rebasage et sélection de *commit*

D'autres mainteneurs préfèrent rebaser ou sélectionner les contributions sur le sommet de la branche `master`, plutôt que les fusionner, de manière à conserver un historique à peu près linéaire. Lorsque plusieurs modifications sont présentes dans une branche thématique et que vous souhaitez les intégrer, vous vous placez sur cette branche et vous lancez la commande `rebase` pour reconstruire les modifications à partir du sommet courant de la branche `master` (ou `develop`, ou autre). Si cela fonctionne correctement, vous pouvez faire une avance rapide sur votre branche `master` et vous obtenez finalement un historique de projet linéaire.

L'autre moyen de déplacer des modifications introduites dans une branche vers une autre consiste à les sélectionner ou les picorer (`cherry-pick`). Un picorage dans Git ressemble à un rebasage appliqué à un *commit* unique. Cela consiste à prendre le patch qui a été introduit lors d'une validation et à essayer de l'appliquer sur la branche sur laquelle on se trouve. C'est très utile si on a un certain nombre de *commits* sur une branche thématique et que l'on veut n'en intégrer qu'un seul, ou si on n'a qu'un *commit* sur une branche thématique et qu'on préfère le sélectionner plutôt que de lancer `rebase`. Par exemple, supposons que vous ayez un projet ressemblant à ceci :

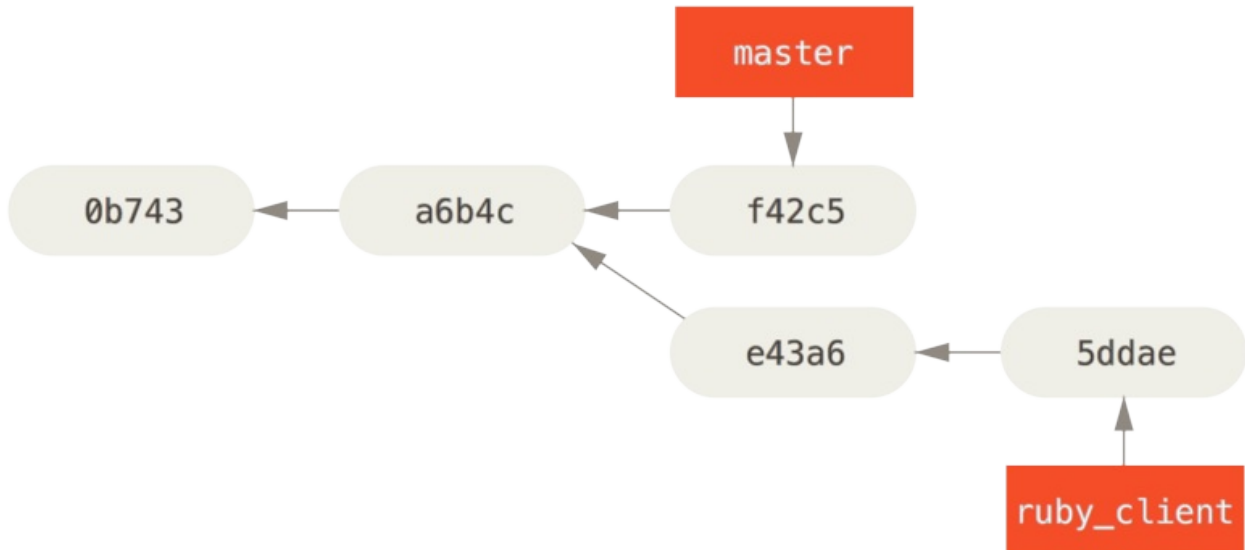


Figure 81 : Historique d'exemple avant une sélection.

Figure 80. Historique d'exemple avant une sélection.

Si vous souhaitez tirer le *commit* `e43a6` dans votre branche `master`, vous pouvez lancer :

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

La même modification que celle introduite en `e43a6` est tirée mais vous obtenez une nouvelle valeur de SHA-1 car les dates d'application sont différentes. À présent, votre historique ressemble à ceci :

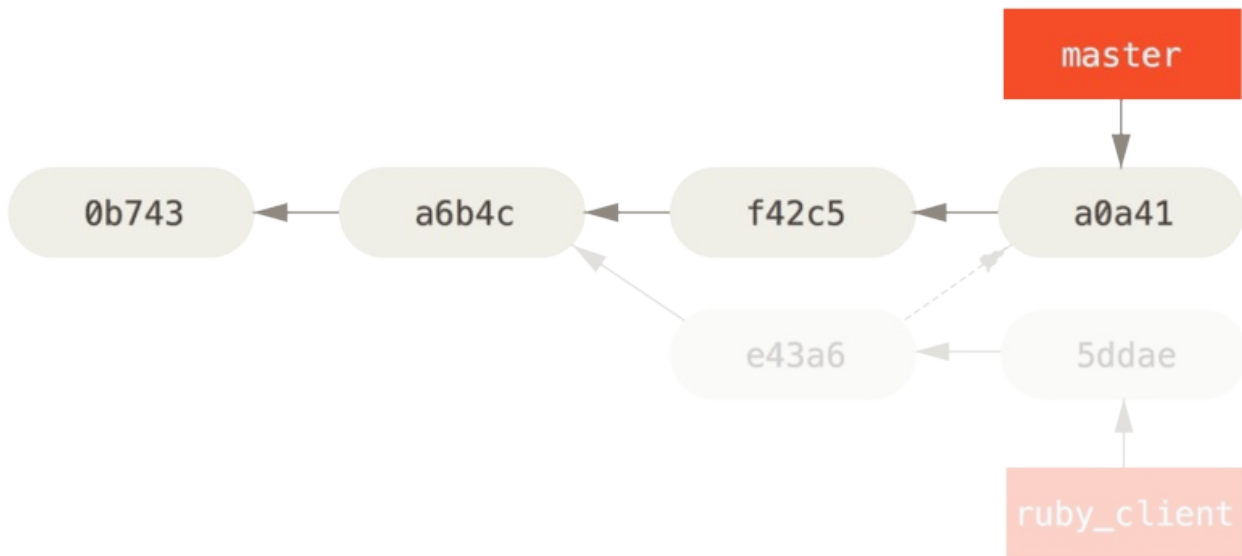


Figure 82 : Historique après sélection d'un `\commit\` dans une branche thématique.

Figure 81. Historique après sélection d'un `commit` dans une branche thématique.

Maintenant, vous pouvez effacer votre branche thématique et abandonner les `commits` que vous n'avez pas tirés dans `master` .

Rerere

Si vous fusionnez et rebasez beaucoup ou si vous maintenez une branche au long cours, la fonctionnalité appelée « rerere » peut s'avérer utile.

Rerere signifie « ré utiliser les ré solutions en re gistrées » (*"reuse recorded resolution"*) - c'est un moyen de raccourcir les résolutions manuelles de conflit. Quand rerere est actif, Git va conserver un jeu de couples d'images pré et post fusion des fichiers ayant présenté des conflits, puis s'il s'aperçoit qu'un conflit ressemble à une de ces résolutions, il va utiliser la même stratégie sans rien vous demander.

Cette fonctionnalité se traite en deux phases : une étape de configuration et une commande. L'étape de configuration est `rerere.enabled` qui active la fonction et qu'il est facile de placer en config globale :

```
$ git config --global rerere.enabled true
```

Ensuite, quand vous fusionnez en résolvant des conflits, la résolution sera enregistrée dans le cache pour un usage futur.

Si besoin, vous pouvez interagir avec le cache rerere au moyen de la commande `git rerere` .

Quand elle est invoquée telle quelle, Git vérifie sa base de données de résolutions et essaie de trouver une correspondance avec les conflits en cours et les résout (bien que ce soit automatique si `rerere.enabled` est à `true`). Il existe aussi des sous-commandes permettant de voir ce qui sera enregistré, d'effacer du cache une résolution spécifique ou d'effacer entièrement le cache. rerere est traité plus en détail dans [Rerere](#).

Étiquetage de vos publications

Quand vous décidez de créer une publication de votre projet, vous souhaiterez probablement étiqueter le projet pour pouvoir recréer cette version dans le futur. Vous pouvez créer une nouvelle étiquette (*tag*) telle que décrite dans [Les bases de Git](#). Si vous décidez de signer l'étiquette en tant que mainteneur, la commande ressemblera à ceci :

```
$ git tag -s v1.5 -m 'mon etiquette v1.5 signée'
Une phrase secrète est nécessaire pour déverrouiller la clef secrète de
l'utilisateur : "Scott Chacon <schacon@gmail.com>"
clé DSA de 1024 bits, identifiant F721C45A, créée le 2009-02-09
```

Si vous signez vos étiquettes, vous rencontrerez le problème de la distribution de votre clé publique PGP permettant de vérifier la signature. Le mainteneur du projet Git a résolu le problème en incluant la clé publique comme blob dans le dépôt et en ajoutant une étiquette qui pointe directement sur ce contenu. Pour faire de même, vous déterminez la clé de votre trousseau que vous voulez publier en lançant `gpg --list-keys` :

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
-----
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid Scott Chacon <schacon@gmail.com>
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Ensuite, vous pouvez importer la clé directement dans la base de données Git en l'exportant de votre trousseau et en la redirigeant dans `git hash-object` qui écrit un nouveau blob avec son contenu dans Git et vous donne en sortie le SHA-1 du blob :

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

À présent, vous avez le contenu de votre clé dans Git et vous pouvez créer une étiquette qui pointe directement dessus en spécifiant la valeur SHA-1 que la commande `hash-object` vous a fournie :

```
$ git tag -a maintainer-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Si vous lancez `git push --tags`, l'étiquette `maintainer-gpg-pub` sera partagée publiquement. Un tiers pourra vérifier une étiquette après import direct de votre clé publique PGP, en extrayant le blob de la base de donnée et en l'important dans GPG :

```
$ git show maintainer-gpg-pub | gpg --import
```

Il pourra alors utiliser cette clé pour vérifier vos étiquettes signées. Si de plus, vous incluez des instructions d'utilisation pour la vérification de signature dans le message d'étiquetage, l'utilisateur aura accès à ces informations en lançant la commande `git show <étiquette>`.

Génération d'un nom de révision

Comme Git ne fournit pas par nature de nombres croissants tels que « r123 » à chaque validation, la commande `git describe` permet de générer un nom humainement lisible pour chaque *commit*. Git concatène le nom de l'étiquette la plus proche, le nombre de validations depuis cette étiquette et un code SHA-1 partiel du *commit* que l'on cherche à définir :

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

De cette manière, vous pouvez exporter un instantané ou le construire et le nommer de manière intelligible. En fait, si Git est construit à partir du source cloné depuis le dépôt Git, `git --version` vous donne exactement cette valeur. Si vous demandez la description d'un instantané qui a été étiqueté, le nom de l'étiquette est retourné.

La commande `git describe` repose sur les étiquettes annotées (étiquettes créées avec les options `-a` ou `-s`). Les étiquettes de publication doivent donc être créées de cette manière si vous souhaitez utiliser `git describe` pour garantir que les *commits* seront décrits correctement. Vous pouvez aussi utiliser ces noms comme cible lors d'une extraction ou d'une commande `show`, bien qu'ils reposent sur le SHA-1 abrégé et pourraient ne pas rester valides indéfiniment. Par exemple, le noyau Linux a sauté dernièrement de 8 à 10 caractères pour assurer l'unicité des objets SHA-1 et les anciens noms `git describe` sont par conséquent devenus invalides.

Préparation d'une publication

Maintenant, vous voulez publier une version. Une des étapes consiste à créer une archive du dernier instantané de votre code pour les malheureux qui n'utilisent pas Git. La commande dédiée à cette action est `git archive` :

```
$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Lorsqu'on ouvre l'archive, on obtient le dernier instantané du projet sous un répertoire `projet`. On peut aussi créer une archive au format zip de manière similaire en passant l'option `--format=zip` à la commande `git archive` :

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Voilà deux belles archives tar.gz et zip de votre projet prêtes à être téléchargées sur un site web ou envoyées par courriel.

Shortlog

Il est temps d'envoyer une annonce à la liste de diffusion des nouveautés de votre projet. Une manière simple d'obtenir rapidement une sorte de liste des modifications depuis votre dernière version ou courriel est d'utiliser la commande `git shortlog`. Elle résume toutes les validations dans l'intervalle que vous lui spécifiez. Par exemple, ce qui suit vous donne un résumé de toutes les validations depuis votre dernière version si celle-ci se nomme v1.0.1 :

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray `puts`
    Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

Vous obtenez ainsi un résumé clair de toutes les validations depuis v1.0.1, regroupées par auteur, prêt à être envoyé sur la liste de diffusion.

Résumé

Vous devriez à présent vous sentir à l'aise pour contribuer à un projet avec Git, mais aussi pour maintenir votre propre projet et intégrer les contributions externes. Félicitations, vous êtes un développeur Git efficace ! Au prochain chapitre, vous découvrirez des outils plus puissants pour gérer des situations complexes, qui feront de vous un maître de Git.