

# Formation Git

## XII - Personnalisation de Git



**Arnaud MERCIER**  
arnaud.mercier@hexotech.fr



Dans allons voir dans ce chapitre différents moyens de configurer git et notamment via les hooks

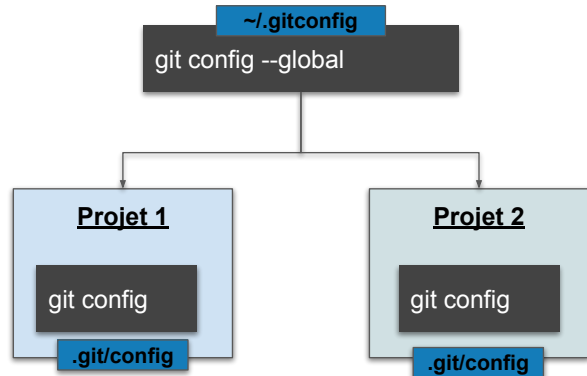
# Configuration de Git

```
git config user.name [nom_utilisateur]
git config user.email [email_utilisateur]
```

```
git config core.editor [editor_path]
git config merge.tool [tool_path]
```

```
git config commit.template [path_file]
```

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.last "log -1 HEAD"
git config --global alias.lg "log --oneline --graph --all --decorate"
```



Git peut être configuré à plusieurs niveaux. soit de manière global pour tous les dépôt sur votre machine soit de manière local, cad au niveau d'un dépôt donné.

les alias sont des raccourcis de commande que vous pouvez créer. Pour cela il faut utiliser la commande "git config alias." suivi du nom de votre alias puis entre simple quotes indiquer la commande qui doit remplacer l'alias lors de son utilisation.

Quand une configuration est définie de manière global et local, c'est la configuration local qui surcharge la global.

Par exemple si vous définissez "user.name" au niveau local à "John Snow" puis dans un projet A à "King of the north". A ce moment là, la configuration prise en compte dans le dépôt A est "King of the North" alors que dans le dépôt B elle sera "John Snow"

Les informations de nom et email, seront insérés automatiquement dans les métadatas de vos commits

# Ignorer des fichiers avec **.gitignore**

```
# pas de fichier .a
*.a

# mais suivre lib.a malgré la règle précédente
!lib.a

# ignorer uniquement le fichier TODO à la racine du projet
/TODO

# ignorer tous les fichiers dans le répertoire build
build/

# ignorer doc/notes.txt, mais pas doc/server/arch.txt
doc/*.txt

# ignorer tous les fichiers .txt sous le répertoire doc/
doc/**/*.*txt
```

**.gitignore**



**Le fichier .gitignore est un fichier caché**

**Le fichier .gitignore doit être ajouté à votre gestion de version**

Dans votre espace de travail, vous pouvez avoir du code source mais aussi d'autres fichiers comme par exemple des binaires générés après compilation ou encore des fichiers permettant de tester votre code (ex fiches clients pour un logiciel de gestion). Ces fichiers ne doivent pas se retrouver dans votre gestion de configuration. Le problème c'est que "git status" va les détecter comme de nouveaux fichiers.

Pour remédier à cela, il est possible d'ajouter à votre projet un fichier ".gitignore" qui va lister les fichiers et dossiers à ne pas prendre en compte.

Attention: ce fichier doit être ajouté à la gestion de version pour qu'il soit partagé avec les autres développeurs et pour suivre son évolution.

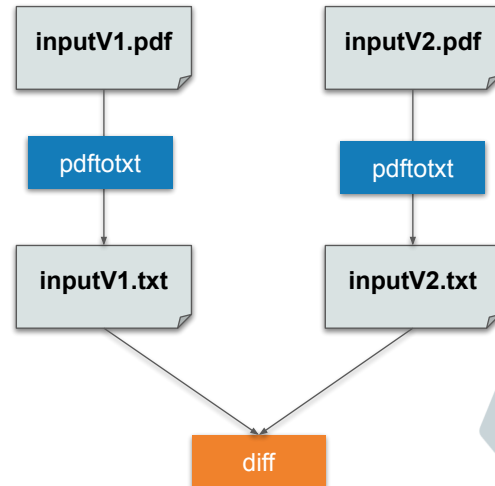
# Les attributs

```
git config diff.word.textconv docx2txt
```

```
git config diff.pdf.textconv pdftotxt
```

## **.gitattributes**

- \*.pbxproj binary
- \*.docx diff=word
- \*.pdf diff=pdf

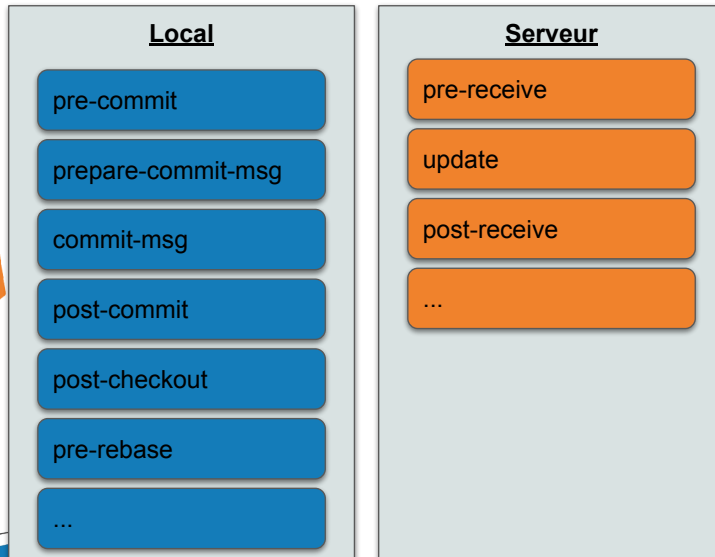


Les attributs sont comme des filtres. on indique dans un fichier “.gitattributes” la liste des règles. on indique le filtrage via un regex puis ce que l’on doit appliquer sur les fichiers correspondant au filtrage. on peut alors indiquer par exemple de traiter ces fichiers comme des binaire ou leur donner un outils qui permet de faire un diff sur ce type de fichier (non txt)

<http://docx2txt.sourceforge.net/>

<https://github.com/euske/pdftotxt/>

# Les Hooks



- Les hooks résident dans le dossier `.git/hooks`.
- Les hooks doivent être exécutables. `chmod +x <script>`
- Les hooks peuvent être écrit dans n'importe quel langage de script.
- Des exemples sont disponibles dans `.git/hooks` avec l'extension `.sample`
- Les hooks ne sont pas dupliqué lors d'un "git clone"

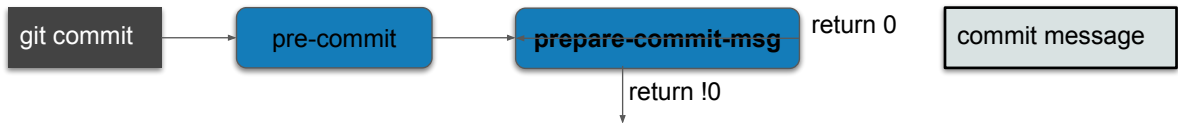
les hooks sont des script qui sont exécuté a des moments précis de l'utilisation de git (par exemple lors du commit). ces script retour 0 pour ok et un nombre différent de 0 pour ko.

## Hooks Locals: **pre-commit**



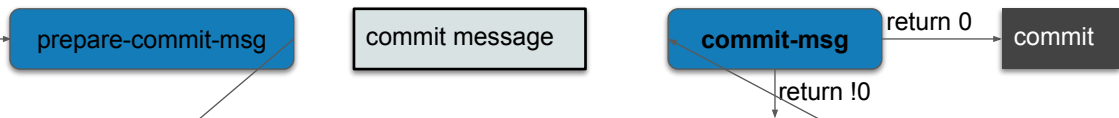
- Permet de réaliser des tests sur son code avant de faire le commit:
  - Si le retour de pre-commit est 0, alors on passe a la prochaine étapes du commit.
  - Sinon le commit est annulé
- Ce script ne prend pas d'arguments
- Peut être ignoré via l'option `-n` de la commande `git commit`

# Hooks Locals: **prepare-commit-msg**



- Permet de modifier le message de commit par défaut
  - Si le retour du script est 0, alors on passe à la prochaine étape du commit.
  - Sinon le commit est annulé
- Ce script comporte 3 arguments:
  - Le nom d'un fichier temporaire qui contient le message. Changez le message de commit en modifiant le fichier en place.
  - Le type de commit. Il peut s'agir de message (option -m ou -F), template (option -t), merge (si le commit est un commit de merge) ou squash (si le commit écrase d'autres).
  - L'empreinte SHA1 du commit concerné. Uniquement attribuée si l'option -c, -C ou --amend a été ajoutée.

## Hooks Locals: **commit-msg**



- Appelé après la saisie utilisateur du message de commit, il permet de vérifier le message de commit
  - Si le retour du script est 0, alors on passe à la prochaine étapes du commit.
  - Sinon le commit est annulé
- Ce script comporte 1 argument:
  - Le nom d'un fichier temporaire qui contient le message.



## Hooks Locals: **post-commit**



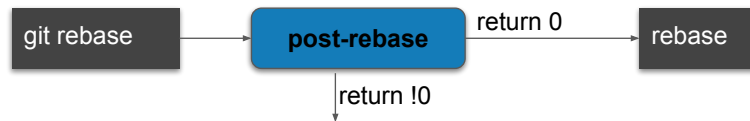
- Il ne peut modifier les conséquences de l'opération git commit, c'est pourquoi il est surtout utilisé à des fins de notification.
- Ce script prend un argument:
  - Le nom du fichier temporaire qui contient le message de commit.

# Hooks Locals: **post-checkout**



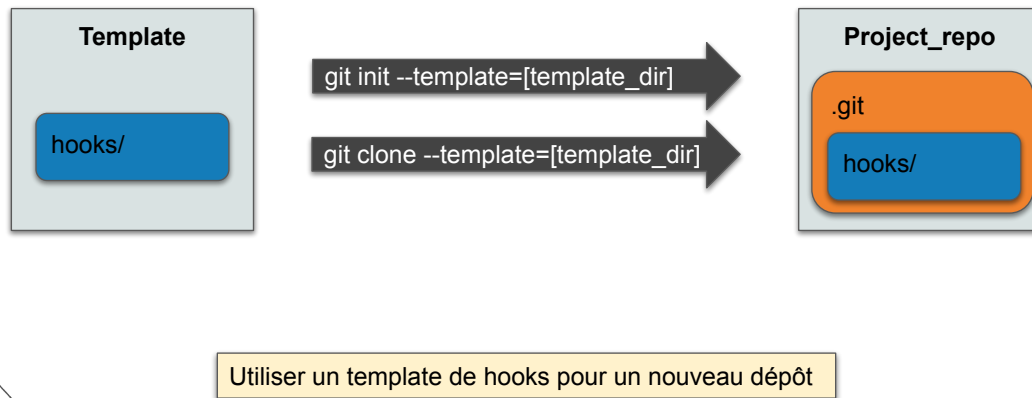
- Appelé immédiatement après un git checkout. Cette fonction est pratique pour supprimer des fichiers qui pourraient semer la confusion dans votre répertoire de travail. Ou encore pour modifier votre espace de travail en fonction de la branche cible.
- Ce script accepte 3 arguments:
  - Réf du HEAD précédent
  - Réf du nouveau HEAD
  - Cette option vous indique s'il s'agissait d'un checkout de branche ou de fichier. Les options seront respectivement 1 et 0.

## Hooks Locals: **post-rebase**



- Appelé avant que git rebase n'apporte le moindre changement. C'est le moment d'empêcher tout scénario catastrophe de se produire.
- Ce script nécessite deux paramètres:
  - La branche upstream à partir de laquelle la série a été forkée.
  - La branche rebasée. Si le rebase est lancé depuis cette dernière le paramètre est vide.

## Partage de hooks: **templates**



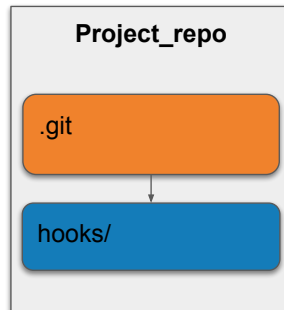
les hooks ne sont pas copiés automatiquement en local lors d'un "git clone" mais il existe des moyens de remédier à cela comme par exemple via les templates.

les templates correspondent à des dossiers qui contiennent le template de certaines parties d'un dépôt comme par exemple les hooks. Vous pouvez alors créer un dossier qui contient des hooks que vous souhaitez utiliser dans des dépôts. Lors de l'init ou du clone, il faut passer l'option "--template=" suivi du chemin vers le template.

A ce moment là, le contenu du dossier template sera copié dans le dossier .git/hooks de votre dépôt.

info: le template peut aussi bien être un simple dossier ou un dépôt

## Partage de hooks: **hooksPath**



```
git config core.hooksPath [hooks_path]
```

les hooks ne sont pas copié automatiquement en local lors d'un "git clone" mais il existe des moyens de remédier à cela comme par exemple via les projets partagés.

le principe est d'utiliser un dossier hooks autre que celui de votre dépôt et ainsi, par exemple, utiliser un dossier hooks partagé entre vos différents dépôts. Pour cela il faut modifier la configuration suivante "core.hooksPath <path\_hooks>".

info: le template peut aussi bien être un simple dossier ou un dépôt

## Partage de hooks: **Outils**



[Lefthook.dev](#)

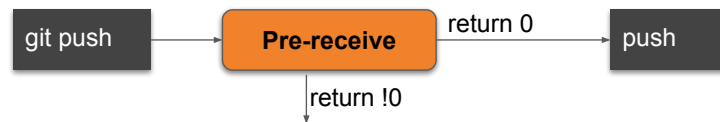
```
pre-commit:
  parallel: true
  jobs:
    - run: yarn run stylelint --fix {staged_files}
      glob: "*.css"
      stage_fixed: true

    - run: yarn run eslint --fix "{staged_files}"
      glob:
        - "*.ts"
        - "*.js"
        - "*.tsx"
        - "*.jsx"
      stage_fixed: true
```

[lefthook.yml](#)

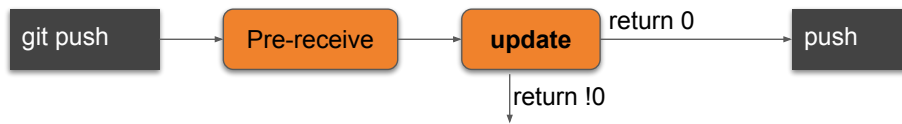
Il existe également des outils dédiés en fonction des technologies utilisées

# Hooks Serveurs: **Pre-receive**



- Appelé à chaque fois qu'un développeur utilise git push pour pusher des commits vers le dépôt.
- Si le retour du script est 0, alors le push est accepté, sinon il est refusé

# Hooks Serveurs: **update**



- Appelé après pre-receive et fonctionne pratiquement de la même manière.
- Si le retour du script est 0, alors le push est accepté, sinon il est refusé
- Ce script accepte 3 arguments:
  - Le nom de la réf mise à jour
  - Le nom de l'ancien objet stocké dans la réf
  - Le nom du nouvel objet stocké dans la réf.



# Hooks Serveurs: **post-receive**



- Le hook post-receive est appelé après un push réussi. C'est un emplacement idéal pour effectuer des notifications. Bon nombre de workflows préfèrent cet emplacement à post-commit, car les changements sont accessibles sur un serveur public et n'apparaissent pas uniquement sur l'ordinateur local de l'utilisateur. Le hook post-receive est souvent utilisé pour envoyer des e-mails à d'autres développeurs ou pour déclencher un système d'intégration continue.
- Ce script n'accepte pas d'arguments.

# TP 12.1

## Scénario

Il est temps pour vous de personnaliser un peu plus votre outil Git

## Objectifs

- 1- Modifier la configuration de Git (par exemple changer l'éditeur de texte)
- 2- Créer des alias pour vous faciliter la vie.  
(par exemple le git superlog = `git log --oneline --decorate --graph --all`)
- 3- Personnaliser l'affichage du git bash

Aide:

- 1- `git config core.editor <editor>`
- 2- `git config --global alias.slog 'git log --oneline --decorate --graph --all'`
- 3- `<git_install_dir>\etc\profile.d\git-prompt.sh`

## TP 12.2

### Scénario

Votre équipe souhaite automatiser certaines tâches en passant par exemple des testU sur le code lors de la mise en gestion de conf

### Objectifs

1- Mettre en place un hook qui permet de faire des testU automatique lors d'une tentative de commit. Si les testU échouent, empêcher le commit. Pour ce TP, le test consistera à vérifier que le dossier images est bien présent dans le WD.

2- Mettre en place un hook qui oblige d'avoir dans les messages de commit un numéro de fait technique sous la forme "**id: msg**"

Aide:

1-----

DIRECTORY=./images

```
echo "---- Start TEST-U"
if [ ! -d "$DIRECTORY" ]; then
    echo "> TEST-U failed: commit rejected"
    exit 1
fi
echo "> TEST-U ok"
```

2-----

```
echo "---- Check issue ID in $1"
test "" != "$(egrep '[0-9]+:' "$1")" || {
    echo "> No issue ID found".
    exit 1
}
echo "> issue ID = $(egrep -o '[0-9]+' "$1")"
```

SPOIL: commit-msg

## TP 12.3

### **Scénario**

Votre équipe souhaite mettre en place une sécurité sur le dépôt du projet via les hooks. l'objectif est alors d'éviter que les développeurs puissent supprimer un tag sur le dépôt distant.

### **Objectifs**

1- Mettre en place un hook dans le dépôt DISTANT qui permet de vérifier si le push correspond à une création de tag. Dans ce cas, vérifier que le tag possède bien une annotation, via `hooks.allowunannotated`. Si ce n'est pas le cas, refuser le push

2- Mettre en place un hook dans le dépôt DISTANT qui permet de vérifier si le push correspond à une suppression de tag. Dans ce cas, vérifier que la configuration `hooks.allowdeletetag` est à `true`. Dans le cas contraire, refuser le push.

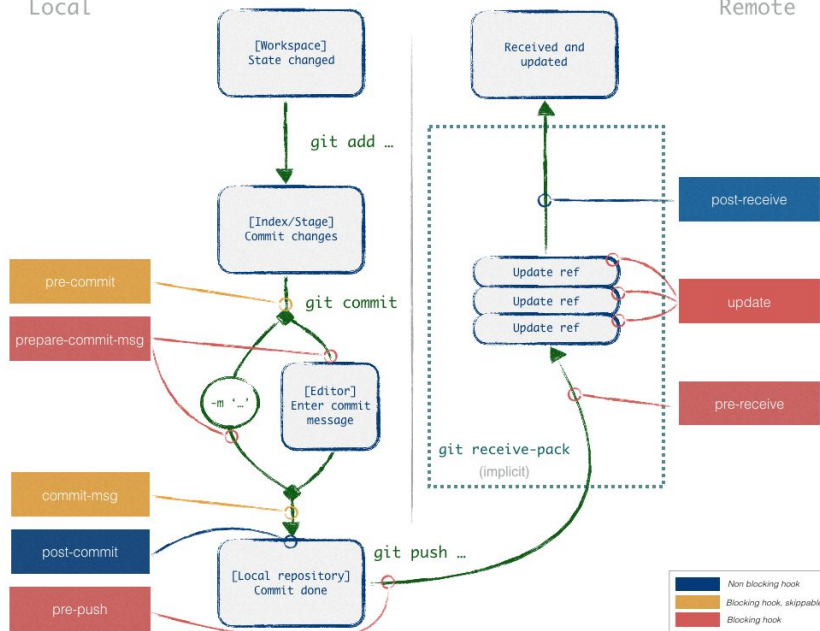
**Il faut ici partir du hook `update.sample`**

renommer le fichier `hooks/update.sample` en `hooks/update`

# Bilan

Local

Remote



Nous avons vu dans ce chapitre comment personnaliser Git via:

- sa configuration
- les alias
- les attributs

Nous avons également vu comment personnaliser le prompt et le terminal de git bash

Enfin nous nous avons utilisés les hooks dans nos dépôts Git pour exécuter des scripts automatiquement lors d'actions gits particulières.