GIT Best Practice Guide

N° de la lecture individuelle :	1
Étudiant	Laurent Térence, 802_1F
Sujet	GIT Best practice Guide, Pidoux Eric, 2014, Packt Publishing, Anglais

Table des matières

1. Choix du sujet ... 3
2. Résumé de la lecture ... 4
   - 2.1 Introduction ... 4
   - 2.2 Overview of GIT ... 4
3. Starting a Git Repository ... 4
   - 3.1 Configuring Git ... 4
   - 3.2 Initializing a new repository ... 4
   - 3.3 Cloning an existent repository ... 5
   - 3.4 Working with the repository ... 5
     - 3.4.1 Adding a file ... 6
     - 3.4.2 Commiting a file ... 6
     - 3.4.3 Pushing a file ... 7
     - 3.4.4 Removing a file ... 8
     - 3.4.5 Checking the status ... 8
     - 3.4.6 Ignoring files ... 9
   - 3.5 Summary ... 10
4. Working in a Team Using Git ... 10
   - 4.1 Creating a server repository ... 10
   - 4.1.1 LOCAL ... 10
     - 4.1.2 SSH ... 10
     - 4.1.3 Git ... 11
     - 4.1.4 HTTPS ... 11
   - 4.2 Pushing data on remote repository – Jim case's ... 11
   - 4.3 Pulling data from the repository ... 12
   - 4.4 Creating a patch ... 12

4.5 Working with branches 12
4.5.1 Playing with branches 13
4.6 Tracking branches 13
4.6.1 Deleting a branch from the remote 14
4.7 Merging 14
4.7.1 Fast forward merge 14
4.7.2 Merge commit without fast-forward. 15
4.7.3 Other merging strategies 16
4.8 Rebase 17
4.9 Cherry-pick 19
4.10 (Using tags) 20
4.11 Summary 20
5. Finding and Resolving Conflicts 21
5.1 Finding content inside your repository 21
5.1.1 Searching file content 21
5.1.2 Exploring the repository history 22
5.1.3 Viewing changes 22
5.2 Stashing your changes 23
5.3 Cleaning your mistake 24
5.3.1 Reverting uncommitted changes 24
5.3.2 The git reset command 25
5.3.3 Editing a commit 26
5.3.4 Canceling a commit 27
5.3.5 Rewriting commit history 28
5.3.6 Solving merge conflicts 28
5.4 Fixing errors by practical examples 29
5.5 Summary 30
6. (Going Deeper into Git) 31
7. Using Git for Continuous Integration 31
7.1 Creating an efficient branching system 31
7.1.1 Git flow 31
7.1.2 BPF – Branch Per Feature 35
7.2 Working with continuous Integration using Git 37
7.3 Git tools you might like. 37
7.3.1 On Linux 37
7.3.2 On Windows 37

7.3.3 On Mac 37

1. Choix du sujet

Je suis actuellement en première année d'Informatique de Gestion et j'ai choisi de traiter les bases et les meilleures pratiques de GIT pour mon sujet de lecture individuelle. J'ai choisi ce sujet car nous avons besoin d'utiliser GIT en tant qu'équipe, et je souhaite être en mesure de comprendre et de contribuer efficacement à notre travail de groupe.

En tant qu'étudiant en informatique, je sais que GIT est un outil de gestion de version populaire et essentiel pour tout développeur travaillant sur un projet logiciel. Je pense que comprendre les concepts de base de GIT et les meilleures pratiques pour travailler en équipe peuvent m'aider à devenir un meilleur collaborateur et à contribuer de manière plus efficace à notre projet.

Je suis convaincu que la compréhension de GIT peut également être utile pour ma future carrière. Les compétences en gestion de version sont recherchées dans de nombreuses entreprises et industries, et la maîtrise de GIT peut me donner un avantage concurrentiel sur le marché du travail.

Dans le cadre de notre projet KOLOKA, nous avons un besoin crucial d'utiliser GIT pour la gestion de notre code source. Comme nous travaillons en équipe sur ce projet, il est essentiel d'avoir un système de contrôle de version efficace qui nous permet de travailler simultanément sur différentes parties du projet sans risquer de perdre ou d'écraser les modifications des autres

2. Résumé de la lecture

2.1 Introduction

Le contrôle de version est une méthode essentielle pour les développeurs afin de gérer efficacement les modifications apportées à un projet. L'un des systèmes de contrôle de version les plus populaires est Git. Ce système est simple, rapide, fiable et adapté à la gestion de projets de toutes tailles. Dans ce document, nous allons nous concentrer sur les principes et commandes de base de Git, ainsi que sur les meilleures pratiques pour son utilisation. Nous allons commencer par une vue d'ensemble de Git, avant de passer à la configuration d'un référentiel Git, puis nous verrons comment travailler avec Git en tant qu'individu et en équipe. Nous aborderons également la résolution des conflits et l'utilisation de Git pour l'intégration continue. Bien que nous n'allions pas trop en profondeur, ce document fournira un aperçu complet pour tout débutant qui souhaite commencer à utiliser Git pour ses projets.

2.2 Overview of GIT

Git est un système de contrôle de version largement utilisé qui permet aux développeurs de collaborer et de gérer les changements apportés à leur code au fil du temps. Il a été développé par Linus Torvalds.

3. Starting a Git Repository

3.1 Configuring Git

Avant de commencer à utiliser Git, il est important de configurer votre nom et votre adresse e-mail. Vous pouvez le faire en utilisant les commandes suivantes :

git config --global user.name "Votre nom"

git config --global user.email votre@adresse-mail.com

Ces informations sont importantes car elles seront enregistrées dans chaque commit que vous effectuerez avec Git. Elles permettent également à d'autres utilisateurs de savoir qui a effectué un certain changement dans le code. Il est donc important de s'assurer que ces informations sont correctes et à jour.

3.2 Initializing a new repository

Lorsque vous commencez un nouveau projet, vous pouvez initialiser un nouveau dépôt Git en utilisant la commande "git init". Si vous souhaitez créer un dépôt dans un projet existant, vous devez vous déplacer dans le répertoire du projet et exécuter la commande "git init ." Cette commande crée un dossier nommé ".git" dans le répertoire courant, qui contient les fichiers suivants utilisés par Git : Config, HEAD et le répertoire Refs qui contient les références à un commit pour une branche.

$ cd myProject

$ git init .

• Config : utilisé pour la configuration du dépôt Git local

• HEAD : liste le fichier de la branche actuelle

• Répertoire Refs : contient les références à un commit pour une branche

3.3 Cloning an existent repository

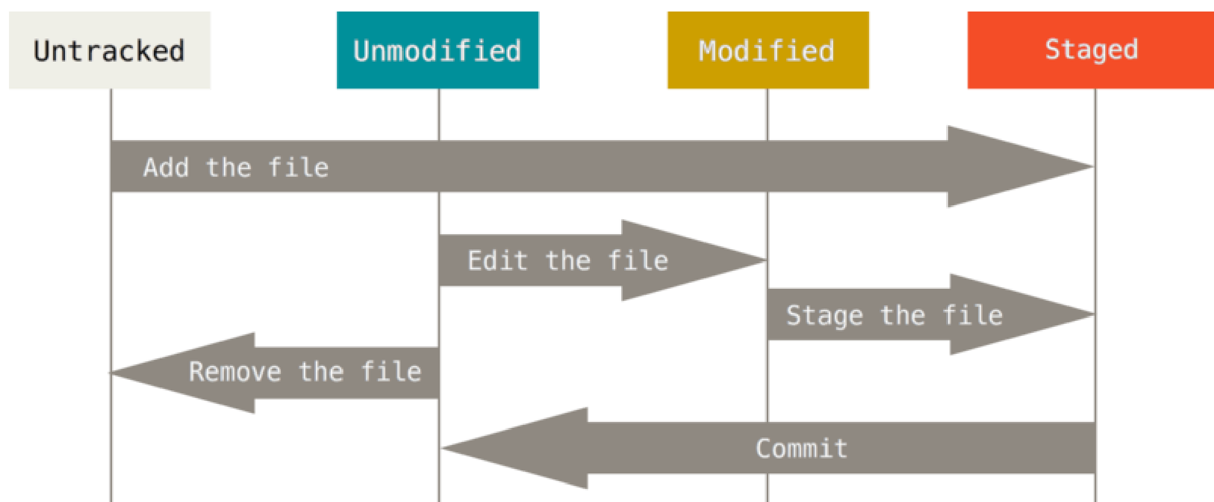
Avec Git, il est possible de cloner un référentiel existant pour y travailler. Il existe plusieurs possibilités pour cloner un référentiel, mais les protocoles http, git et ssh sont les plus couramment utilisés. Si le référentiel est public, Git créera un dossier et tout ce qui s'y trouve. Cependant, si le référentiel est privé ou protégé, vous devrez entrer des informations d'accès ou fournir une clé ssh privée. Par exemple, si vous voulez cloner un référentiel Symfony2, tapez cette ligne pour le cloner en utilisant "myProjectName" comme nom de dossier :

$ git clone https://github.com/symfony/symfony.git myProjectName

Notez que le nom après la commande clone est facultatif. S'il n'y a pas de paramètre après l'emplacement du référentiel, le nom du référentiel sera utilisé. Avant d'envoyer le contenu, Git compresse les objets pour accélérer la transmission. Nous verrons plus d'utilisations de la commande clone dans le prochain chapitre.

3.4 Working with the repository

Lorsque vous travaillez avec un dépôt Git, un fichier peut passer par plusieurs états différents, que vous pouvez contrôler grâce à certaines commandes. Le cycle de vie d'un fichier dans Git se divise en quatre états : **untracked, unmodified**, **modified**, et **staged**. Chaque fois que vous modifiez un fichier, il passe de l'état unmodified à modified, puis vous pouvez le préparer pour la prochaine validation en utilisant la commande "git add", qui mettra le fichier à l'état **staged**. Lorsque vous êtes satisfait de vos modifications, vous pouvez les valider en utilisant "git commit", qui ramènera le fichier à l'état unmodified. La boucle unmodified-modified-staged est continue, vous pouvez donc ajouter de nouvelles modifications à chaque fois. L'état untracked est celui dans lequel se trouve un fichier lorsque vous le créez, mais que Git ne le suit pas encore.



3.4.1 Adding a file

Lorsque vous créez un nouveau référentiel Git et ajoutez un fichier, il sera dans l'état "non suivi", ce qui signifie qu'il n'est pas dans le référentiel Git. Pour suivre un fichier, vous devez exécuter la commande `git add` en lui donnant le nom du fichier que vous voulez suivre. Une fois le fichier suivi, toutes les modifications apportées à ce fichier seront détectées par Git.

$ git add MyFileName.txt

Si vous souhaitez ajouter tous les fichiers d'un répertoire existant dans le référentiel Git, vous pouvez ajouter un point (`.`) après `git add` pour spécifier que tous les fichiers du répertoire courant doivent être ajoutés. Le fichier est alors prêt à être validé (ou "commité") dans le référentiel Git.

$ git add .

3.4.2 Commiting a file

Lorsque votre fichier est suivi, toutes les modifications seront notifiées par Git et vous devrez valider les changements sur le référentiel. Il est important de valider vos modifications dès que possible, sans pour autant le faire pour chaque ligne. La commande de validation (commit) est locale à votre propre référentiel, personne d'autre que vous ne peut la voir.

La commande commit propose différentes options. Par exemple, vous pouvez valider un fichier en utilisant la commande suivante :

$ git commit -m 'Ce message explique les changements' MonNomDeFichier.txt

Pour valider tous les fichiers, utilisez la commande suivante :

$ git commit -m 'Mon message de validation'

Cela créera un nouvel objet de validation (commit) dans le référentiel Git. Cette validation est référencée par une somme de contrôle SHA-1 et comprend différentes données (fichiers de contenu, répertoires de contenu, historique de validation, personne ayant validé les modifications, etc.). Vous pouvez afficher ces informations en exécutant la commande suivante :

$ git log

Cela affichera quelque chose de similaire à ceci :

Commit f658e5f22afe12ce75cde1h671b58d6703ab83f5

Auteur : Eric Pidoux contact@eric-pidoux.com

Date : Mon Jun 2 22 :54:04 2014 +0100

Mon message de validation

Le fichier est dans l'état "non modifié" car vous venez de valider le changement. Vous pouvez maintenant envoyer les fichiers dans le référentiel

3.4.3 Pushing a file

Une fois que vous avez commité vos modifications, vous pouvez les envoyer dans le dépôt distant (remote repository). Cela peut être un dépôt nu (bare repository) en utilisant init avec la commande git init --bare, il suffit alors de taper la commande suivante :

$ git push /home/erik/remote-repository.git.

Si vous créez un dépôt distant sur un autre serveur, vous devez configurer votre dépôt Git local. Si vous utilisez Git 2.0 ou une version ultérieure, la commande précédente affichera quelque chose comme ceci : Attention : push.default n'est pas défini ; sa valeur implicite changera dans Git 2.0 de « matching » en « simple ». *« Warning: push.default is unset; its implicit value is changing in*

Git 2.0 from 'matching' to 'simple'. To squelch this message

and maintain the current behavior after the default changes, use:

gitconfig --global push.default matching

To squelch this message and adopt the new behavior now, use:

gitconfig --global push.default simple“ Pour étouffer ce message et maintenir le comportement actuel après le changement par défaut, utilisez **: git config --global push.default matching**. Pour étouffer ce message et adopter le nouveau comportement maintenant, utilisez : **git config --global push.default simple**.

La valeur "matching" de la variable de configuration push.default signifie que git push poussera toutes les branches locales vers les branches ayant le même nom sur le dépôt distant. Cela facilite la perte accidentelle de branches que vous n'aviez pas l'intention de pousser.

La valeur "simple" de la variable de configuration push.default signifie que git push ne poussera que la branche courante sur la branche que git pull tirera ; il vérifie également que leurs noms correspondent. Il s'agit d'un comportement plus intuitif, c'est pourquoi la valeur par défaut doit être changée en cette valeur de configuration.

Tout d'abord, vérifiez si un dépôt distant est défini : **$ git remote**. Si ce n'est pas le cas, définissez le dépôt distant nommé "origin" **: $ git remote add origin http://github.com/myRepoAddress.git**.

Maintenant, poussez les modifications en utilisant la commande suivante : **$ git push -u origin master**.

Après cela, vous aurez un résumé de ce qui a été poussé. En fait, le dépôt distant vérifiera la référence au commit actuel et la comparera avec la sienne. S'il y a des différences entre elles, cela échouera.

3.4.4 Removing a file

Si vous ne voulez plus d'un fichier, il y a deux façons de le supprimer :

• Supprimer manuellement le fichier et committer les changements. Cela supprimera le fichier localement et sur le dépôt. Utilisez la commande suivante :

$ git commit -m 'supprimer ce fichier'

• Supprimer le fichier uniquement à travers Git **: $ git rm --cached NomDeMonFichier.txt**

3.4.5 Checking the status

La commande "git status" permet d'afficher l'état de l'arborescence de travail, c'est-à-dire les fichiers qui ont été modifiés et ceux qui doivent être envoyés. Si tout est à jour, le résultat affichera "nothing to commit, working directory clean". Si un fichier est ajouté, Git demandera de le suivre avec la commande "git add". Si un fichier est modifié et que "git status" est réexécuté, Git affichera les fichiers prêts à être committés et ceux qui ne sont pas encore prêts. En utilisant "git add", le fichier sera prêt à être committé et sera affiché sous "Changes to be committed" avec la commande "git status".

- Si tout est correct et à jour :

  $ git status

  # On branch master

  nothing to commit, working directory clean

- Si vous ajoutez un fichier, Git vous alerte afin que vous suiviez ce nouveau fichier en utilisant « **git add** »

  $ touch text5.txt

  $ git status

  # On branch master

  # Untracked files:

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

  #

  #   text5.txt

  nothing added to commit but untracked files present (use "git add" to

  track

1. Si vous modifier un fichier and retaper « git status », vous aurez :

   $ echo "I am changing this file" > MyFile2.txt

   $ git status

   # On branch master

   # Changes to be committed:

   # (use "gitreset HEAD<file>..." to unstage)

   #

   # new file: text5.txt

   #

   # Changes not staged for commit:

   # (use "git add <file>…" to update what will be committed)

   #

   # modified: MyFile2.txt

2. Après avoir utilisé la commande « git add file », « git status » affiche également tout les fichiers modifiés avec les états et vous verrez les fichiers prêts à être « commiter »

   $ git add MyFile2.txt

   $ git status

   # On branch master

   # Changes to be committed:

   # (use "git reset HEAD <file>..." to unstage)

   #

   # new file: text5.txt

   # modified: MyFile2.txt

3.4.6 Ignoring files

Git peut facilement ignorer certains fichiers ou dossiers de votre arborescence de travail. Par exemple, si vous travaillez sur un site web et qu'il y a un dossier "upload" que vous ne souhaitez pas pousser sur le référentiel pour éviter d'avoir des images de test dans votre référentiel.

Pour cela, créez un fichier ".gitignore" à la racine de votre arborescence de travail :

$ touch .gitignore

Ensuite, ajoutez cette ligne dans le fichier ; cela permettra de ne plus suivre le dossier "upload" et son contenu :

upload

Les fichiers ou dossiers que vous définissez dans ce fichier ne seront plus suivis par Git.

Vous pouvez ajouter facilement des expressions régulières, comme :

Si vous voulez ignorer tous les fichiers PHP, utilisez l'expression régulière suivante : "*.php"

Si vous voulez ignorer tous les fichiers ayant "p" ou "l" à la fin de leur nom, utilisez l'expression régulière suivante : "*.[pl]"

Si le fichier est déjà poussé sur le référentiel, il est suivi par Git. Pour le supprimer, vous devrez utiliser la commande "git rm" en tapant ceci :

$ git rm --cached MonNomDeFichier.txt

3.5 Summary

Dans ce chapitre, nous avons vu les bases de Git : comment créer un référentiel Git, comment y mettre du contenu et comment pousser des données vers un référentiel distant. Dans le prochain chapitre, nous verrons comment utiliser Git avec une équipe et gérer toutes les interactions avec un référentiel distant.

4. Working in a Team Using Git

Dans ce chapitre, nous apprenons comment créer un dépôt Git serveur qui stockera et gérera le code. Il existe quatre protocoles que Git peut utiliser pour transporter les données : Local, Secure Shell (SSH), Git et HTTP. Nous verrons comment et quand utiliser ces protocoles et distinguerons les avantages et les inconvénients de chacun d'eux. Pour tous les protocoles, il faut créer un dépôt "bare repository" en exécutant des lignes de commande sur le serveur. Le protocole local est le plus simple à utiliser, car le dépôt distant est un répertoire local accessible par tous les membres de l'équipe. Le protocole SSH est le plus couramment utilisé lorsque le dépôt distant est sur un serveur distant.

4.1 Creating a server repository

4.1.1 LOCAL

Le protocole local est le protocole de base de Git. Dans ce protocole, le dépôt distant est un répertoire local. Ce protocole est utilisé si tous les membres ont accès au dépôt distant (par exemple, via NFS). Chaque programmeur doit ensuite cloner le dépôt distant en local. Par exemple, Jim, l'un des programmeurs, a déjà écrit des lignes de code et doit initialiser un dépôt local Git dans son répertoire et définir un emplacement distant pour le dépôt distant. Les avantages de ce protocole sont qu'il est facile à partager avec d'autres membres et qu'il permet un accès rapide au dépôt. Cependant, les inconvénients sont qu'il est difficile à configurer sur un réseau partagé et qu'il n'est rapide que si l'accès aux fichiers est rapide.

Jim@local:~/webproject$ git init

Jim@local:~/webproject$ git remote add origin /opt/git/webproject.git

4.1.2 SSH

Le protocole Secure Shell (SSH) est le plus utilisé, en particulier si le référentiel distant se trouve sur un serveur distant. Les programmeurs doivent d'abord le cloner localement en utilisant la commande :

git clone ssh: // username @ server / webproject.git .

Les programmeurs doivent installer leurs clés SSH sur le référentiel distant pour pouvoir pousser et tirer, sinon ils doivent spécifier le mot de passe pour chaque commande à distance. SSH compresse les données pendant le transport, ce qui le rend rapide. Les avantages de ce protocole sont qu'il est facile à partager en utilisant un serveur distant et qu'il n'y a pas d'accès anonyme, mais il a l'inconvénient de nécessiter une configuration plus complexe pour la mise en place d'un réseau partagé.

4.1.3 Git

Le protocole Git est similaire à SSH, mais sans aucune sécurité. Par défaut, il n'est pas possible d'y pousser des données, mais cette fonctionnalité peut être activée. Cependant, cela n'est pas du tout recommandé, car n'importe qui ayant l'adresse du dépôt pourrait y pousser des données. Comme pour les autres protocoles, le programmeur doit d'abord le cloner en local, comme suit **:**

Erik@local:~$ git clone git://username@server/webproject.git .

Les avantages et les inconvénients du protocole Git sont : Avantages : plus rapide que les autres. Inconvénients : pas de sécurité (le protocole Git est le même que SSH, sans la couche de sécurité).

4.1.4 HTTPS

Le protocole HTTPS est le plus facile à configurer et permet à toute personne ayant accès au serveur web de le cloner. Les programmeurs peuvent commencer à le cloner en local en utilisant la commande :

git clone https://server/webproject.git .

Dans le cas de Jim, il doit d'abord initialiser un dépôt Git local dans son dossier et ensuite définir une adresse distante pour le dépôt bare. Les avantages de ce protocole sont sa facilité de configuration, mais le transport de données est très lent.

4.2 Pushing data on remote repository – Jim case's

Jim a créé un nouveau référentiel Git local et ajouté un référentiel distant en utilisant le protocole SSH. Ensuite, il a ajouté et validé ses modifications en utilisant les commandes

git add . et **git commit -m 'add my code'**.

Enfin, il a poussé ses modifications vers le référentiel distant en utilisant la commande

git push -u origin master.

Maintenant, le référentiel distant contient deux fichiers (index.html et readme.txt).

Jim@local:~/webproject$ git add .

Jim@local:~/webproject$ git commit -m 'add my code'

[master (commit racine) 83fcc8a] add my code

2 files changed, 0 insertions(+), 0 deletions(-)

create mode 100644 index.html

create mode 100644 readme.txt

Jim@local:~/webproject$ git push –u origin master

Counting objects: 3, done.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 225 bytes | 0 byte/s, done.

Total 3 (delta 0), reused 0 (delta 0)

4.3 Pulling data from the repository

Les autres programmeurs doivent récupérer les nouveaux fichiers en exécutant la commande git pull pour maintenir et assurer que le code est à jour. Cette commande va vérifier et comparer le hachage de commit local avec le hachage distant. Si le distant est à jour, la commande essaiera de fusionner les données avec la branche master locale. Git pull est équivalent à exécuter les commandes git fetch (obtenir les données distantes) et git merge (fusionner avec votre branche). Le nom de l'un de nos dépôts distants est **origin**, et **master** est la branche locale actuelle.

$ git pull origin master

4.4 Creating a patch

Dans certains cas, vous pouvez vouloir envoyer des modifications à quelqu'un sans lui donner accès au dépôt. Pour ce faire, vous pouvez créer un patch et l'envoyer par courriel.

Pour créer un patch, utilisez la commande git format-patch. Par exemple, si vous voulez créer un patch pour tous les commits qui n'ont pas encore été envoyés à l'origine, vous pouvez utiliser cette commande :

git format-patch origin patch-webproject.patch

Cela créera un ou plusieurs fichiers « .patch ». Vous pouvez ensuite envoyer le fichier « .patch » par e-mail à la personne appropriée.

Pour appliquer un patch, utilisez la commande git apply. Par exemple, si vous avez reçu un patch appelé patch-webproject.patch, vous pouvez l'appliquer en utilisant la commande suivante :

Jim@local:~/webproject$ git apply /tmp/patch-webproject.patch

Cependant, cela ne créera pas de commit. Si vous voulez créer des commits à partir du patch, utilisez la commande git am. Par exemple :

Jim@local:~/webproject$ git am /tmp/patch-webproject.patch

Cela appliquera le patch et créera des commits en conséquence.

4.5 Working with branches

L'outil Git permet de créer des branches, qui sont des pointeurs nommés vers des commits. Les branches permettent de travailler indépendamment les unes des autres, et la branche par défaut est

généralement appelée "master". La création d'une branche dans Git est peu coûteuse en termes de ressources, grâce à l'utilisation d'un pointeur de 41 octets. Pour travailler sur une branche spécifique, il faut d'abord la "**checkout**" avec la commande :

$git checkout mabranche

Cela permet à Git de restructurer l'arborescence de travail avec le contenu du commit associer à la branche et de déplacer le pointeur HEAD sur la nouvelle branche.

$git branch

Permet de lister toutes les branches locales disponibles dans le référentiel, et la commande :

$git branch -a

Permet de lister toutes les branches, y compris les branches distantes.

Pour créer une branche à partir de la branche ou vous vous trouvez, utilisez la commande

$git branch mabranche

4.5.1 Playing with branches

Il existe plusieurs commandes utiles pour certaines fonctionnalités des branches. Tout d'abord, vous pouvez facilement renommer une branche en exécutant ceci :

$ git branch -m old_name new_name.

Pour changer la branche actuelle, vous pouvez contourner la variable old_name :

$ git branch -m new_name.

Ensuite, vous pouvez supprimer une branche :

$ git branch -d test.

Vous pourriez obtenir une erreur s'il y a des modifications non validées. Vous pouvez le forcer :

$ git branch -D test.

Enfin, vous pouvez pousser les modifications d'une branche vers un dépôt distant. En exécutant la commande push, vous pouvez spécifier la branche distante à utiliser :

$ gitpush origin test.

Après avoir effectué une vérification sur master, vérifiez le contenu du fichier readme.txt. Vous verrez que le contenu est l'ancien. Pour voir la différence entre deux branches, vous pouvez exécuter ceci :

$ git diff master test.

4.6 Tracking branches

Avec Git, une branche peut suivre une autre branche. Cela permet d'utiliser les commandes pull et push sans spécifier la branche et le dépôt. Par exemple, si vous clonez un dépôt Git, votre branche principale locale est créée en tant que branche de suivi de la branche principale du dépôt d'origine. Pour configurer une branche de suivi, exécutez les commandes :

//Créer une branch local dev a partir de main et push origin

//on branch main :

$git branch dev

$git checkout dev

$git push origin dev

//Retourner sur branch main

Pour tracker dev local avec dev origin

//on branch main

$git checkout –track origin/dev

Erreur?

($git checkout -b new_branch origin/branch_to_track)

ou

$git branch new_branch origin/master.

De même, vous pouvez spécifier de ne pas suivre une branche distante avec la commande

$git branch --no-track new_branch origin/master

Plus tard, il est possible de mettre à jour cette branche et tracker origin/master :

$ git branch -u origin/master new_branch

4.6.1 Deleting a branch from the remote

Enfin, si vous voulez supprimer une branche dans le dépôt distant, utilisez la commande

git push origin -d <branch>.

4.7 Merging

Le processus de fusion de branches dans Git permet de combiner les modifications de deux branches différentes. Pour ce faire, on utilise la commande "**git merge**" qui permet de fusionner les modifications d'une branche dans la branche courante. Par exemple, si on veut fusionner les changements de la branche "test" dans la branche courante "master", on exécute les commandes suivantes :

$git checkout master

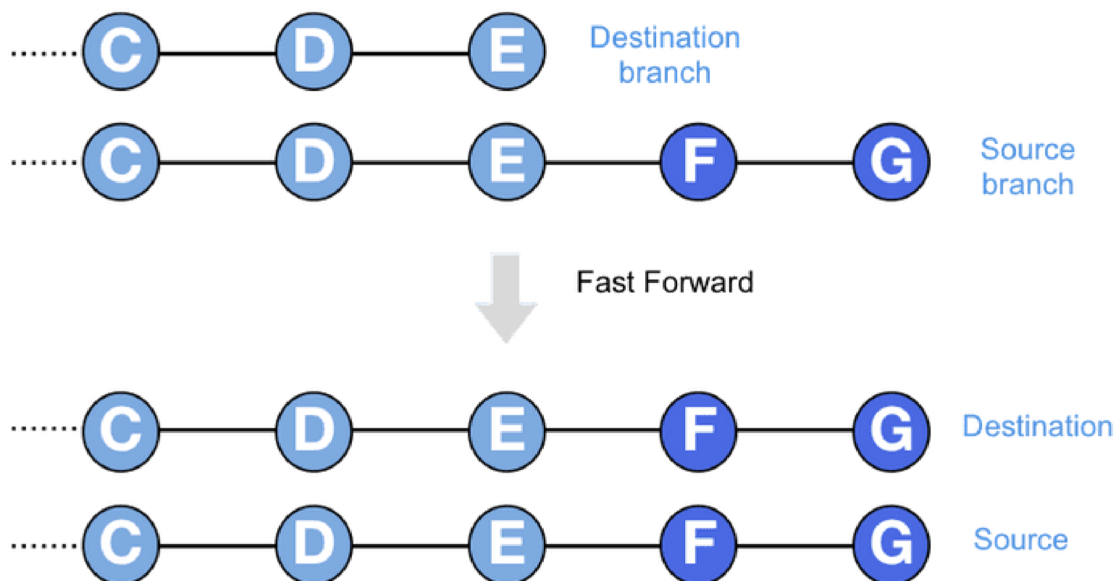
Puis

$git merge test.

4.7.1 Fast forward merge

Lorsque les commits fusionnés sont des prédécesseurs directs du pointeur HEAD de la branche actuelle, Git simplifie les choses en effectuant une fusion dite "fast forward". Cette fusion fast forward

déplace simplement le pointeur HEAD de la branche actuelle sur le dernier commit de la branche fusionnée. Cette méthode est représentée dans le diagramme ci-dessous montrant le processus de fusion entre deux branches. Si la branche fusionnée contient des commits qui sont les prédécesseurs directs de la branche actuelle, Git effectuera une fusion fast forward. Le schéma explique comment git effectue cette fusion avec la commande "git merge" et comment cela affectera la branche master.



Après un merge fast-forward, la branche master va pointer sur le dernier commit de la branche « source ».

4.7.2 Merge commit without fast-forward.

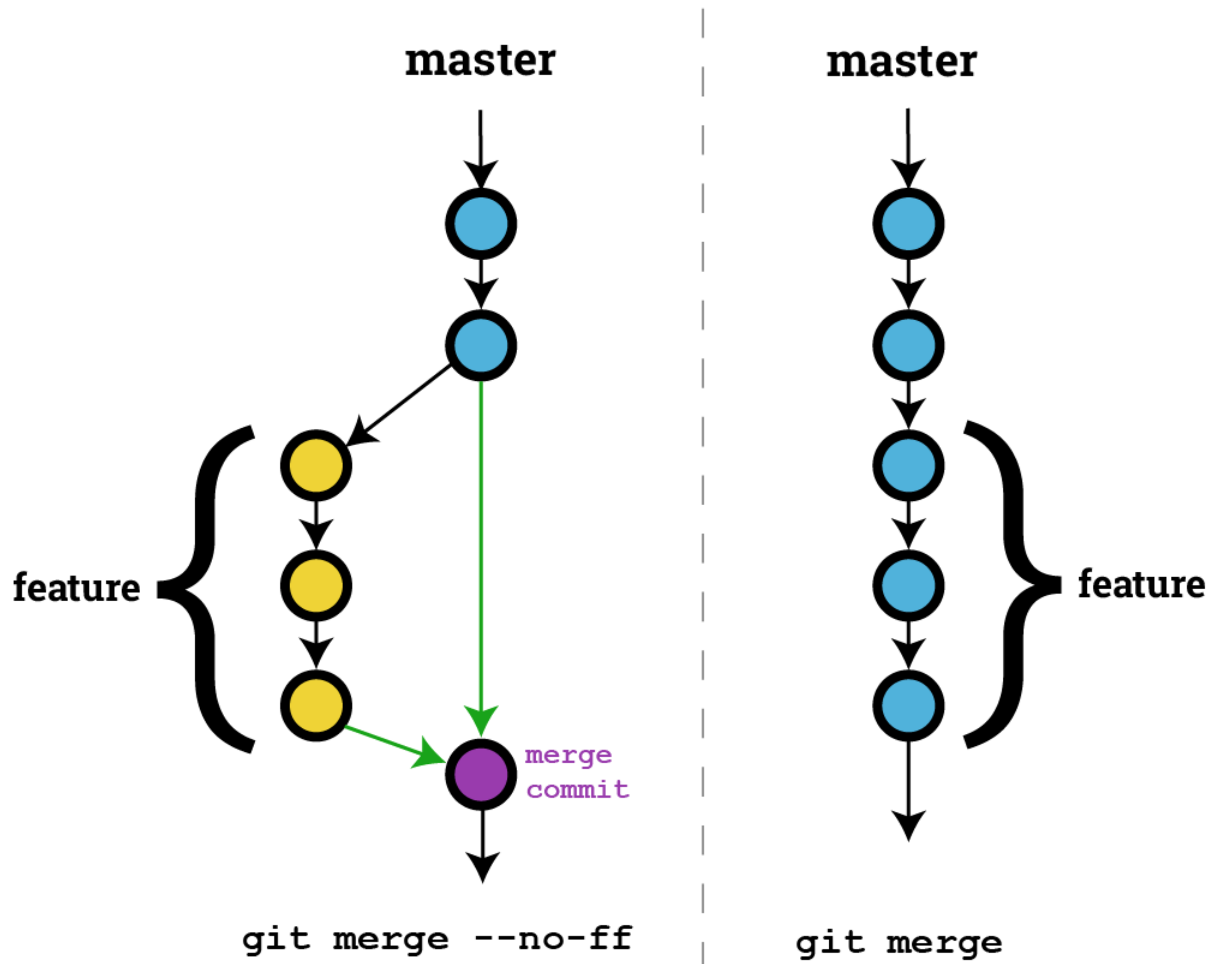
La stratégie de merge commit sans fast forward est une alternative à la fusion rapide (fast forward merge) lorsque la fusion des branches n'est pas une simple avancée (fast forward). Dans ce cas, Git crée un nouveau commit de fusion qui combine les modifications de chaque branche. Ce commit de fusion a deux parents, les deux branches fusionnées, ce qui permet de conserver l'historique de chaque branche et d'avoir une trace claire de la fusion. Cette stratégie est utile lorsque plusieurs personnes travaillent sur la même branche et que les modifications doivent être fusionnées en un seul commit pour éviter les conflits.

Pour réaliser correctement la stratégie "Merge Commit" sans fast forward, il faut exécuter la commande git merge avec l'option --no-ff qui désactive la fusion en avance rapide. Par exemple, si vous voulez fusionner la branche "feature" dans la branche "master" en utilisant cette stratégie, vous pouvez exécuter les commandes suivantes :

$git checkout master

$git merge –no-ff feature

Cela créera un commit de fusion qui incorporera tous les changements de la branche "feature" dans la branche "master". Ce commit de fusion aura deux parents, le dernier commit de la branche "master" et le dernier commit de la branche "feature"



4.7.3 Other merging strategies

La commande "git merge" permet de fusionner les modifications de deux branches. Il existe plusieurs stratégies de fusion disponibles en utilisant l'option -s, qui permet de spécifier la stratégie de fusion à utiliser.

La stratégie "**ours**" (la nôtre) permet d'ignorer complètement les modifications de la branche fusionnée et de conserver les modifications actuelles de la branche courante. Par exemple, si vous avez modifié le fichier index.html et que Jim a également apporté des modifications inutiles à ce fichier, vous pouvez utiliser la stratégie "ours" pour conserver uniquement vos modifications.

$ git merge -s ours test.

La stratégie "**theirs**" (la leur) fait exactement l'inverse de la stratégie "ours". Elle supprime toutes les modifications de la branche courante et conserve uniquement les modifications de la branche fusionnée. Par exemple, si Jim a apporté les mêmes modifications que vous mais de manière plus efficace, vous pouvez utiliser la stratégie "theirs" pour supprimer vos modifications et conserver celles de Jim.

$ git merge -s theirs test.

La stratégie "**recursive**" permet de spécifier l'option -X pour préférer les modifications locales ou distantes en cas de conflits entre les deux branches fusionnées. Cette option est utile pour fusionner des branches qui ont des modifications similaires mais qui ont été modifiées de manière différente.

$ git merge -s recursive -x ours test.

$ git merge -s recursive -x theirs test.

Il est important de faire attention à ne pas mélanger les stratégies "ours" ou "theirs" avec la stratégie "recursive", car cela peut avoir des conséquences différentes sur le résultat de la fusion.

4.8 Rebase

La commande "rebase" est utilisée pour réécrire l'historique de commits en déplaçant une branche sur une autre branche.

Lorsque vous exécutez la commande "git rebase", Git prend la branche actuelle (généralement la branche où vous voulez appliquer les changements) et la rebase sur une autre branche (généralement la branche master). Git va alors récupérer tous les commits qui ont été effectués sur la branche actuelle depuis son point de divergence avec la branche master, et les appliquer un par un sur la branche master.

Cela signifie que les commits sur la branche actuelle auront maintenant des parents différents (les commits de la branche master). Git va donc recréer une nouvelle séquence de commits qui ressemblera à celle de la branche actuelle, mais qui sera appliquée sur la branche master.

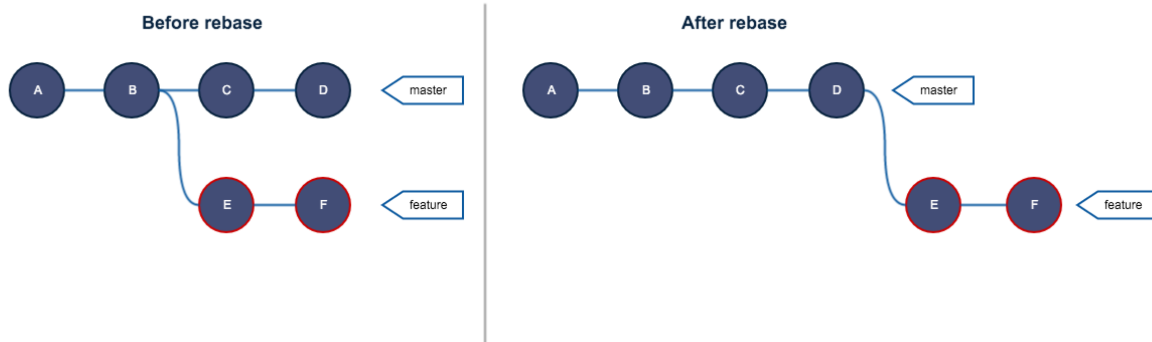
Le processus de rebase peut entraîner des conflits lors de la l'application des commits de la branche actuelle sur la branche master. Dans ce cas, Git va s'arrêter et vous devrez résoudre les conflits manuellement. Vous pouvez également annuler le rebase à tout moment en utilisant la commande **"git rebase --abort".**

Il est important de noter que la commande "**rebase**" est différente de la commande "**merge**". Alors que la commande "**merge**" combine deux branches en créant un nouveau commit de fusion, la commande "**rebase**" récrit l'historique de la branche. Par conséquent, il est généralement recommandé de n'utiliser la commande "**rebase**" que sur des branches locales et jamais sur des branches distantes partagées avec d'autres développeurs, afin d'éviter des conflits lors de la synchronisation des branches.

Example : Si nous voulons rebase une branche à partir de « master », nous devons aller avec checkout sur la branche à rebase et rebase à partir de master.

$ git checkout branch

$ git rebase master



L'option "--interactive" permet d'interagir après chaque commit pour changer les messages, ajouter des fichiers ou effectuer toute autre action. Cette option offre plusieurs options, notamment "**pick**" pour inclure un commit, "**reword**" pour modifier son message, "**edit**" pour amender un commit, "**squash**" pour combiner plusieurs commits en un seul, "**fixup**" pour combiner des commits sans changer le message et "**exec**" pour exécuter des commandes shell sur un commit.

La commande **"git rebase --interactive**" vous permet de réécrire l'historique des commits d'une branche en modifiant, réorganisant ou supprimant des commits. Elle permet également de fusionner plusieurs commits en un seul ou de changer le message de commit.

Voici comment utiliser la commande "git rebase --interactive" :

1. Commencez par vous assurer que vous êtes sur la branche que vous voulez réécrire. Par exemple, si vous voulez réécrire l'historique des commits de la branche "feature-branch", vous pouvez taper la commande suivante : "**git checkout feature-branch**".

2. Lancez la commande **"git rebase --interactive**" suivie de l'identifiant du commit parent de la branche que vous voulez réécrire. Par exemple, si vous voulez réécrire les commits de la branche "feature-branch" depuis le commit parent "master", vous pouvez taper la commande suivante : "**git rebase --interactive master**".

3. Git ouvrira alors un éditeur de texte avec une liste des commits que vous allez réécrire. Chaque commit sera précédé d'un code d'opération (par exemple, "pick", "reword", "squash", etc.).

4. Pour modifier un commit, changez le code d'opération correspondant. Par exemple, si vous voulez fusionner deux commits en un seul, changez le code d'opération du deuxième commit en "squash".

5. Enregistrez vos modifications et quittez l'éditeur de texte.

6. Git appliquera alors les modifications selon vos instructions et vous permettra de modifier les messages de commit ou de fusionner plusieurs commits en un seul.

7. Une fois que vous avez terminé de modifier les commits, utilisez la commande "**git log**" pour vérifier que l'historique a été réécrit correctement.

8. Enfin, utilisez la commande "**git push --force**" pour pousser les modifications sur le serveur. Attention, cette commande force la mise à jour de la branche distante, ce qui peut affecter les autres utilisateurs travaillant sur la même branche.

4.9 Cherry-pick

La commande git cherry-pick vous permet de sélectionner un commit d'une branche pour l'appliquer à une autre. La modification sera considérée comme un nouveau commit dans la branche sélectionnée. **Par exemple**, si Jim crée une branche "jim" à partir de la branche "master", ajoute un nouveau fichier et fait deux commits dans cette branche, il peut appliquer le premier commit à la branche "master" en utilisant la commande "**git cherry-pick**".

$git checkout -b jim

$ touch home.html

$ git add home.html.

$ git commit -m 'add homepage'.

$ echo "<html>…</html>" > home.html

$ git commit -a -m 'add content inside home'.

Nous constatons que jim à bien créer un nouveau fichier, il l'a ensuite ajouter à Git et l'a commit. Ensuite il a éditer le fichier et à refait un commit. Maintenant nous pouvons voir l'historique des commit avec :

$ git log --oneline

4f6ec45 add content inside home

22c45b7 add homepage

Pour finir Jim va appliquer le premier commit à la branche « master » :
$ git checkout master

$ git cherry-pick 22c45b7

Si quelque chose ne va pas lors de l'application du cherry-pick, il peut être annulé en utilisant la commande suivante :

$git cherry-pick --abort .

Si nous voulons restaurer (« Rollback ») un « **Cherry-pick** », nous avons 2 moyens de le faire :

Si le cherry-pick a déjà été poussé sur une branche publique, il est préférable d'utiliser la commande suivante pour annuler les changements :

$git revert

Si le cherry-pick a été poussé sur une branche privée, on peut utiliser la commande :

$git rebase

4.10 (Using tags)

Les tags sont utilisés pour marquer des commits dans le dépôt Git. Le plus souvent, les tags sont utilisés pour marquer une version d'application sur un commit, afin de faciliter leur gestion. La commande pour créer un tag est simple :

$git tag <nom_de_tag>.

Vous pouvez également ajouter une description avec l'option `**-m**`.

$ git tag 1.0.0

#Annotated tag contains a small description

$ git tag 1.0.0 -m 'Release 1.0.0'

#Use a commit

$ git tag 1.0.0 -m 'Release 1.0.0' commit_hash

Les tags peuvent être supprimés mais par défaut seulement à l'intérieur de dépôt local. Pour le « push », il faut :

1. Lister les tags disponibles

$ git tag

0.1.0

0.1.5

0.2.0

0.9.0

1.0.0

2. Supprimer le dernier tag localement :

$ git tag -d 1.0.0

3. **«** push remotely » sur le serveur distant

$ git push origin tag 1.0.0

Les tags de version suivent une convention de nommage de la forme «major_version.minor_version.patch_version» par exemple, 1.0.0. , où la version de correctif est incrémentée pour les corrections de bogues compatibles, la version mineure est incrémentée pour les corrections de bogues incompatibles et la version majeure est incrémentée pour les bogues rétro-incompatibles.

4.11 Summary

Dans ce chapitre, nous avons vu comment travailler en équipe en utilisant Git, ce qui est très courant pour la plupart des développeurs. Maintenant, vous comprenez ce qu'est une branche, comment nous pouvons les fusionner et comment rebaser une branche sur une autre. Nous avons également vu

comment étiqueter un commit. Maintenant, vous êtes prêt à préparer et travailler sur votre référentiel Git, mais il y a quelque chose qui a été laissé de côté : que devez-vous faire s'il y a des conflits ? Le prochain chapitre est consacré à cette question. Nous verrons comment trouver quelque chose dans votre référentiel, explorer le référentiel et surtout, comment résoudre les conflits et les erreurs.

5. Finding and Resolving Conflicts

Ce chapitre couvre une partie de Git que vous rencontrerez certainement : les conflits. Comment pouvons-nous les résoudre ? En travaillant ensemble en tant qu'équipe sur un projet, vous travaillerez sur les mêmes fichiers. La commande pull ne fonctionnera pas car il y a des conflits et vous avez peut-être essayé des commandes Git qui ont entraîné des problèmes. Dans ce chapitre, nous trouverons des solutions à ces conflits et verrons comment les résoudre. Nous aborderons les sujets suivants :

- Trouver du contenu dans votre référentiel Git
- Mettre en cache vos modifications
- Correction d'erreurs par des exemples pratiques

5.1 Finding content inside your repository

5.1.1 Searching file content

Ce chapitre traite de la recherche de contenu à l'intérieur de votre dépôt Git. Vous pouvez rechercher du texte à l'intérieur de vos fichiers en utilisant la commande "git grep". Il affichera tous les résultats correspondant au mot-clé donné dans votre code avec le modèle :

"[commitref]: [chemin du fichier]: [numéro de ligne]: [contenu correspondant]".

Exemple :

$ git grep "Something to find"

$git grep -n body

Master:Website.Index.html:4: <body>

Master:Website.Index.html:12: </body>

Nous pouvons également specifier la référence du commit dans lequel « grep » va chercher le mot-clé :

$ git grep -n body d32lf56 p88e03d HEAD~3

Master:Website.Index.html:4: <body>

Master:Website.Index.html:12: </body>

Git permet d'utiliser des expressions régulières à l'intérieur de la fonction de recherche en remplaçant "quelque chose à trouver" par une expression régulière. Vous pouvez également utiliser les opérateurs logiques "ou" et "et" avec grep pour des recherches plus avancées.

$ git grep -e myRegex1 --or -e myRegex2

$ git grep -e myRegex1 --and -e myRegex2

Example pratique :

$ git grep -e [A-Z] --and -e [0-9] HEAD

Master:Website.Test.html:6: TEST01

5.1.2 Exploring the repository history

Le meilleur moyen d'explorer les commits précédents dans votre dépôt est d'utiliser la commande git log. Pour cette partie, nous supposons qu'il n'y a que deux commits. Pour afficher tous les commits, utilisez la commande suivante :

$git log --all.

Cependant, cette commande peut afficher beaucoup d'informations, il est donc important de savoir comment filtrer les résultats. Voici quelques exemples de filtres :

- $**git log -1** : affiche le dernier commit.

- $**git log --author=Erik -1** : affiche le dernier commit d'Erik.

- $git log --author=Erik --before "2014-07-20" --after "2014-07-18" : affiche tous les commits d'Erik entre les deux dates.

- $**git log --author=Jim --stat** : affiche un résumé des modifications apportées à chaque commit par Jim.

La commande git log a de nombreux paramètres pour filtrer les résultats, qui peuvent être consultés en utilisant la commande

$git help log.

Il est également possible d'utiliser le paramètre -p pour afficher les modifications complètes de chaque commit. Enfin, il est possible de restreindre la recherche à un fichier spécifique en utilisant le paramètre [file].

$git log [file]

5.1.3 Viewing changes

Il y a deux façons de voir les changements dans un référentiel Git :

$git diff

et

$git show.

La commande **$git diff** permet de voir les changements qui **ne sont pas encore** validés. Par exemple, si nous avons un fichier index.php et que nous remplaçons le contenu du fichier par une ligne, nous verrons un signe plus (+) ou moins (-) juste avant la ligne. Le **signe +** signifie que le contenu a été ajouté et **le signe -** indique qu'il a été supprimé.

$ git diff

```
diff --git a/index.php b/index.php
indexb4d22ea..748ebb2 100644
--- a/index.php
+++ b/index.php
@@ -1,11 +1 @@
-<html>
-
-<head>
-<title>Git is great!</title>
-</head>
-<body>
-<?php
- echo 'Git is great';
-?>
-</body>
-</html>
+<b> I added a line</b>
```

Pour analyser un commit, il est conseillé d'utiliser la commande

$git show commitId,

qui affichera la liste complète des modifications apportées par le commit.

Si vous voulez afficher les commits pour un fichier spécifique, vous pouvez utiliser la commande

$ git blame index.php

e4bac680 (Erik 2014-07-20 19:00:47 +0200 1) I added a line

Cette commande vous permet de voir l'historique d'un fichier et de savoir qui a modifié chaque ligne.

5.2 Stashing your changes

Lorsque vous travaillez sur un projet git, vous pouvez effectuer des modifications locales qui ne sont pas encore prêtes à être validées et qui ne doivent pas être perdues. Pour sauvegarder l'état actuel de

votre dépôt de travail local et revenir à la dernière révision validée, Git dispose d'une commande appelée stash.

Cette commande est vraiment utile lorsque vous devez effectuer une correction urgente et que vous devez interrompre votre travail actuel pour la réaliser. Une fois cela fait, vous pouvez restaurer les modifications sauvegardées et continuer avec votre travail.

Pour utiliser cette commande, exécutez simplement la commande suivante :

$ git stash

#Do your fix and then unstash edit

$ git stash pop

Nous pouvons faire plus , comme par exemple :

#See the list of available stashes

$ git stash list

#Apply the second stash

$ git stash apply stash@"1}

#Delete a stash

$ git stash drop stash@"1}

#Or delete all stashes

$ git stash clear

5.3 Cleaning your mistake

Cette section explique comment corriger les erreurs avec Git. Il est toujours possible de corriger une erreur avec Git. Voici comment supprimer les fichiers non suivis avec Git. Pour cela, il faut utiliser la commande suivante pour effectuer un test à sec avant de supprimer les fichiers non suivis. :

$git clean -n.

Si vous voulez supprimer également les répertoires et les fichiers cachés, il faut utiliser la commande suivante avec les options -d pour supprimer les nouveaux répertoires, -x pour supprimer les fichiers cachés et -f pour forcer la suppression :

$ git clean -fdx

5.3.1 Reverting uncommitted changes

Supprimez des modifications non enregistrées sur un fichier en utilisant la commande git checkout. L'exemple donné suppose que vous avez modifié un fichier sur le répertoire de travail de production sans le « commiter ». Sur votre dernier push, vous l'avez modifié et les modifications en production ne

sont plus nécessaires. Votre objectif est donc d'effacer les modifications sur ce fichier et de le réinitialiser à la dernière version validée(commit).

La commande à utiliser est :

$git checkout filename

Cette commande permet également de restaurer un fichier supprimé.

Il est possible de spécifier un pointeur de commit à utiliser (utile si vous avez caché vos modifications) avec la commande :

$git checkout HEAD myfile.txt.

5.3.2 The git reset command

Cette section explique comment retrouver et résoudre les conflits de code en utilisant la commande "git reset". Elle donne également des explications détaillées sur les options que vous pouvez utiliser avec la commande "git reset".

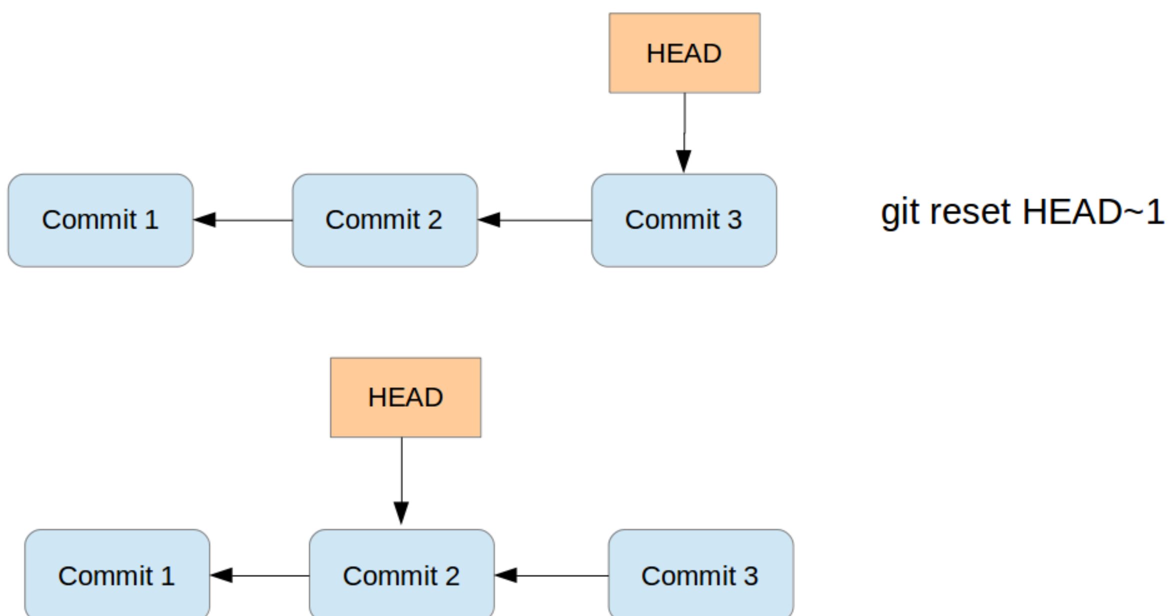
La commande "git reset" permet de revenir en arrière dans l'historique des commits et de restaurer les fichiers locaux à une version précédente. Il existe différentes options que vous pouvez spécifier avec la commande "git reset" pour définir le comportement de la restauration.

Par exemple, si vous souhaitez revenir à la dernière version commitée d'une branche, vous pouvez utiliser la commande suivante :

$ git reset commitID

Cela ramènera les fichiers locaux à l'état du dernier commit.

Le diagramme suivant illustre le comportement de la commande "git reset" :



Cette commande utilisera l'option par défaut "hard", qui restaurera le contenu à la version commitée et effacera toutes les modifications locales.

Il existe cependant trois options différentes que vous pouvez utiliser avec la commande « **git reset** » :

- --**hard**: Cette option est la plus simple. Elle restaurera le contenu à la version commitée donnée. Toutes les modifications locales seront effacées. La commande "git reset --hard" signifie "git reset --hard HEAD", qui réinitialisera vos fichiers à la version précédente et effacera vos modifications locales.

- --**mixed**: Cette option réinitialise l'index, mais pas l'arborescence de travail. Elle réinitialisera vos fichiers locaux, mais les différences trouvées lors du processus seront marquées comme des modifications locales si vous les analysez en utilisant "git status". Cette option est très utile si vous avez fait des erreurs dans les commits précédents et que vous souhaitez conserver vos modifications locales.

- --**soft**: Cette option permet de conserver tous vos fichiers, comme "mixed". Si vous utilisez "git status", cela apparaîtra comme des modifications à committer. Vous pouvez utiliser cette option lorsque vous n'avez pas commité des fichiers comme prévu, mais que votre travail est correct. Vous devez simplement les commiter à nouveau comme vous le souhaitez.

Le tableau suivant explique exactement ce que les options changent :

OPTION	**Head pointer**	**Working tree**	**Staging area**
Soft	YES	NO	NO
Mixed	YES	NO	YES
Hard	YES	YES	YES

La commande "git reset" ne supprime pas les fichiers non suivis ; utilisez plutôt "git clean".

5.3.3 Editing a commit

Cette section explique plusieurs astuces pour éditer votre dernier commit.

Si vous voulez modifier la description de votre dernier commit, utilisez la commande suivante :

$ git commit --amend

Supposons maintenant que votre dernier commit contient un code avec un bug. Vous pouvez spécifier que vos modifications sur le fichier font partie du dernier commit en utilisant :

$ git add filename.txt

$ git commit -v --amend

Si vous souhaitez supprimer un fichier inclus accidentellement dans le dernier commit (parce que ce fichier mérite un nouveau commit), vous pouvez utiliser les commandes suivantes :

$ git reset HEAD^1 filename.txt

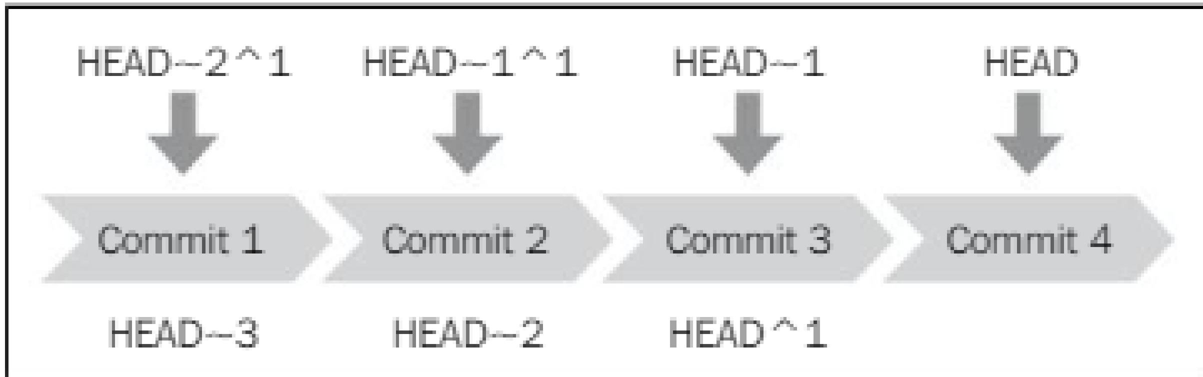
$ git commit --amend -v

$ git commit -v filename.txt

Il est possible d'utiliser les notations HEAD^ et HEAD~ pour spécifier les commits sans utiliser l'ID de commit.

HEAD~ ou HEAD^ signifie **HEAD~1** et est le premier parent du commit**. HEAD~2** désigne le grand-parent du commit.

Ils peuvent être utilisés ensemble, par exemple HEAD~3^2 signifie l'ancêtre de la deuxième génération du commit HEAD.



5.3.4 Canceling a commit

La commande **git revert** permet « d'annuler » votre dernier commit non-poussé. Cependant, Git ne supprime pas le commit original, mais crée un nouveau commit qui exécute l'opposé de votre commit. Il est important de noter qu'un commit poussé est irréversible, il est donc impossible de le changer.

Pour annuler le dernier commit, il faut d'abord examiner les derniers commits à l'aide de la commande git log. Ensuite, pour annuler un commit, il suffit d'utiliser la commande **git revert** suivi du numéro de commit ou des premiers caractères de celui-ci (au moins 6 caractères sont nécessaires pour être sûr qu'aucun autre commit commence par les mêmes caractères).

$ git log

commite4bac680c5818c70ced1205cfc46545d48ae687e

Author: Eric Pidoux

Date: Sun Jul 20 19:00:47 2014 +0200

replace all

commit0335a5f13b937e8367eff35d78c259cf2c4d10f7

Author: Eric Pidoux

Date: Sun Jul 20 18:23:06 2014 +0200

commitindex.php

We want to cancel the 0335… commit:

$ git revert 0335a5f13

5.3.5 Rewriting commit history

Il peut arriver qu'on souhaite supprimer un fichier de tous les commits car il contient des informations confidentielles. Pour cela, on peut utiliser la commande "**git filter-branch**". Voici un exemple de commande pour supprimer le fichier "myconfidentialfilename.txt" de tous les commits :

$ git filter-branch --index-filter 'git rm --cached --ignore-unmatch myconfidentialfilename.txt' HEAD

L'option "--ignore-unmatch" est utilisée ici car la commande "git rm" échouera si le fichier n'est pas présent dans l'arbre.

5.3.6 Solving merge conflicts

Dans certains cas, il peut y avoir un conflit lors de la fusion de deux branches, par exemple si Jim et Erik ont tous les deux modifié le fichier index.html sur deux branches différentes. Dans ce cas, Git va marquer le conflit et vous devrez le résoudre manuellement. Pour cela, vous devrez éditer le fichier pour trouver les différences entre les deux modifications. Les marques <<<<<<<, ======= et >>>>>>> indiquent les points de divergence. Après avoir résolu le conflit**, il faut supprimer** ces marques ajoutées par Git et commiter les changements.

Si le conflit est trop compliqué, Git fournit un outil utile pour vous aider, la commande "**git diff**" vous permettra de trouver les différences entre les modifications. Cette commande affichera les différences entre les deux versions du fichier, en marquant avec **un +** les modifications de la branche source et avec **un -** les modifications de la branche locale.

Exemple :

$ git dif

Diff --git erik/mergetestjim/mergetest

Index.html 88h3d45..92f62w 130634

--- erik/mergetest

+++ jim/mergetest

@@ -1,3 +1,4 @@

<body>

+I added this code between

This is the file content

-I added a third line of code

+And this is the last one

Les lignes avec "+" proviennent de origin/master et les lignes avec un « - proviennent du dépôt local. Les lignes sans signe sont similaires dans les 2 dépôts.

5.4 Fixing errors by practical examples

Ce chapitre présente les différentes erreurs courantes rencontrées dans Git et comment les résoudre. Voici quelques exemples d'erreurs et de solutions :

1. Remote origin already exists: This error occurs when you already have a remote repository specified and the remote origin removed and added:

   $ git remote rm origin

   $ git remote add origin https://github.com/sexyboys/InflexibleBundle.git

2. Git push fails with rejected error: This error occurs because you didn't execute git pull before git push:

   $ git pull

   $ git push

3. **"Git push fails with "fatal: The remote end hung up unexpectedly" :** Cette erreur est courante et vous devriez vérifier si votre URL distante est correcte et si Git a accès au dépôt distant.
4. **Restauration d'un fichier modifié à son état précédemment validé** : Exécutez git checkout suivi du nom de fichier et vous perdrez vos modifications. Cependant, le fichier sera restauré comme une copie propre :
   $git checkout nomDuFichier
5. **Unstaging a file :** Avez-vous exécuté git add trop tôt ? Exécutez git reset HEAD suivi du nom de fichier pour l'annuler :
   $ git reset HEAD nomDuFichier
6. **Comment corriger le message de commit le plus récent** ? L'option --amend éditera le message de commit :
   $ git commit --amend. L'éditeur s'ouvrira pour éditer le dernier message.
7. **Annuler le commit le plus récent** : Il existe deux façons de le faire : avec et sans vos modifications.
   Sans perdre les modifications :
   $ git reset HEAD~1
   En perdant les modifications :
   $ git reset --hard HEAD~1
8. **J'ai trouvé un bogue après la publication du produit, mais il était dans le commit que j'ai fait il y a longtemps, comment le corriger** ? Dans ce cas, vous ne devriez pas utiliser git reset car cela réécrit l'historique et le produit est déjà publié. Par conséquent, vous devriez faire un commit qui annule le commit contenant le bogue et le pousser pour le partager avec vos collègues :
   $ git revert commitID
9. **Il y a beaucoup de fichiers indésirables dans le répertoire de travail. Comment les supprimer ?** Dans cette situation, les fichiers ne sont pas gérés par Git, vous devez donc utiliser git clean :
   # Pour vérifier les fichiers qui seront supprimés :
   $ git clean -n
   # Pour les supprimer :
   $ git clean -f.¨

10. **Je pense avoir commis une erreur en résolvant des conflits de fichiers. Comment puis-je le restaurer à l'état juste après la fusion Git?** Pour cela, vous pouvez utiliser checkout :
    $git checkout --merge leNomDuFichier

11. **Le fichier d'index Git est corrompu.** Git affichera l'erreur comme une mauvaise signature sha1 du fichier d'index : fatal *: index file corrupt*. Vous devez simplement supprimer le fichier d'index de sauvegarde, puis réinitialiser le référentiel:
    $ mv .git/index .git/indexOLD

    Erik@server:~$ git reset

Git refuse de démarrer une commande de merge/pull. Les messages d'erreur typiques ressemblent à ceci :

- Error : L'entrée index.html n'est pas à jour; elle ne peut pas être « merge ».
- Error : L'entrée index.html sera écrasée par la commande « merge ».

Pour résoudre cela, effectuez les étapes suivantes :

1. Mettre en réserve les modifications ou les jeter :
   $ git stash save "mon message"

   $ git checkout index.html

2. Vérifier que les modifications sont « staged » :

$ git status

2. Apporter les modifications du référentiel distant :
   $ git pull
3. Recharger le stash si vous avez fait une mise en réserve :
   $ git stash pop

5.5 Summary

Dans ce chapitre, nous avons appris comment trouver quelque chose dans votre dépôt Git et comment résoudre les erreurs. Maintenant, vous ne devriez pas craindre de commettre des erreurs car vous savez que vous pouvez les réparer. À la fin de ce chapitre, nous avons vu les erreurs les plus courantes qui se produisent. Pour le prochain chapitre, vous apprendrez à approfondir Git et à examiner toutes les possibilités offertes par le système de versionnage.

6. (Going Deeper into Git)

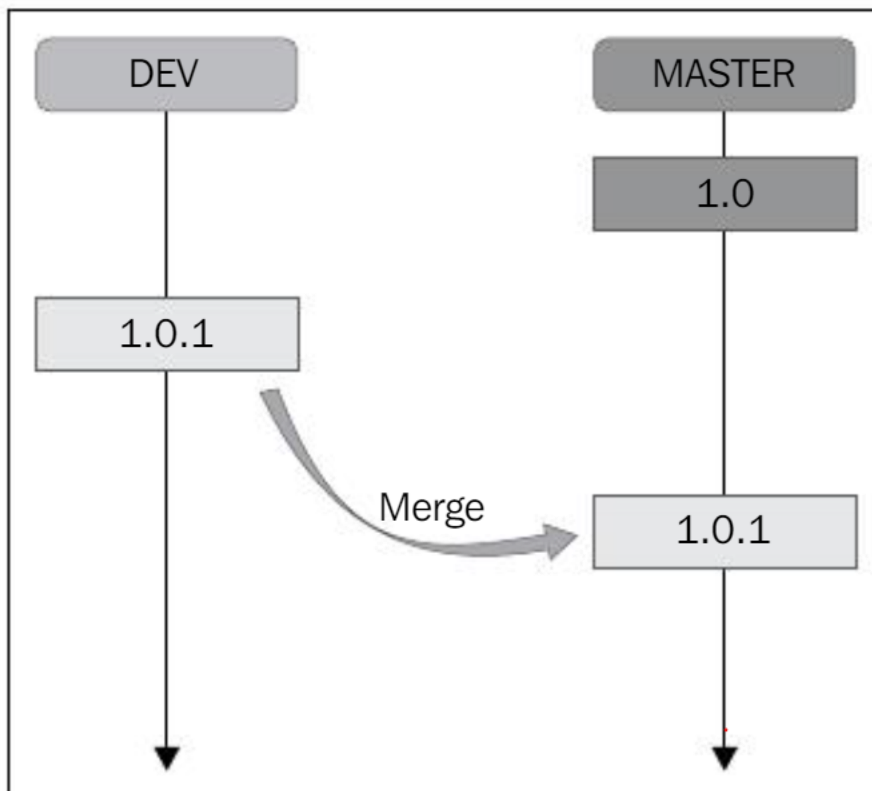
7. Using Git for Continuous Integration

Le dernier chapitre du livre a pour but de montrer comment utiliser Git de manière efficace dans son flux de travail. L'auteur abordera notamment la création de branches spécifiques pour améliorer son travail, l'utilisation de Git dans un environnement agile, le travail avec l'intégration continue et l'utilisation d'autres outils Git.

7.1 Creating an efficient branching system

7.1.1 Git flow

En 2010, un développeur iOS hollandais nommé Vincent Driessen a publié l'article intitulé "Git flow". Dans cet article, il présente sa stratégie de branche. Sa stratégie de branche commence par la création de deux branches principales : "Master" et "Dev". La branche Master est la branche principale du projet et sera dans l'état prêt pour la production. Elles sont sur le référentiel distant (origin). Donc, chaque fois que vous clonez le référentiel sur la branche Master, vous obtiendrez la dernière version stable du projet, ce qui est très important.



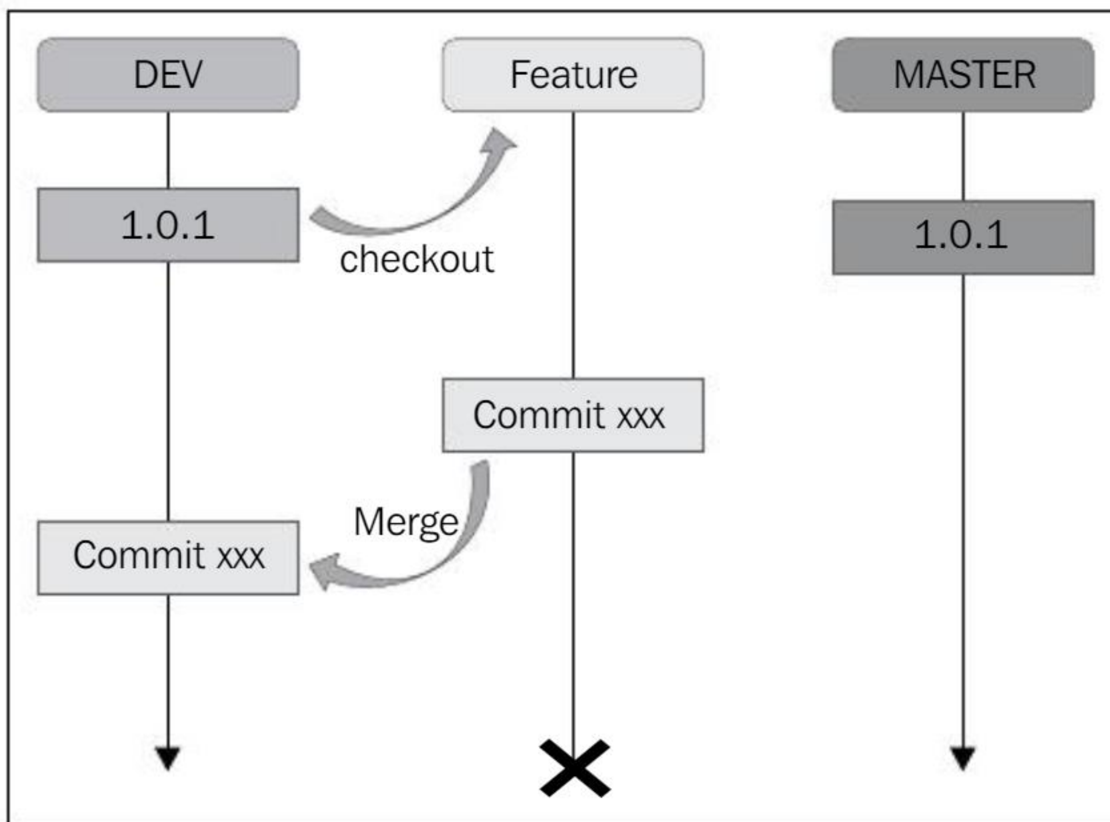
La branche Dev reflète toutes les nouvelles fonctionnalités pour la prochaine version. Lorsque le code dans la branche Dev est stable (ce qui signifie que vous avez effectué toutes les modifications pour les prochaines versions et les avez testées), vous atteignez le point stable sur la branche Dev. Ensuite, vous pouvez fusionner la branche Dev dans Master.

Vincent Driessen a également utilisé d'autres branches autour de ces deux branches principales, qui peuvent être classées en trois types différents :

• Les branches de fonctionnalités (feature branches)

Une branche *feature* est nommé en fonction de ce que votre fonctionnalité concerne et existera aussi longtemps que la fonctionnalité est en développement. Les branches *feature* n'existent que dans les dépôts de développeurs locaux ; ne les poussez pas sur le dépôt distant. Lorsque votre fonctionnalité est prête, vous pouvez fusionner votre branche **feature** avec **develop** et supprimer la branche. Voici les étapes à suivre :

1. Revenir à la branche dev :
   $ git checkout dev
2. Fusionner la branche à la branche dev en créant un nouvel objet de commit :
   $ git merge --no-ff featureBranch
3. Supprimer la branche (la branche explique une fonctionnalité qui fait maintenant partie de la branche dev, il n'y a donc aucune raison de la conserver) :
   $ git branch -d featureBranch
4. Pousser vos modifications sur le dépôt distant dev :
   $ git push origin dev



• Les branches de sortie (release branches)

Dans le processus de développement Git, la branche de sortie est utilisée pour mettre à jour le code pour des changements mineurs entre deux grandes versions. Cette branche est nommée en fonction du numéro de version du projet. Par exemple, si le site Web a été publié avec la version 1.0, et que le prochain grand développement prévoit d'inclure un blog, alors qu'un bug mineur se présente en

production, une branche de sortie sera créée à partir de la branche de développement, nommée "release-1.1". Après la correction du bogue, la branche de sortie doit être fusionnée avec la branche principale (Master), puis le projet est tagué avec la nouvelle version. Ensuite, pour que la branche de développement inclue également les modifications, il est nécessaire de la fusionner avec la branche de sortie. Enfin, la branche de sortie doit être supprimée.

$ git checkout -b release-1.1 dev

1. First, you have to merge this branch release into master:

   $ git checkout master

   $ git merge --no-ff release-1.1

2. Then, you can tag your project to the new release version:

   $ git tag -a 1.1

3. You will probably notice that your dev branch didn't include the changes!

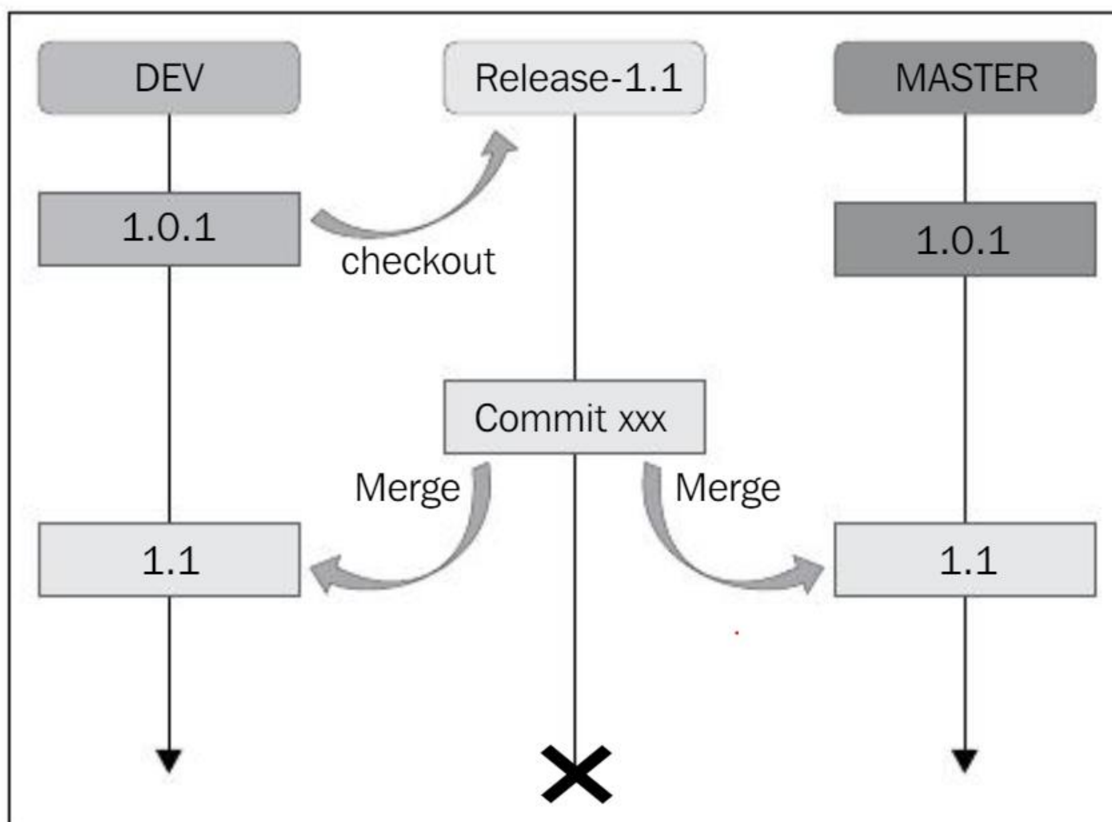
   To fix this, you have to merge it into the dev:

   $ git checkout dev

   $ git merge --no-ff release-1.1

4. When it's done, delete the release branch:

   $ git branch -d release-1.1



• Les branches de correction rapide (hotfix branches)

Les branches de type "hotfix" sont similaires aux branches de version. Elles sont utilisées pour corriger rapidement les bogues critiques en production, afin que les autres membres de l'équipe puissent continuer à travailler sur leurs fonctionnalités. Par exemple, si votre site Web est étiqueté comme étant en version 1.1 et que vous travaillez toujours sur la fonctionnalité de blog sur la branche "blog", vous pouvez trouver un bogue important sur le curseur à l'intérieur de la page principale. Dans ce cas, vous travaillez sur la branche de correction rapide pour le corriger dès que possible.

Exemple :

1. Créer une branche « hotfix » nommée :

   $ git checkout -b hotfix-1.1.1 master

2. Fixer le bogue et merger la branche dans la branche master (après un commit 😊)

   $ git checkout master

   $ git merge --no-ff hotfix-1.1.1

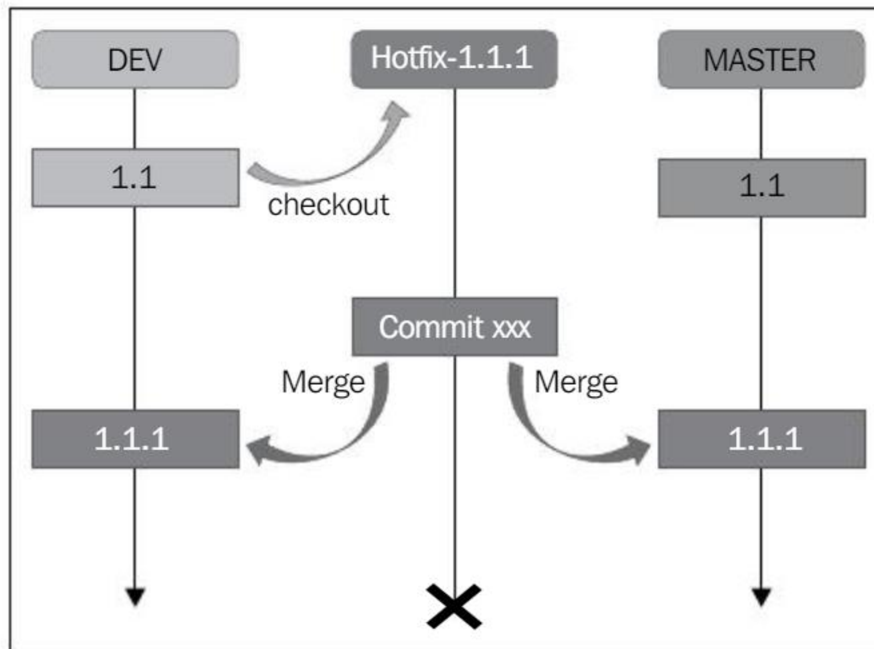
   $ git tag -a 1.1.1

3. Comme pour les branches de type « release », merger la branche « hotfix » dans la branche « release » courante ou dans la branche « dev » et ensuite supprimer la :

   $ git checkout dev

   $ git merge --no-ff hotfix-1.1.1

   $ git branch -d hotfix-1.1.1



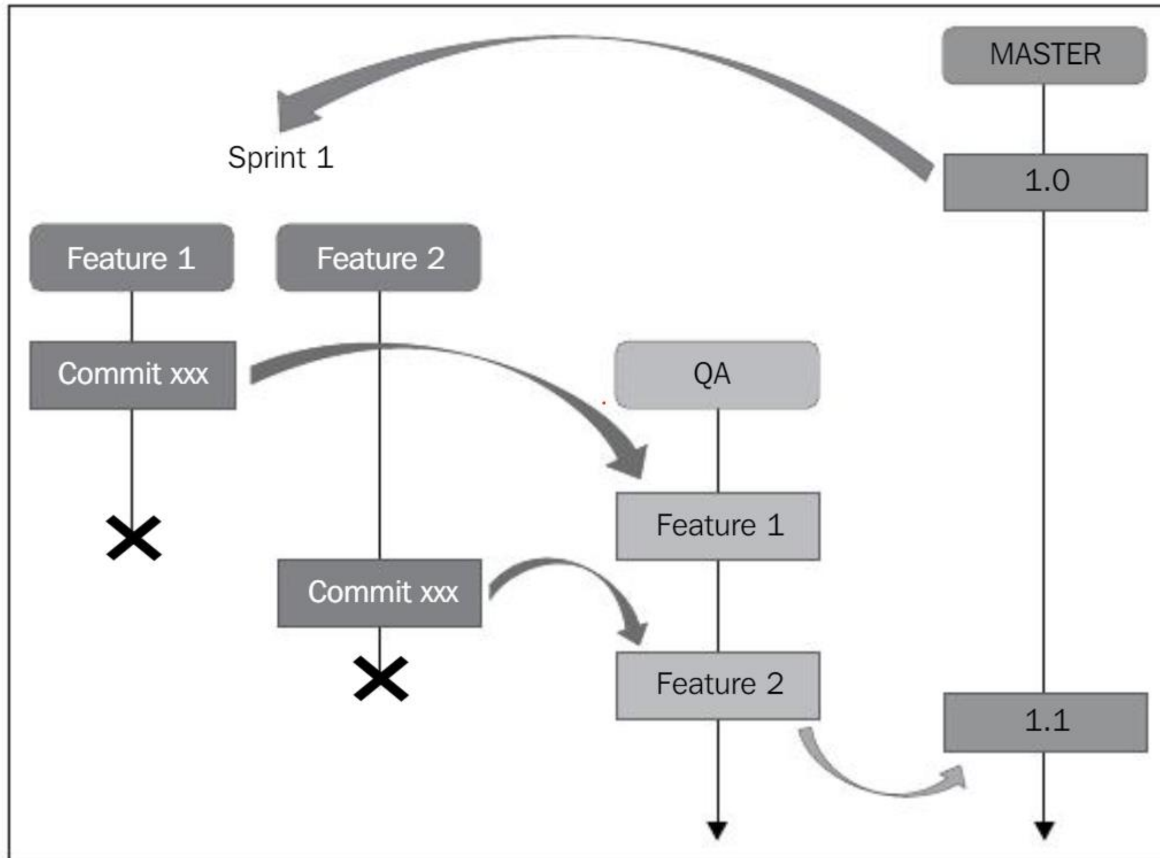
Le tableau suivant montre les différentes branches que vous devriez créer si vous choisissez de mettre en œuvre la stratégie Git Flow.

Branches	Aim	Naming convention	Created from	Merge into	Delete it after merging?
Master	Production state	master	-	-	No
Dev	Features for the next big release	dev	Master	Master	No
Feature	Creating a new feature	Name of the feature	Dev	Master and Dev	Yes
Release	Minor changes between two big releases	Name of the current project versions 1.1, 1.2, and so on	Dev	Master and Dev	Yes
Hotfix	To fix critical bugs on production	Name of the current versions 1.1.1, 2.1.3, and so on	Master	Master, Release, or Dev	Yes

7.1.2 BPF – Branch Per Feature

Le modèle de flux Git peut être adapté à un projet utilisant GitHub, mais cela ne convient pas toujours. Adam Dymitruk a proposé un modèle de stratégie de branches plus efficace en combinant Git avec l'Intégration Continue. Selon lui, pour un développement efficace, il est recommandé de diviser le projet en plusieurs sprints avec plusieurs fonctionnalités à développer. Chaque fonctionnalité doit être

développée dans une branche séparée et dédiée, puis fusionnée avec la branche de développement une fois prête. Pour être notifié rapidement en cas d'erreur, il est conseillé d'utiliser un outil d'Intégration Continue sur une branche d'Assurance Qualité. Enfin, une fois que les tests sont réussis, la branche d'Assurance Qualité est fusionnée avec la branche principale et une nouvelle version est créée.



L'objectif de cette stratégie est le suivant :

• Tous les travaux sont divisés en branches de fonctionnalités.

• Toutes les branches de fonctionnalités sont créées à partir de la branche master (à partir de la dernière version publiée). Lorsque vous commencez un sprint, vous créez simultanément vos branches de fonctionnalités.

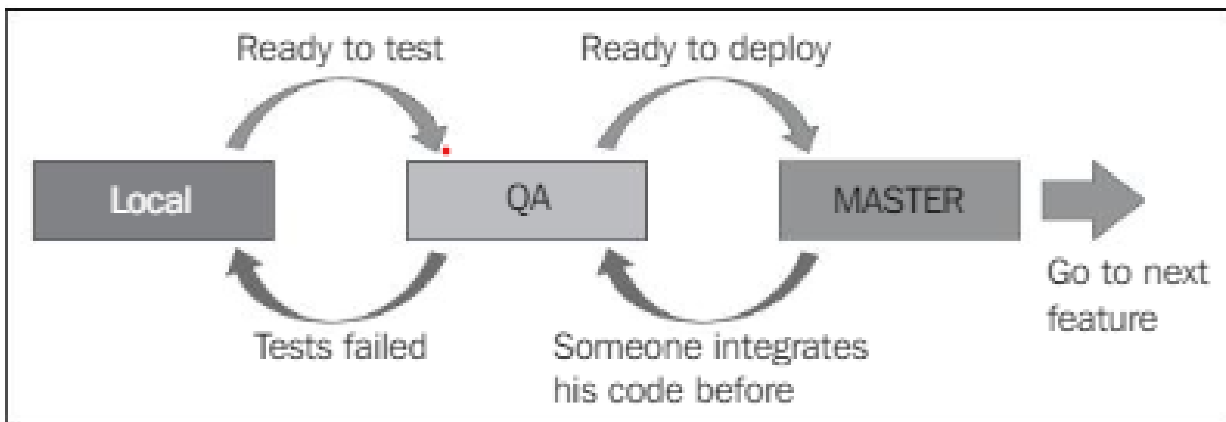
• Tester votre code plus tôt.

Every time you start a sprint, create the feature

branches and QA.

7.2 Working with continuous Integration using Git

Travailler avec l'intégration continue (CI) signifie que vous devez combiner le travail fréquemment et pousser les fonctionnalités dès qu'elles sont prêtes. Le but de cette méthode est de livrer les fonctionnalités plus rapidement au client en les déployant dès qu'elles sont prêtes. De plus, cela permet de réduire les problèmes de déploiement car les déploiements sont plus petits. Git est capable de mettre en place un modèle agile efficace en permettant la création et la fusion de branches. Pour intégrer les modifications, il faut les intégrer localement et les tester sur une machine d'intégration privée. Si les tests échouent, il faut corriger les bugs et tester à nouveau. Lorsque les tests réussissent, le code peut être promu sur une branche publique. En cas d'intégration de code par un tiers, il faudra recommencer les tests.



7.3 Git tools you might like.

7.3.1 On Linux

- Git-cola
- Gitg
-

7.3.2 On Windows

- TortoiseGit
- GitHub client
- msysGit

7.3.3 On Mac

- GitX
- Gitbox