

Formation Git

XIII - Projets multi-dépôts

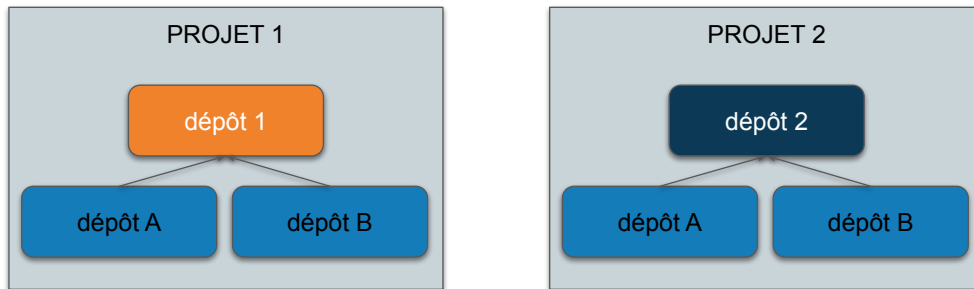


Arnaud MERCIER
arnaud.mercier@hexotech.fr



Nous allons voir dans ce chapitre comment gérer un projet qui est constitué de plusieurs dépôts Git

Enjeux des projets multi-dépôt

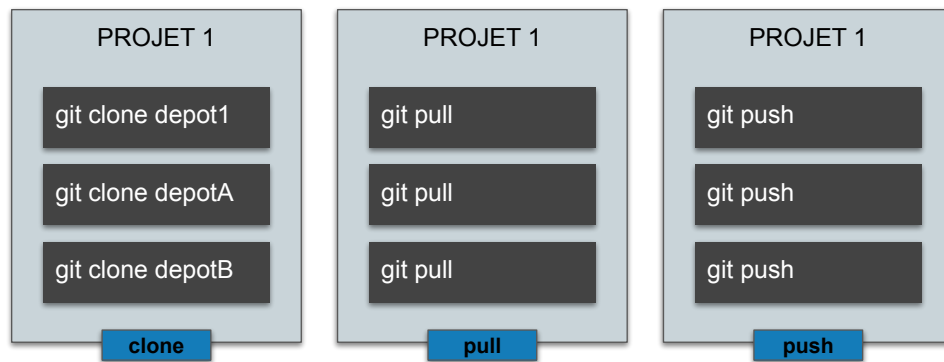


Souvent, les projets utilisant git ne se contentent pas d'un seul dépôt. on se retrouve donc dans la majorité des cas avec plusieurs dépôts. Chaque dépôt correspond à une partie du projet (par exemple une lib).

Plusieurs questions apparaissent alors:

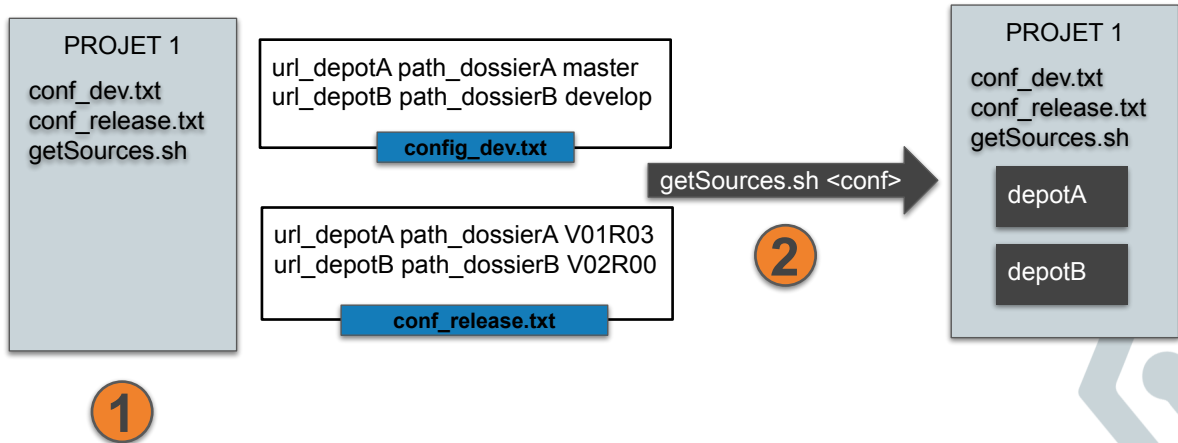
- quelle est la liste des dépôts pour mon projet?
- quelle est l'url de mes dépôts?
- quelle version doit on prendre pour telle ou telle dépôt?
- etc

Projet multi-dépôt: à la main



Il est possible de faire tout cela à la main en partant du principe que l'on peut répondre à toutes les questions. Malheureusement cela est loin d'être simple et surtout fiable. Sans compter que si vous avez des dizaines de dépôts, cela peut vite devenir très fastidieuse.

Projet multi-dépôt: via des scripts

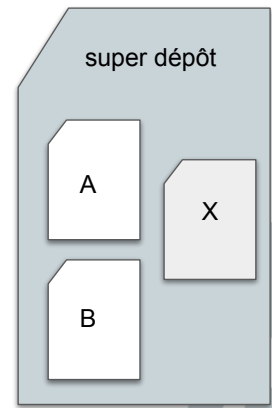
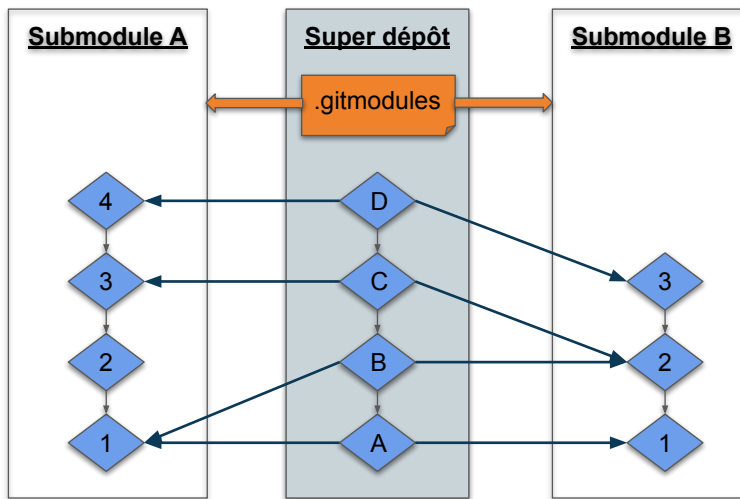


On peut alors penser aux scripts. c'est à dire créer un script maison qui permet automatiquement de cloner et mettre à jours les dépôts de votre projet.

Le mieux, est alors de créer un "super dépôt" qui vas contenir les scripts et des fichiers de configuration de votre projet. ces fichiers vont alors lister les dépôts indispensables à votre projet ainsi que la version à utiliser pour vos sous dépôts. ces fichiers seront alors les entrées pour vos scripts.

On peut alors versionner et tagger ce "super dépôt" pour gérer les version "chapeau" de votre projet.

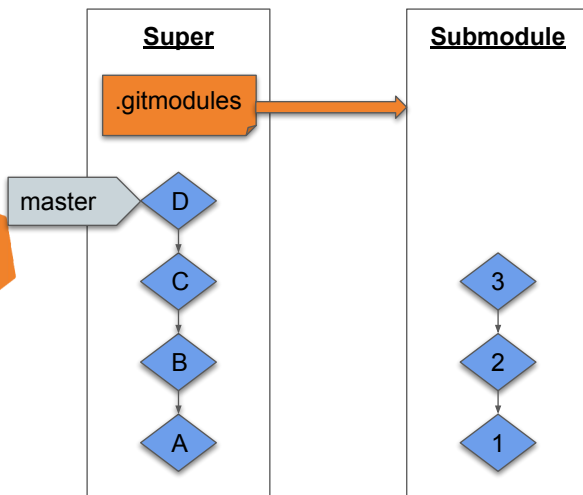
Projet multi-dépôt: Via les submodules



Le fichier `.gitmodules` permet de lister les sous modules du dépôt principale aussi appelé "super dépôt"

Git détectera alors les évolutions des sous dossiers de super dépôt comme de nouvelles références aux sous modules

Submodules: ajout



```
git submodule add [url] [local-path]
```

```
git status
```

```
Changes to be committed:  
  new file:   .gitmodules  
  new file:   module
```

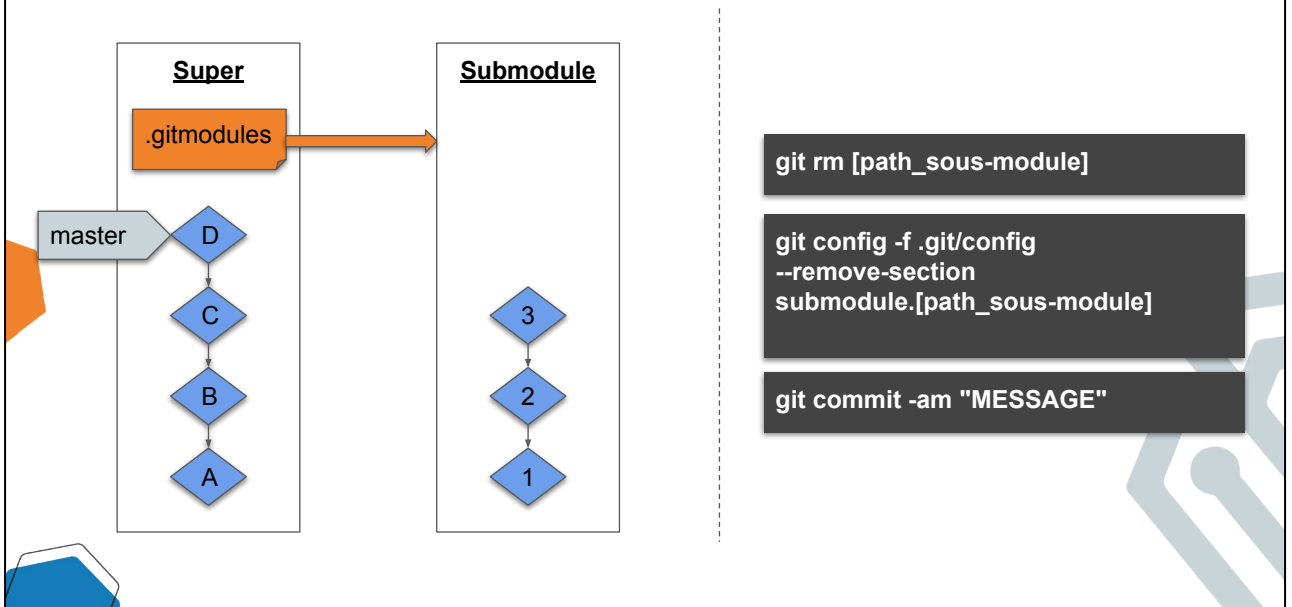
```
git diff --cached --submodule
```

```
+ [submodule "module"]  
+   path = module  
+   url = ../module  
Submodule module 0000000...c69c6d5  
(new submodule)
```

```
git commit -m 'add submodule'
```

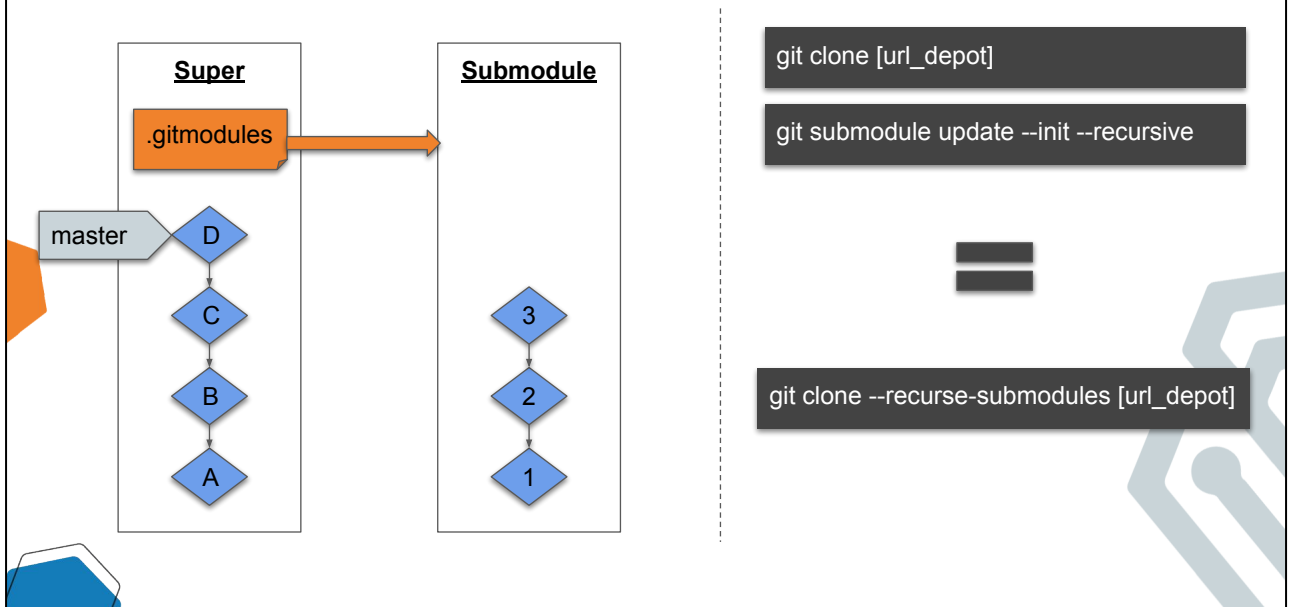
Comme l'URL dans le fichier, .gitmodules est ce que les autres personnes essaieront en premier de cloner et de tirer, assurez-vous que cette URL est effectivement accessible par les personnes concernées. Par exemple, si vous utilisez une URL différente pour pousser que celle que les autres utiliseront pour tirer, utilisez l'URL à laquelle les autres ont accès. Vous pouvez surcharger cette URL localement pour votre usage propre avec la commande `git config submodule.DbConnector.url PRIVATE_URL`.

Submodules: supprimer un sous module



Comme l'URL dans le fichier, `.gitmodules` est ce que les autres personnes essaieront en premier de cloner et de tirer, assurez-vous que cette URL est effectivement accessible par les personnes concernées. Par exemple, si vous utilisez une URL différente pour pousser que celle que les autres utiliseront pour tirer, utilisez l'URL à laquelle les autres ont accès. Vous pouvez surcharger cette URL localement pour votre usage propre avec la commande `git config submodule.DbConnector.url PRIVATE_URL`.

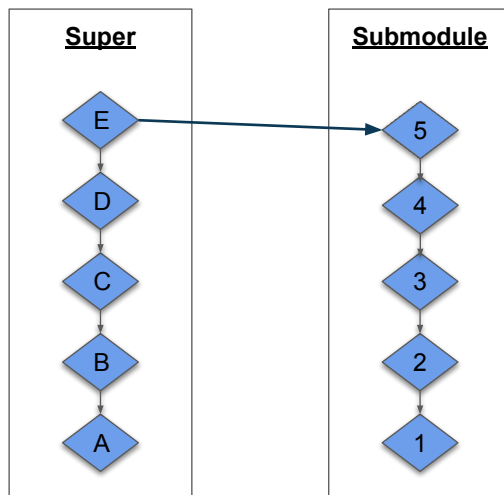
Submodules: cloner



Maintenant, vous allez apprendre à cloner un projet contenant des sous-modules. Quand vous récupérez un tel projet, vous obtenez les différents répertoires qui contiennent les sous-modules, mais encore aucun des fichiers

Le répertoire css est présent mais vide. Vous devez exécuter deux commandes : git submodule init pour initialiser votre fichier local de configuration, et git submodule update pour tirer toutes les données de ce projet et récupérer le commit approprié tel que listé dans votre super-projet SITE_SAFRAN

Submodules: pull



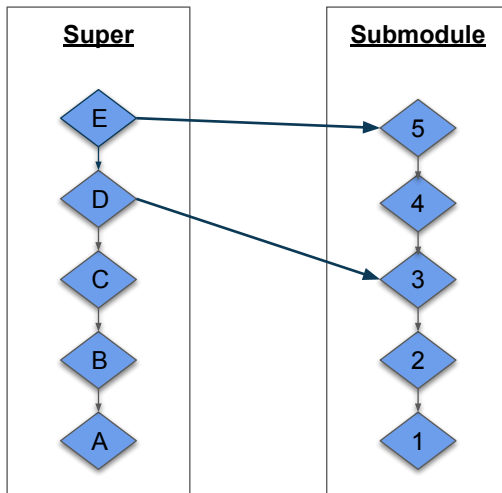
```
git pull
```

```
git submodule update --remote
```

Pour se mettre à jour depuis le dépôt distant, il est nécessaire de réaliser deux actions:

- mettre à jour le Super Dépôt via la commande: `git pull`
- mettre à jour les sous modules via la commande: `git submodule update --remote`

Submodules: update



```
git submodule update --remote
```

```
git diff --submodule
```

```
Submodule module c69c6d5..4d802d1:  
> commit 4  
> commit 5
```

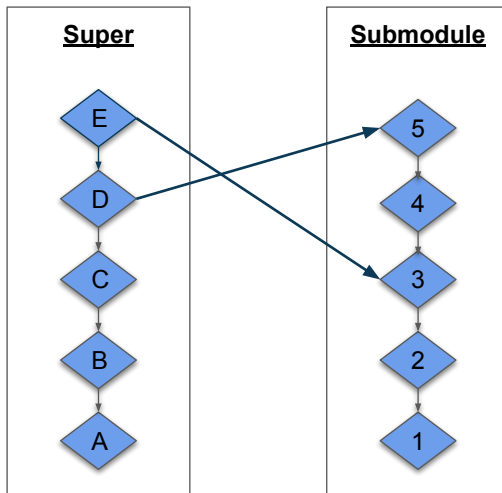
```
git add [module]
```

```
git commit -m"update du module"  
git push
```

Pour modifier la version de sous module utilisée par le Super Dépôt, il faut:

- Mettre à jour le sous module via la commande : `$ git submodule update --remote`
- Se positionner sur le commit souhaité
- Faire le commit de la nouvelle référence dans le Super Dépôt

Submodules: Changer la version utilisée



```
cd [path_submodule]
git fetch / git pull
git checkout [REF]
cd ..
```

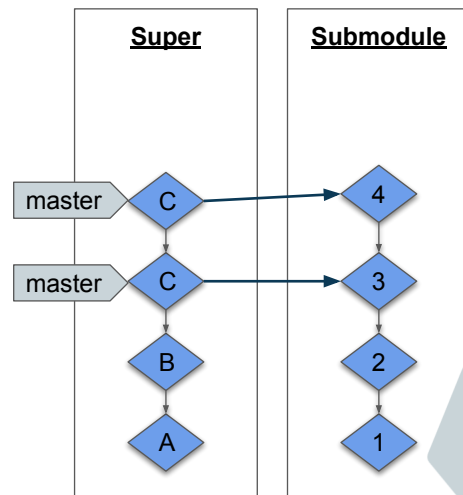
```
git status
git commit -am "MESSAGE"
```

Pour modifier la version de sous module utilisée par le Super Dépôt, il faut:

- Mettre à jour le sousmodule via la commande : `$ git submodule update --remote`
- Se positionner sur le commit souhaité
- Faire le commit de la nouvelle référence dans le Super Dépôt

Submodules: modifier

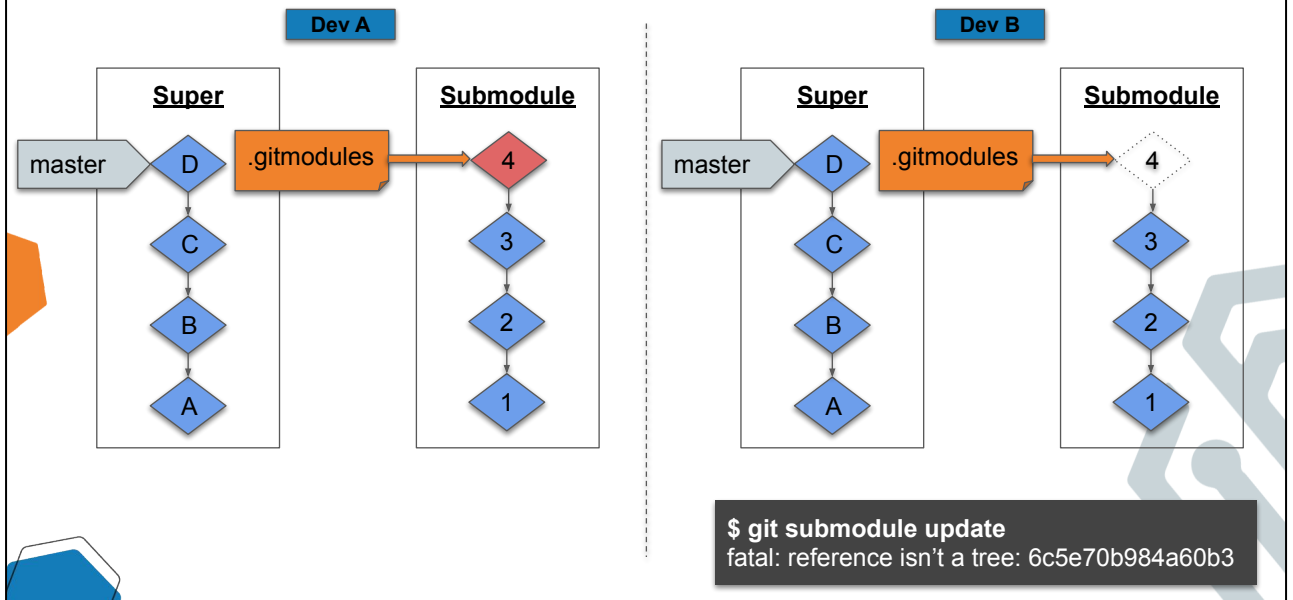
- 1 `cd [sub_module]`
faire les modifications
- 2 `git add [fichiers_modifiés]`
`git commit`
- 3 `git push`
- 4 `cd -`
`git commit -am "message"`
`git push --recurse-submodules=check`



Pour modifier le contenu d'un sous module et le prendre en compte dans le Super Dépôt, il faut:

- Faire les commits et push dans les sous modules
- Faire un commit et push des nouvelles références dans le Super Dépôt

Submodules: les dangers



Des pièges liés à l'utilisation de sous module peuvent arriver.

par exemple lorsqu'un développeur modifie un sous module et le prend en compte dans le Super Dépôt. Si il oublie de push le commit du sous module, les autres développeurs auront une erreur de type (fatal: reference isn't a tree:...) car le Super Dépôt pointera sur un commit non présent.

Submodules: Autres outils

```
git submodule foreach [cmd] // applique la commande aux sous modules
```

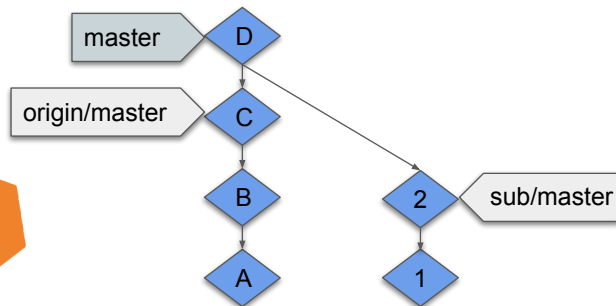
```
git config alias.sdiff "!git diff && git submodule foreach 'git diff'"  
git config alias.spush 'push --recurse-submodules=on-demand'  
git config alias.supdate 'submodule update --remote --merge'
```

alias

Des outils peuvent vous aider dans l'utilisation de Submodule:

- la commande `submodule foreach <cmd>`, permet d'appliquer la commande donné dans tous les sous modules. Par exemple pour lancer la compilation ou les tests u

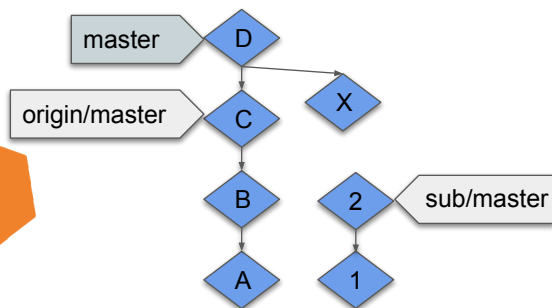
Projet multi-dépôt: Via subtree



Le fichier `.gitmodules` permet de lister les sous modules du dépôt principale aussi appelé "super dépôt"

Git détectera alors les évolutions des sous dossiers de super dépôt comme de nouvelles références aux sous modules

Subtree: Add subtree



1

```
git clone [main_repo]
cd main_repo
```

2

```
git remote add [remote_name] [url]
git fetch [remote_name]
```

3

```
git subtree add --prefix=[path] [remote] [ref] --squash
```

4

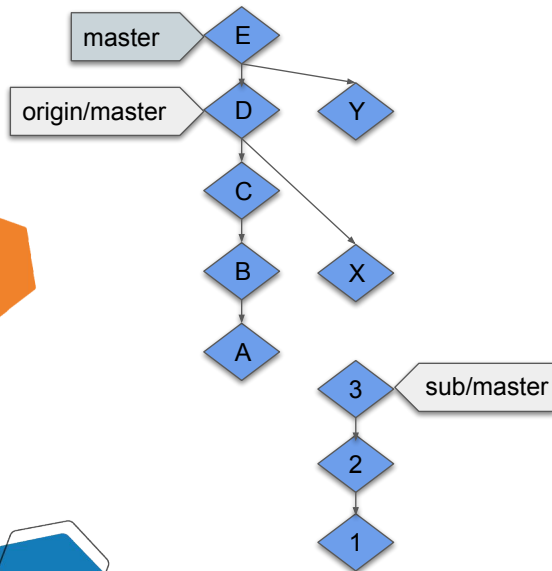
```
git push
```

Importer un dépôt externe dans un sous-dossier de ton projet **avec historique**, en une commande.

--squash (optionnel) : compresse tout l'historique en un seul commit. Si tu veux tout l'historique, enlève-le.

Git détectera alors les évolutions des sous dossiers de super dépôt comme de nouvelles références aux sous modules

Subtree: Update subtree



1

`git fetch [remote] [ref]`

2

`git subtree pull --prefix=[path] [remote] [ref] --squash`

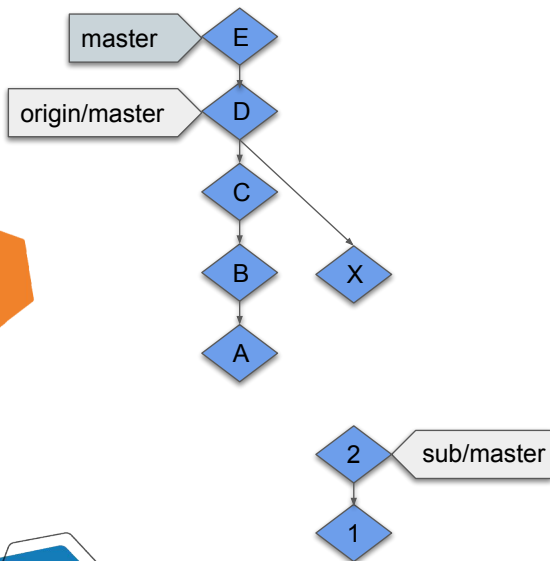
3

`git commit`
`git push`

Récupérer les **derniers changements du repo distant** dans ton projet principal.

Git détectera alors les évolutions des sous dossiers de super dépôt comme de nouvelles références aux sous modules

Subtree: Push subtree



1

git commit
git push

2

git subtree push --prefix=[path] [remote] [ref]

Exporter les modifs que tu as faites **localement** sur le code du subtree, puis **les pousser** vers le dépôt d'origine.

Git détectera alors les évolutions des sous dossiers de super dépôt comme de nouvelles références aux sous modules

multigit (IDEMIA)

```
{
  "description": "repo example",
  "fileFormatVersion": "1.0",
  "postCloneCommands": [],
  "repositories": [
    {
      "destination": ".",
      "head": "main",
      "head_type": "branch",
      "url": "https://github.com/bluebird75/multigit-playground.git"
    },
    {
      "destination": "src/extern/component1",
      "head": "VERSION_1",
      "head_type": "tag",
      "url": "https://github.com/bluebird75/multigit-playground.git"
    }
  ]
}
```

exemple.mgit

MultiGit OpenSource v1.6.1-rc1 - C:/d/work/multigit/playground/project1

File View Git Git Programs About

project1

Base directory : C:/d/work/multigit/playground/project1

Git Repo Path	Head	Status	Last Remote Synchro	SHA1
.	branch main	OK	Up-to-date	b58e9cb
src\extern\component1	tag VERSION_1	OK		ec2e13a
src\extern\component2	tag VERSION_2	OK		5194691
src\extern\component3	tag VERSION_3	OK		cb03c8e
src\modules\module1	branch main	OK	Up-to-date	b58e9cb
src\modules\module2	branch main	OK	Up-to-date	b58e9cb
src\modules\module3	branch main	OK	Up-to-date	b58e9cb
src\modules\module4	branch main	OK	Up-to-date	b58e9cb
src\modules\module5	branch main	OK	Up-to-date	b58e9cb
src\modules\module6	branch main	OK	Up-to-date	b58e9cb
src\modules\module7	branch main	OK	Up-to-date	b58e9cb

Multigit est un outil récent développé par Idemia (ex Safran Morpho). Il a la particularité d'avoir une interface graphique (python+QT)

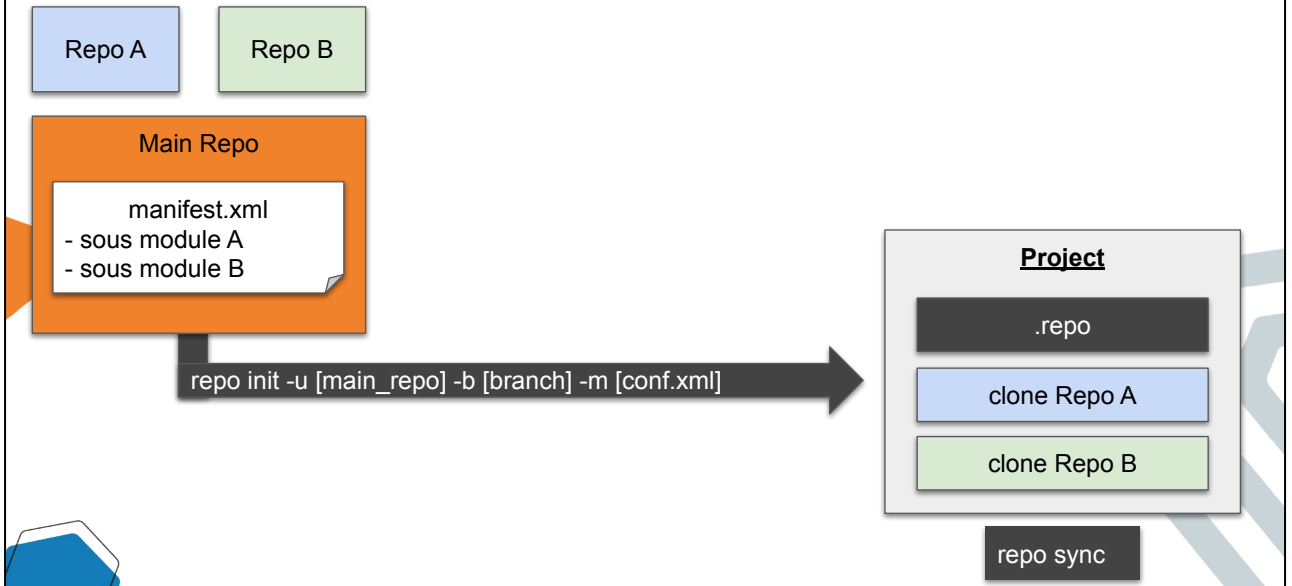
Pour lister les sous dépôts, leur destination et leur version, on utilise des fichiers de configuration avec l'extension .mgit.

Ces fichiers sont fortement inspiré de google repo

Il est possible dans l'interface graphique de sélectionner plusieurs dépôts pour faire des actions sur ces derniers (pull, push, branch, ...)

Il est également possible à tout moment de voir la liste des sous dépôts et leur version courante

Repo de google



Repo est un outil développé par google dans le but de gérer efficacement le projet Android et ses très nombreux dépôts. Repo se base sur Git et propose une surcouche aux différentes commandes.

Le principe de repo est le suivant:

- un fichier xml comporte la liste des dépôts et la révision à utiliser pour chacun. Ce fichier est en gconf également
- une commande repo "repo init" permet de récupérer la version souhaitée de ce fichier xml et le place dans un dossier en local du nom de .repo. Celui stocke alors les informations permettant le bon fonctionnement de repo et les .git de tous les sous dépôts.
- La commande "repo sync" permet alors d'utiliser les .git contenus dans le .repo pour peupler les WD des modules

Repo de google: installation



```
mkdir ~/bin
cd bin
git clone https://gerrit.googlesource.com/git-repo
chmod a+x ~/bin/git-repo/repo
export PATH=~/bin/git-repo/repo
repo init
```



```
cd c://programmes/repo
git clone https://gerrit.googlesource.com/git-repo

Ajouter à PATH le chemin "c://programmes/repo"

repo init
```

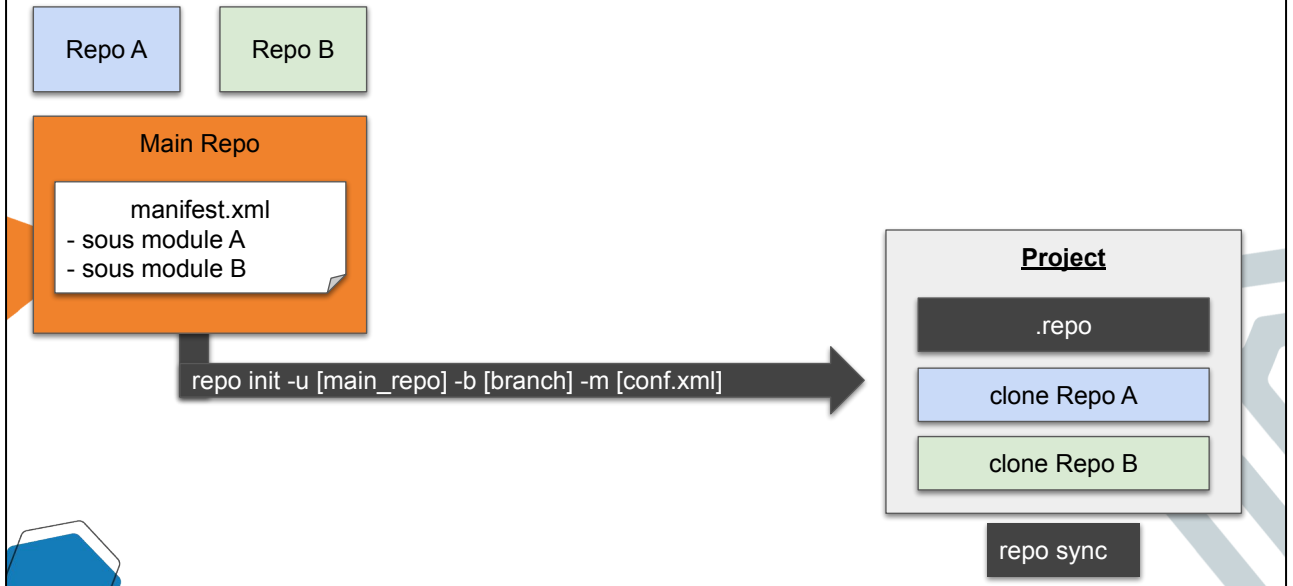


Doc: <https://source.android.com/setup/build/downloading#installing-repo>

Attention: il faut déjà avoir python d'installé sur sa machine

[http://www.yoannsculo.fr/git-repo-outil-gestion-multiples-repositories-arracher-cheveu
x/](http://www.yoannsculo.fr/git-repo-outil-gestion-multiples-repositories-arracher-cheveu-x/)
<https://gerrit.googlesource.com/git-repo/>

Repo de google: Initialisation



Le principe de repo est le suivant:

- un fichier xml comporte la liste des dépôts et la révision à utiliser pour chacun. Ce fichier est en gconf également
- une commande repo "repo init" permet de récupérer la version souhaitée de ce fichier xml et le place dans un dossier en local du nom de .repo. Celui stocke alors les informations permettant le bon fonctionnement de repo et les .git de tout les sous dépôts.
- La commande "repo sync" permet alors d'utiliser les .git contenus dans le .repo pour peupler les WD des modules

Repo de google: manifest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>

  <remote name="origin" fetch="ssh://projet/modules/" review="ssh://projet/modules/" />
  <remote name="kernel" fetch="ssh://linux/modules/" review="ssh://linux/modules/" />
  <default revision="master" remote="origin" />

  <project path="depot_A" name="depot_A.git" />
  <project path="lib/depot_B" name="depot_B.git" />
  <project path="os" name="os.git" remote="kernel" />

</manifest>
```

manifest.xml

```
ls -al .repo/
manifests
manifests.git
manifest.xml
project.list
project-objects
projects
repo
```

.repo/

repo init -u [main_repo] -b [branch] -m [conf.xml]

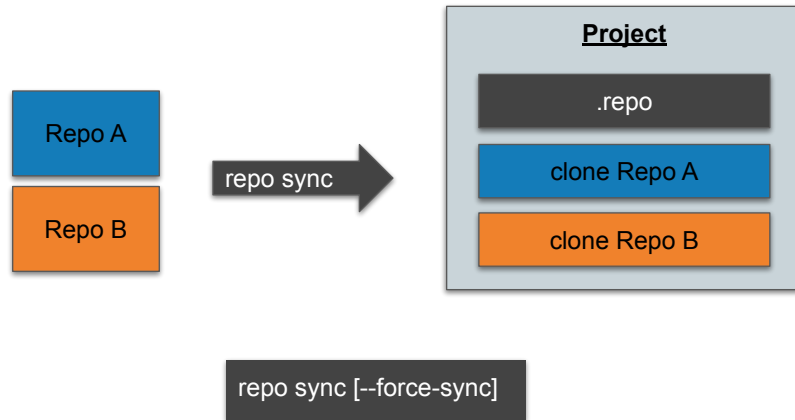
le fichier xml contient la liste des dépôt du projet ainsi que la révision à utiliser pour chacun.

Comme repo est un outil google, il s'interface très simplement avec Gerrit. En effet, il faut simplement indiquer le serveur "review"

Le fichier xml a également son dépôt Git pour suivre l'évolution du projet. On peut également avoir plusieurs .xml, un par besoins. Par exemple un fichier develop.xml pour récupérer les master des dépôts et un release.xml pour enregistrer les livraisons (versions pour chaque sous modules)

Une fois initialisé à partir de ce fichier xml, le projet en local comporte un dossier .repo. Celui-ci contient alors la copie du fichier xml

Repo de google: pull les sous repos



la commande "repo sync" permet de mettre à jour le dossier .repo et les .git qu'il contient. Puis mettre à jour éventuellement les WD des sous modules

Repo de google: travailler sur son projet



Pour réaliser des modifications dans un des sous modules, il faut réaliser 3 étapes:

- 1- utiliser la commande repo start: pour indiquer le début d'un nouveau développement. Repo vas alors automatiquement créer une branche de développement et vous positionner dessus.
- 2- faire vos modifications, vos commits et vos push avec git
- 3- utiliser la commande repo prune. Cette commande vérifie que la branche a bien été versé et si c'est le cas, la supprime

Note: vous pouvez aussi tilliser la commande "repo start" avec l'option --all pour créer la branche sur tous vos sous dépôts

Repo de google: autres outils



Source

<https://source.android.com/setup/develop/repo>

```
repo status // git status sur tous les repos
```

```
repo diff // git diff sur tous les repos
```

```
repo forall -c "commande" // "commande" sur tous les repos
```

Repo propose des commandes macro (qui s'appliquent sur l'ensemble des sous dépôts):

- repo status: fait un git status
- repo diff: fait un git diff
- repo forall -c "cmd": applique la commande cmd sur l'ensemble des sous dépôts

Gestionnaire de package



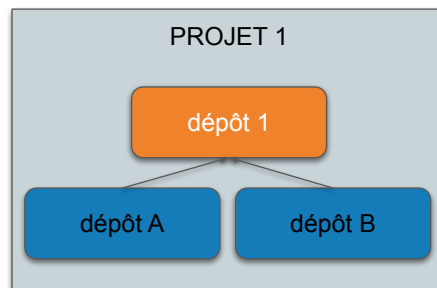
YARN



PYPI



CARGO



Vous pouvez aussi utiliser des gestionnaires de package pour gérer des sous projet a un projet principale

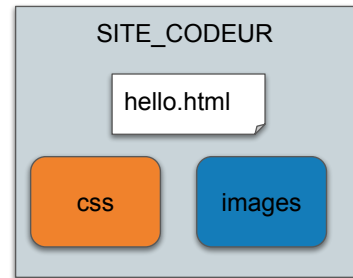
TP 13.1

Scénario

Vous souhaitez ne plus devoir cloner chaque dépôt de votre projet et gérer manuellement la compatibilité des version entre les différents modules qui le compose.

Le projet est composé comme suite:

- le dépôt SITE_CODEUR est le dépôt principale
- le dépôt css est un module commun
- le dépôt images est un module commun



Objectifs

- 1- Cloner le projet SERVEUR/SITE_CODEUR . Puis ajouter le submodule SERVEUR/TC_IMAGES sous images
- 2- Faire le commit de cet ajout
- 3- Faire une évolution dans hello.html pour ajouter la médaille d'argent puis faire le commit
- 4- Ajouter la médaille d'or dans le dépôt images
- 5- Mettre à jour le projet SITE_CODEUR pour prendre en compte l'évolution du module images.
- 6- Faire une évolution dans hello.html pour ajouter la médaille d'or et faire le commit
- 7- Cloner, dans un dossier CLONE_SITE_CODEUR, le projet pour vérifier que tout à été correctement push

Bilan

Méthode	Avantages	Inconvénients
A la main	<ul style="list-style-type: none">- Rien à installer ou configurer	<ul style="list-style-type: none">- Fastidieuse- Erreurs fréquentes
Scripts maison	<ul style="list-style-type: none">- Sure mesure	<ul style="list-style-type: none">- Temps de création- Maintenance
Submodules	<ul style="list-style-type: none">- Rien à installer ou configurer	<ul style="list-style-type: none">- Difficile à utiliser- Source d'erreur
SubTree	<ul style="list-style-type: none">- Rien à installer ou configurer	<ul style="list-style-type: none">- Difficile à utiliser- N'est pas un vrai sous module
MultiGit	<ul style="list-style-type: none">- Interface graphique	<ul style="list-style-type: none">- Outil à installer
Google repo	<ul style="list-style-type: none">- Simple et puissant	<ul style="list-style-type: none">- Il faut installer repo

Nous avons vu différentes méthodes pour gérer un projet multi dépôts.

Je vous recommande d'utiliser soit les gestionnaires de package ou submodule.
Petite astuce pour ce dernier, les interfaces graphique GIT modernes gères très bien git submodule. C'est le cas par exemple de vscode et gitlens