Les branches

Une branche dans Git est un pointeur mobile léger vers un de ces commit. Par exemple lorsque vous créez un dépôt avec la commande suivante :

```
git init
```

- Git crée automatiquement la branche **master**.

Au fur et à mesure de la création de commits Git déplace le pointeur **master**, un pointeur mobile on le répète, sur le dernier commit.

Si on crée une autre branche alors on crée un autre pointeur mobile. Ci-dessous on a créé une branche **dev**.

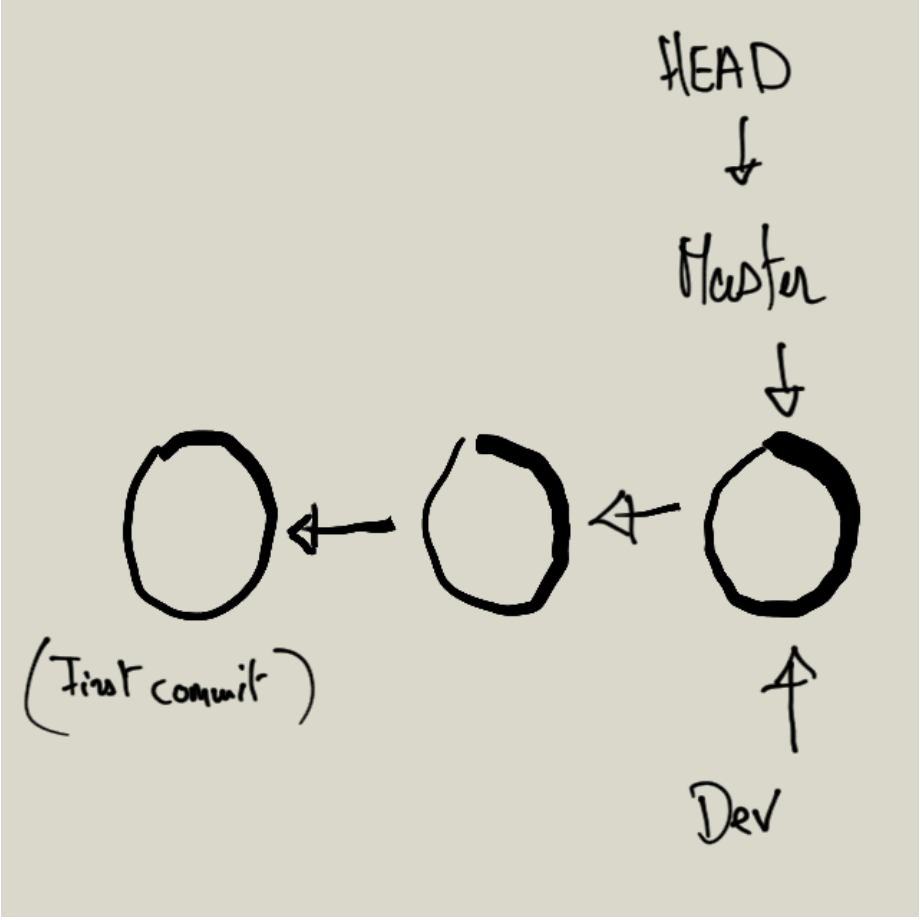

Figure 1: mobile_pointeur

Le système de branche dans Git est très optimisé, en effet une branche dans

Git n'est qu'un simple fichier d'environ 40 caractères. Donc vous pouvez créer autant de branche que vous le souhaité.

- La commande pour créer une branche est simple à retenir :

```
$ git branch dev
```

Affichez maintenant l'ensemble des branches se trouvant sur votre dépôt :

```
git branch
```

Pour savoir sur quelle branche Git se trouve il utilise un autre pointeur spécial : **HEAD**.

git checkout

Pour se déplacer sur l'autre branche, on utilise la commande checkout comme suit :

```
# on déplace le HEAD sur la branche dev
$ git checkout dev
```

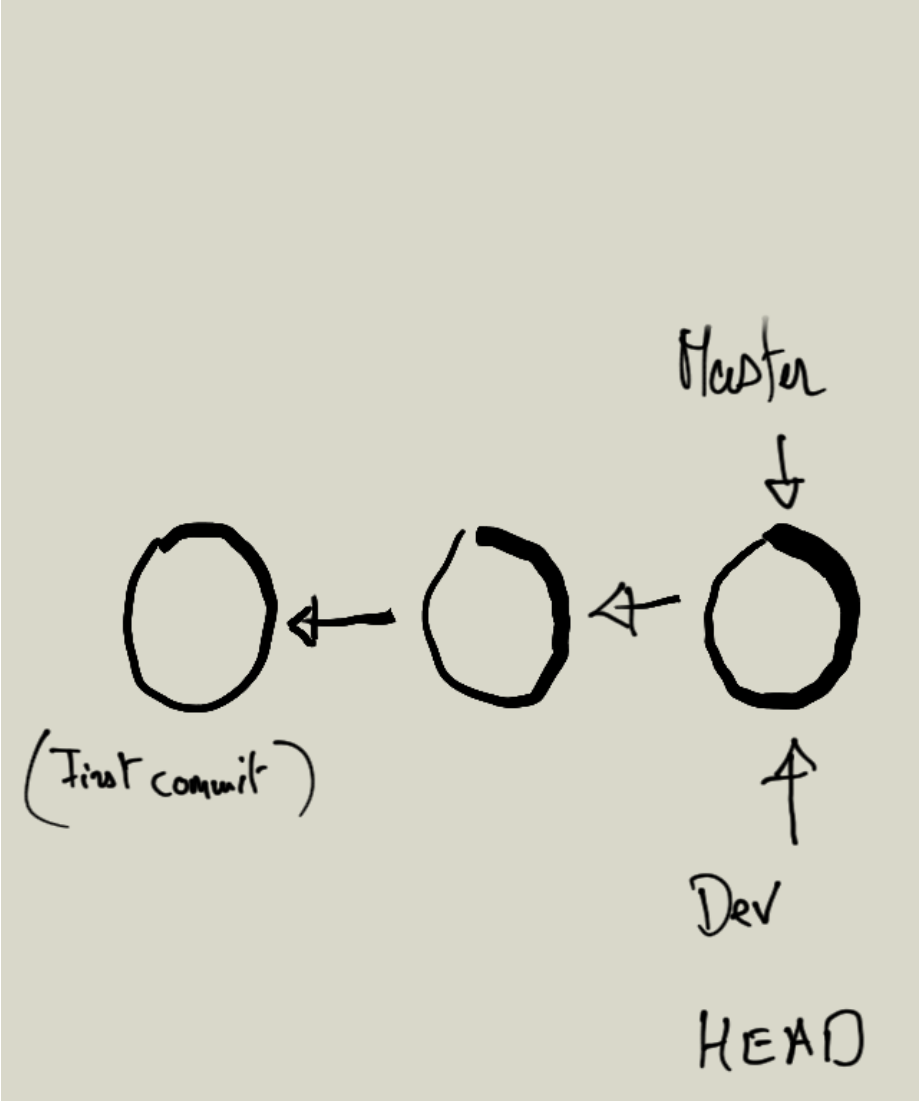

Figure 2: pointeurs

Maintenant si on crée un commit sur cette nouvelle branche dev les 2 pointeurs HEAD et dev se déplacent sur le dernier commit créé Dev + HEAD :

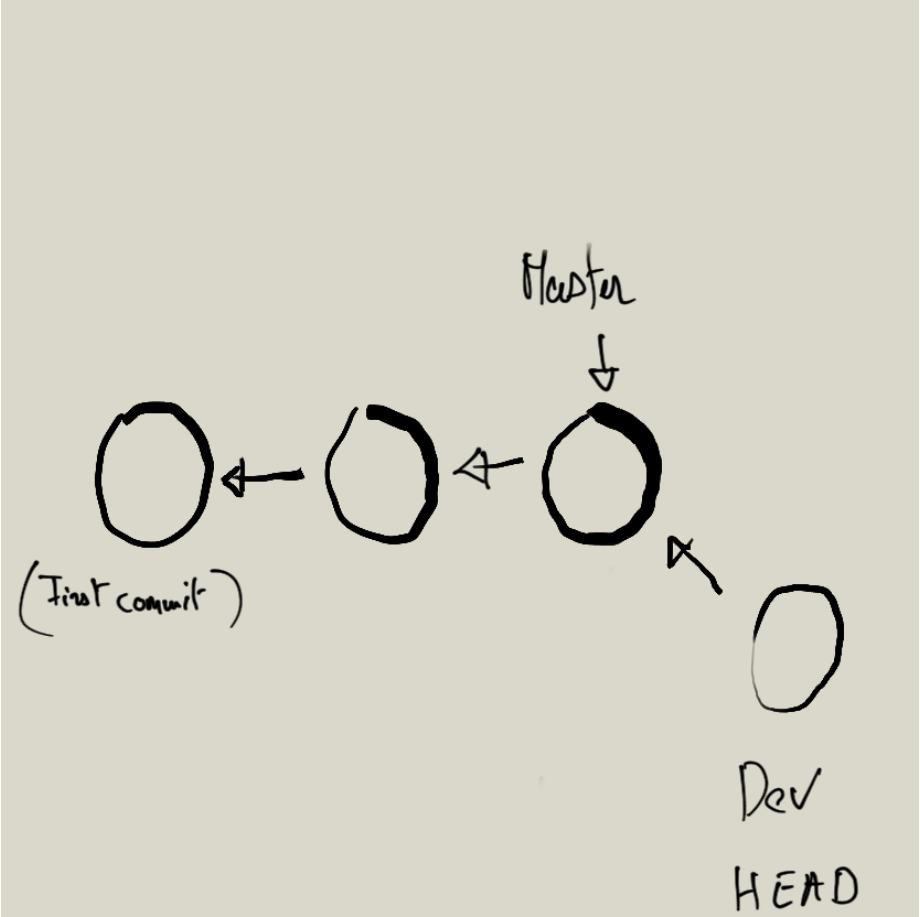

Figure 3: deplacement

Si on souhaite revenir sur la branche master on exécutera la commande suivante :

```
# On déplace le HEAD sur la branche master
$ git checkout master
```

Vous pouvez également créer une branche et vous déplacez dessus en une seule commande :

```
# on déplace le HEAD sur la branche master
$ git checkout -b feature_header
```

Les branches peuvent divergées

Il arrive que l'historique sur la branche master avance par rapport à une autre branche. Dans ce cas l'historique Git diverge :

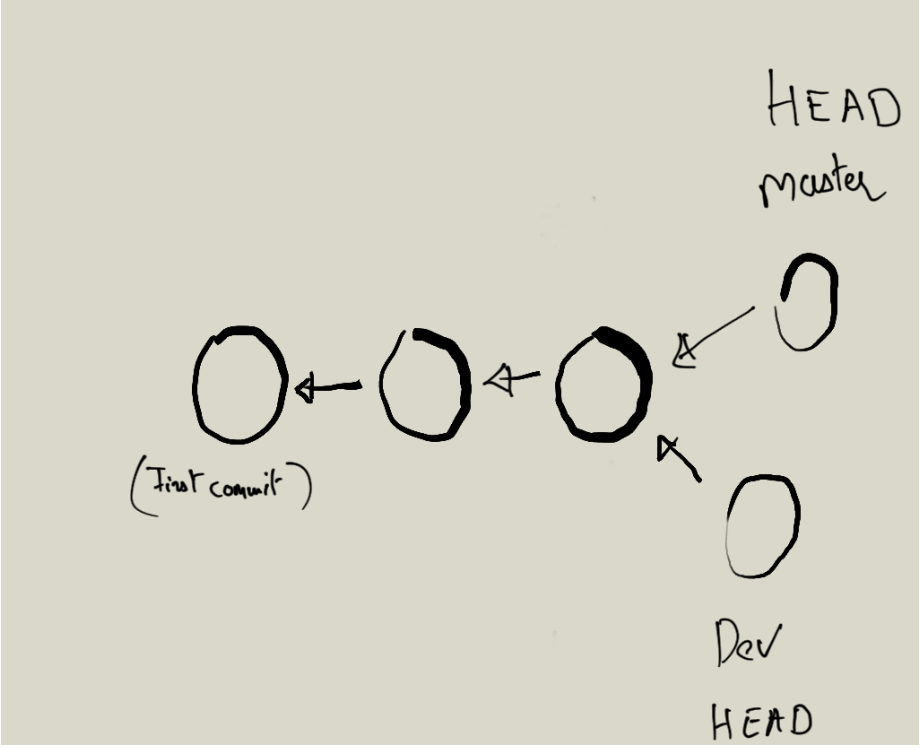

Figure 4: diverge

Lorsque vous allez merger il y aura alors peut-être des conflits entre deux versions d'un même fichier. Git propose un système de gestion de conflit de version que nous verrons plus loin.

Suppression d'une branche

Vous pouvez facilement supprimer une branche en utilisant l'une des commandes suivantes :

```
git branch -d [nom de votre branche]
```

Forcer la suppression d'une branche :

```
git branch -D [nom de votre branche]
```

Git vous empêchera de supprimer une branche qui n'est pas dans l'état "copie de l'espace de travail propre".

Merge sans divergence : fast-forward

Nous allons voir comment merger (mélanger) deux branches.

Supposons qu'il n'y a pas eu de divergence entre le master et la branche dev.

Une fois que vous avez terminé votre travail sur la branche dev, vous pouvez merger celle-ci dans la branche master.

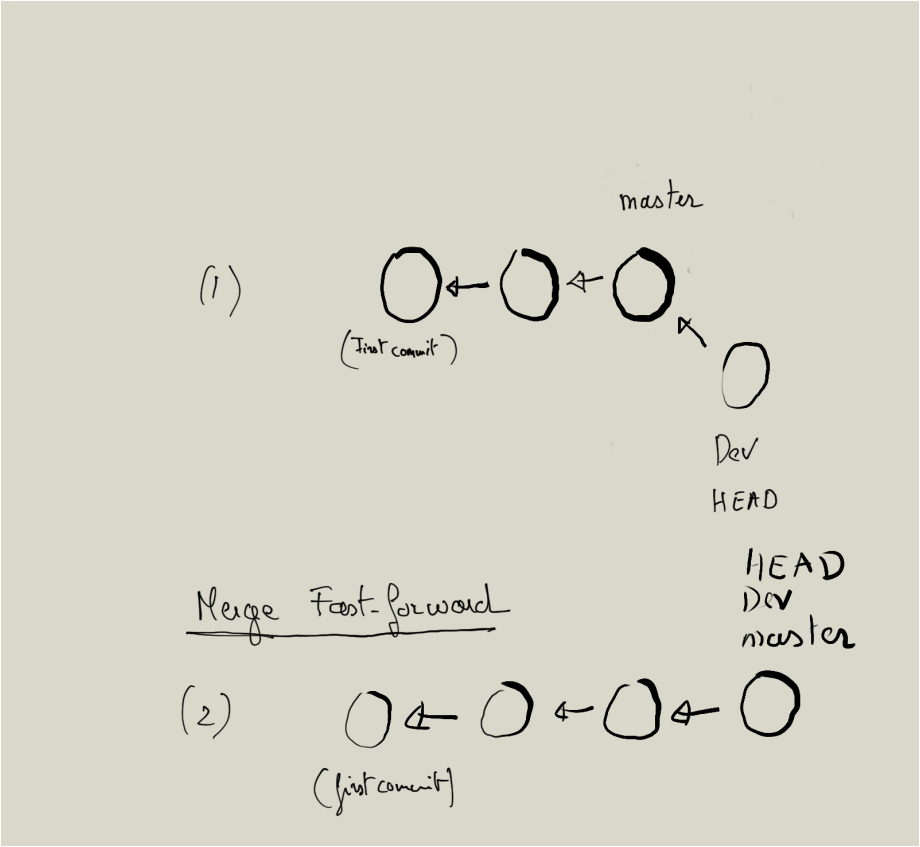

Figure 5: merge

Voici les commandes à exécuter pour faire le merge :

```
# On suppose que l'on est sur dev
git checkout master

# On merge dev dans master
git merge dev
```

Notez qu'il existe une commande pour faire un commit de merge même si on est dans une situation sans divergence de branche :

```
git merge dev --no-ff
```

En réalité lorsque la branche master n'avance pas par rapport à la branche dev par exemple, Git est en mesure de linéariser l'historique des commits. Et donc il ne crée pas de commit supplémentaire. Créez un commit de merge dans cette situation vous permettra de revenir éventuellement plus facilement sur ce dernier et de savoir qu'il y a une branche de faite à ce moment là précisément.

```
M1 -- -- -- -- M2
  \            /
   D1 - D2 - D3
```

Dans ce cas il est alors plus facile de retrouver le commit M1 (directement).

Dans le cas où vous avez fait un fast-forward voilà ce qui se passe pour le même exemple ci-dessus :

```
M1 - D1 - D2 - D3
```

Revenir au commit M1 est plus long dans ce cas.

Lister toutes les branches non mergées

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

Exercices d'application

01 Exercice merge sans divergence

Pour la suite des exercices récupérez les sources du projet : branche demo et dezippez le projet sur votre bureau. Ouvez vscode et un terminal Git bash dans le projet.

Initiailisez le projet Git.

Créez un fichier README.md et placez le texte suivant dans ce fichier :

```
# Projet de branches Git

Dans ce projet nous allons créer des branches pour apprendre à utiliser Git et son système d
```

Faites votre premier commit “first commit” sur la branche master. Puis créez la branche feature_colors.

Modifiez les couleurs de fond de la page index.html en utilisant les couleurs suivantes :

- #8c0032
- #fa5788
- #c2185b

Changez la couleur du texte également, utilisez le code hexa suivant :

- #ffeeff

Terminez en faisant un commit pour ces changements (un commit pour le changement de couleur).

Vous allez maintenant quitter la branche feature_color et revenir sur la branche master. Que constatez-vous au niveau du rendu de votre page HTML ?

Terminez l’exercice en mergent la branche dev dans la branche master.

Si vous affichez maintenant vos logs théoriquement vous ne devriez en avoir que 2 :

- firts commit
- le commit réalisé sur la branche feature_color

Que pouvez-vous en déduire sur la manière de merger de Git dans ce cas de figure ?

- Comme le travail a été réalisé sur la branche feature_color supprimez cette branche.

Créez maintenant la branche feature_image et récupérez le dossier images à l’adresse suivante : image

- Placez l’image dans la section.content et faites un commit pour les changements sur cette branche.
- Mergez maintenant les modifications dans la branche master à l’aide de la commande suivante, Git vous demandera d’écrire un commit de merge :

```
git merge feature_image --no-ff
```

Visualisez le graph des commit à partir de la branche master maintenant :

```
git log --oneline --graph
```

Normalement vous devriez voir la branche feature_image avec son commit de merge, notez qu'elle est non-linéarisée.

02 Exercice gestion des conflits

Créez la branche **feature_header** et ajoutez le code suivant dans le header :

```
<h1>Git introduction à la notion de branche
    <br><small>Author : Alan</small>
</h1>
```

Faites votre commit habituel pour cette feature.

- Revenez sur la branche master sans merger la branche feature_header dans la branche master.
- Ajoutez maintenant le code suivant dans le header :

```
<h1>Git introduction à la notion de branche
    <br><small>Author : Tony</small>
</h1>
```

- Créez un commit sur la branche master. Ainsi votre branche master a avancé par rapport à la branche feature_header.
- Essayez de merger maintenant la branche feature_header dans la branche master. Un conflit aura lieu précisément sur les changements que nous avons apporté dans le header :

```
<header>
    <h1>Git introduction à la notion de branche
<<<<<<< HEAD
        <br><small>Author : Tony</small>
=======
        <br><small>Author : Alan</small>
>>>>>>> feature_header
    </h1>
</header>
```

Tout ce qu'il y a entre la tags <<<<<<< HEAD et ======= correspond vous l'aurez compris à la version du master et tout ce qu'il y a en dessous du tag ======= jusqu'à >>>>>>> feature_header correspond à la version de la branche feature_header.

Choisissez la version de la branche feature_header.

Git n'a pas encore mergé votre branche pour finaliser vous devez ajouter dans l'index les modifications que vous venez de faire dans le fichier index.html. Ce qui marquera comme résolu les conflits dans ce fichier pour Git. Et faire un commit pour créer effectivement le commit de merge.

Notez qu'il y a forcément un commit de merge dans ce cas, car les branches diverges.

- Supprimez les branches features qui ne sont plus utiles.

Créez maintenant la branche feature_footer et modifiez le footer comme suit :

```
<footer>Git version 2.23</footer>
```

Faites un commit sur cette branche pour cette modification.

- Vous devez maintenant retourner sur la branche master et fixer un bug dans cette branche.

Créez une fois que vous êtes sur la branche master la branche **fix_nav** et ajoutez le lien suivant (en bas du ul.li) :

```
<nav id="navigation" role="navigation">
    <a href="#">git init</a>
    <a href="#">git diff</a>
    <a href="#">git log</a>
    <a href="#">git status</a>
    <a href="#">git merge</a>
    <a href="#">git checkout -b</a>
</nav>
```

Vous faites vos tests et tout marche comme vous voulez, mergez la branche fix_nav dans la branche master et la branche feature_footer.

Et enfin ajoutez un lien vers la documentation de Git dans le footer.

Terminez votre travail et mergez cette branche dans le master.

Bravo vous avez terminé votre travail !

Pour la suite récupérez le dossier deploy.zip sur le serveur de cours.

03 Exercice gestion de conflit

Exécutez le script deploy_conflit.sh dans votre dossier.

Placez-vous sur la branche **master** modifiez les pages suivantes : index.html, category.html et contact.html.

Modifiez le code suivant dans chacun de ces fichiers, faites un commit par modification.

Dans la page category.html :

```
<html><header></header><body><h1>Categories</h1>
<footer>home/category/mentions légales</footer></body></html>
```

Dans la page contact.html :

```
<html><header></header><body><h1>Contactez-nous</h1>
<footer>home/category/mentions légale</footer></body></html>
```

Dans la page index.html :

```
<html><header></header><body><h1>Page d'accueil</h1>
<footer>home/category/mentions légales</footer></body></html>
```

Une fois que vous avez fait ces modifications essayez de merger la branche **feature_pages** dans la branche **master**.

Le remisage

Lorsque votre branche est dans un état "epace de travail propre" ou "nothing to commit", vous pouvez quitter la branche sur laquelle vous êtes pour aller sur une autre branche. Mais si la branche n'est pas dans l'état "nothing to commit" alors Git vous empêchera de quitter cette branche. Git vous oblige à finir la feature correctement avant de quitter la branche.

Notez bien que sur une branche donnée pour des fichiers suivis en version, donc connus de Git, **si votre/vos fichier(s) est/sont modifié(s) dans l'espace de travail (Working directory) ou/et est:sont dans l'index alors Git vous empêchera le basculement vers une autre branche.** Insistons sur le fait que le/les fichier(s) doit/doivent être connu par Git, si vous ajoutez un fichier non suivi dans votre dossier et même si vous placez ce fichier dans l'index, mais qu'il n'est pas encore suivi, vous pouvez alors dans ce cas quitter la branche.

Il est important de noter que vous ne devez pas faire un commit pour faire un commit. . . Et surtout pensez qu'un commit est une sorte de sauvegarde, c'est faux! On vous le rappelle un commit va valider une fonctionnalité ou un groupe de fonctionnalités dans votre application, il est donc très important de faire ce commit en accord avec le développement pour que votre historique reste cohérent.

Il arrivera cependant des situations où vous serez obligé de quitter la branche où vous vous trouvez alors que votre feature n'est pas encore terminée, par exemple si vous devez fixer un bug sur la branche master en urgence (. . .). Dans ce cas Git possède une commande pour remiser le code, comprendre le mettre de côté. Elle vous permettra de quitter la branche en cours, sans faire de commit, et d'y revenir plus tard en récupérant le code remisé pour terminer votre feature. En effet, la remise met le travail en cours de côté et à pour conséquence de mettre votre dépôt dans l'état "rien à valider, la copie de travail est propre".

```
# Remiser le code non terminé
git stash
```

Git vous retournera un code similaire à ce qui suit :

```
Saved working directory and index state \
```

```
  "WIP on master: 675467 added the index file"
HEAD is now at 089876 added the index file
(To restore them type "git stash apply")
```

Vous pouvez alors quitter cette branche.

Git a remisé votre code et vous pouvez maintenant basculer sur une autre branche. Le stash vous permettra également de récupérer le code remisé plus tard.

```
git branch
# affiche dev

# On remise
git stash

# Rebasculement sur la branche dev
get checkout dev

# Listez sur la branche les stash
git stash list

# On recupère ce qui était dans la stash
# Et on l'applique => récupération de son code modifié
git stash apply

#On supprime ce qui se trouve dans la remise
git stash drop
```

L'option de commande **apply** essayera d'appliquer les modifications de la pile des stash correspondante à votre branche. **Il appliquera la dernière remise dans la pile que vous avez laissée**.

Par défaut Git remettra tout dans le WD, si vous aviez du code dans l'index, il faudra taper la commande suivante :

```
# Essaye de remettre ce qui était dans l'index
git stash apply --index
```

Vous pouvez maintenant supprimer ce qui se trouve dans la remise et terminer votre développement :

```
git stash drop
```

Appliquer un stash de la remise en particulier :

```
$ git stash list

# Vous pouvez avoir plusieurs stash dans la remise
stash@{0}: WIP on master: 01524a bug important
```

```
stash@{1}: WIP on master: 452ets un test non validé

# Remet le code remisé dans le WD, on peut ajouter --index pour le re-stagé si il l'était
$ git stash apply

# ou appliquer un stash particulier
$ git stash apply stash@{1}

# Supprimer le stash appliqué
$ git stash drop stash@{1}

# supprime le premier stash de la pile
$ git stash drop
```

04 Exercice stash

Récupérez le fichier deploy_stash.sh sur le serveur, il va créez un dossier et un dépôt, et exécutez le code dans votre dossier d'exercices :

```
# Création de notre dépot example-stash
sh deploy_stash.sh

# Faites un git status pour voir l'état de votre dépot
git status

# Puis repérez sur quelle branche vous êtes
git branch
```

Ajoutez un footer dans le fichier index.html, faites cela sur la branche **feature_html**.

Essayez de quitter la branche feature_html où vous vous trouvez, théoriquement c'est impossible. Git vous empêchera du quitter une branche qui n'est pas propre.

Remisez votre code puis une fois sur la branche master ajoutez le fichier readme.md avec le code suivant :

```
# Stash
```

Commitez et revenez sur la branche feature_html pour terminer le travail en cours.

Tags

Il existe deux types de tags : les tags annotés ou les tags légers. On les utilise pour marquer un état important du logiciel.

- Un tag léger est similaire à une branche, mais contrairement à une branche, un tag n'est pas mobile. Il est associé à un commit.
- Un tag annoté est un objet dans Git, il pointe également sur un commit donné et n'est donc pas mobile. Il possède une somme de contrôle et contient le nom l'email de l'auteur, la date, un message et peut être signé et vérifié (GPG).

Nous vous conseillons de créer des tags annotés si ils doivent restés dans l'historique. Dans le cas contraire si vous souhaitez taguer un commit de manière temporaire on utilisera un tag léger.

Voici quelques commandes utiles pour la création de ces tags :

```
# Tag annoté
$ git tag -a v1.0 -m "version 1 de l'application"

# Tag léger peu utilisé
$ git tag v1

# Etiquetter après coup sur un commit donné
$ git tag -a v1.0.1 -m "version 1.0.1" 9fceb2
```

list & show tags

```
# liste les tags
git tag

# Rechercher des tags particuliers
git tag -l 'v8.*'

# Voir un tag annoté
git show v8.2
```

Introduction aux branches distantes

Git server

Vous pouvez créer un serveur git sur un serveur distant, sur votre machine ou même sur une clé USB ou un autre prériphérique de stockage.

Créez dans le dossier de cours un dossier **GitServer** dans lequel nous allons travailler.

Pour des raisons pratiques dans un premier temps nous allons créer un serveur Git localement.

Créez un serveur Git **my__lessons__server.git** sur votre machine locale :

```
# Sur votre bureau
$ cd GitServer
$ mkdir my_lessons_server.git
$ cd my_lessons_server.git

# Création du server Git
$ git --bare init
```

Dossiers du serveur Git :

```
# Contenu du dossier my_lessons_server.git
branches  config  description  HEAD  hooks  info  objects  refs
```

Sur le bureau également ou dans votre dossier de cours créez maintenant un dossier puis récupérez la branche distante du serveur Git :

```
# Sur le même poste
$ cd my_lessons
$ git init
$ echo "# Lessons" > README.md
$ git add .
$ git commit -m 'first commit'
# Spécifiez un chemin absolu sur votre machine
# pour indiquer où se trouve votre serveur
# origin est arbitraire mais par convention on l'utilise
# pour nommer son dépôt distant
$ git remote add origin C:\Labs\my_lessons_server.git

# Sous Linux ou Mac dans un media
# $ git remote add origin /media/antoine/3065-6232/my_lessons_server.git

# Publier sur la branche distante (serveur)
# git push [nom-distant] [nom-de-branche]

# push la branche master dans votre dépôt dans
# le serveur origin
$ git push origin master
```

04 Exercice

Créez maintenant un dossier **my__app2** et récupérez ce que vous avez publié sur le serveur **my__lessons__server.git** à l'aide des commandes suivantes :

```
$ cd my_app2
$ git init
```

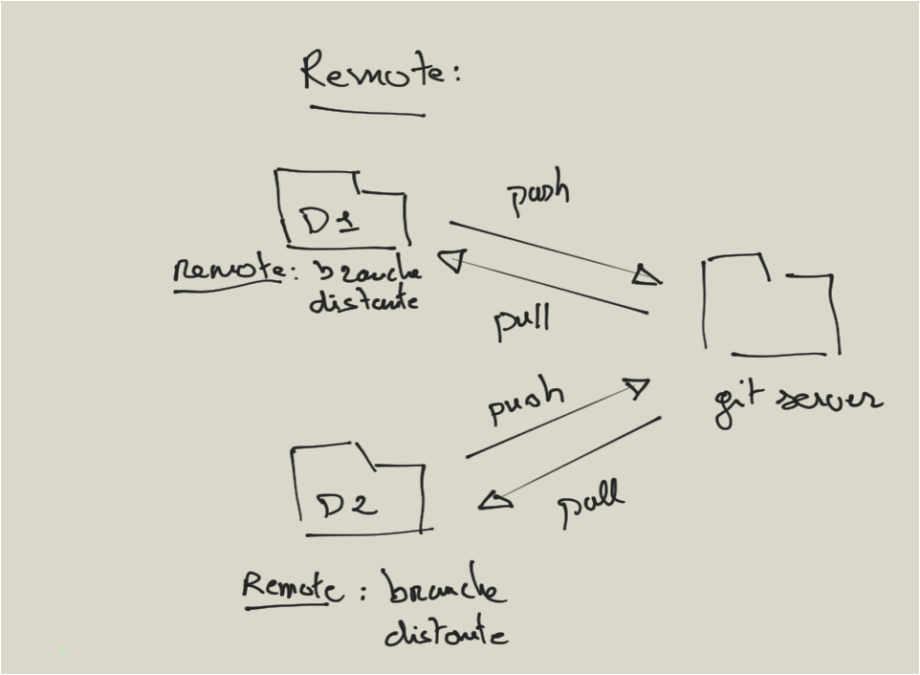

Figure 6: server

```
$ git remote add origin C:\Labs\my_lessons_server.git

$ git pull origin master
```

Listez ce que vous avez dans votre dossier vous devriez avoir le fichier README.md

Résumé des commandes remote

La commande remote permet de travailler avec des serveurs distants. Voici un résumé des commandes utiles.

```
# Lister les dépôts distants
git remote

# Lister en mode verbeux
git remote -v

# Ajouter un dépôt distant
git remote add [alias] [chemin_du_serveur_distant]

# Supprimer ou renommer un dépôt distant
# [alias] permet de nommer votre dépôt distant
git remote rm [alias]
git remote rename [alias] [new_alias]
```

Précision sur les commandes de publication et de récupération de branches distantes :

```
# Fetch récupérer la branche distante
# localement sans fusion avec sa branche master
$ git fetch origin

# Comparer les différences entre master
# local et distant (après un fetch)
$ git log master..origin/master

# git pull équivalent à git fetch + git merge
$ git fetch origin
$ git merge origin/master

# La commande suivante vous permettra de publier une branche distante
$ git push [nom-distant] [nom-de-branche]
```

Si on veut avoir des informations sur une branche distante :

```
# Inspecter un dépôt distant
git remote show origin
```