

# Personnaliser Git – Rendre Git à ton image

Ynov Sophia Antipolis



## Introduction : Pourquoi personnaliser Git ?

Git, c'est un peu comme ton espace de travail personnel. Au début, tout est "par défaut". Mais plus tu l'utilises, plus tu veux qu'il te ressemble : des couleurs, des raccourcis, un style clair et des automatismes. Quand Linus Torvalds a créé Git, il l'a voulu libre et adaptable : chacun peut régler Git à sa façon, sans casser le système.

En clair : Git, c'est comme une voiture neuve. Tu peux changer le siège, le rétro et la radio sans toucher au moteur.

# Chapitre 1

## Git Configuration : personnaliser ton environnement

C'est quoi la configuration Git ?

Git garde ses réglages dans des fichiers qu'il lit à différents niveaux :

Niveau	Portée	Exemple de fichier
Système	pour tout l'ordinateur	/etc/gitconfig
Global	pour ton utilisateur	~/.gitconfig
Local	pour un projet précis	.git/config

**Note :** "Global" = pour toi sur ton PC, "Local" = juste pour le projet sur lequel tu travailles.

### Exemple 1 – Définir ton nom et ton email

```
git config --global user.name "billgates"
git config --global user.email "billgates@ynov.fr"
```

**Explication simple :**

- global = pour tous tes projets.
- user.name et user.email = les infos que Git mettra sur tes commits.
- Ces infos s'affichent sur GitHub quand tu pushes ton code.

**Pour vérifier :**

```
git config --list
```

Tu verras quelque chose comme :

```
user.name=Firas Bouricha
user.email=firas@ynov.fr
```

### Exemple 2 – Choisir ton éditeur préféré

```
git config --global core.editor "code --wait"
```

**Explication :**

- core.editor = l'éditeur que Git ouvrira quand tu écris un message de commit.
- Ici, "code --wait" veut dire : "ouvre Visual Studio Code et attends que je ferme le fichier avant de continuer".

**Comment tester :**

- Fais un commit sans message :
- git commit
- VS Code s'ouvre → tu écris ton message → tu sauvegardes → Git continue

### Exemple 3 – Activer les couleurs dans Git

```
git config --global color.ui auto
```

**Explication :**

- color.ui = dit à Git d'utiliser des couleurs pour les messages dans le terminal.
- auto = il colorie seulement quand c'est utile (ex. dans un vrai terminal, pas dans un script).

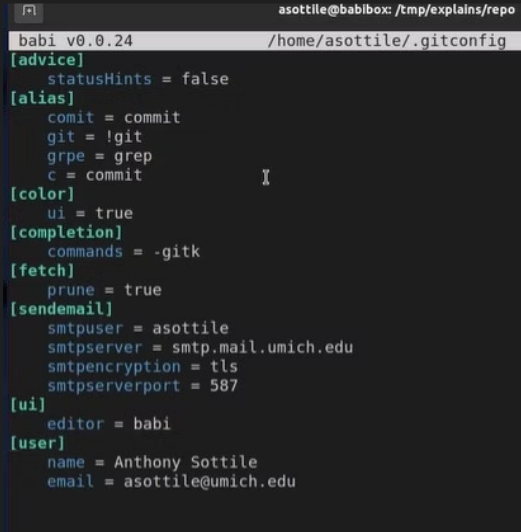
**Comment tester :**

Tape :

```
git status
```

Tu verras les fichiers en rouge (modifiés) et en vert (ajoutés). C'est bien plus lisible !

### Exemple 4 – Créer un raccourci (alias)



```
asottile@babibox: /tmp/explains/epo
babi v0.0.24 /home/asottile/.gitconfig
[advice]
  statusHints = false
[alias]
  commit = commit
  git = !git
  grpe = grep
  c = commit
[color]
  ui = true
[completion]
  commands = -gitk
[fetch]
  prune = true
[sendemail]
  smtpuser = asottile
  smtpserver = smtp.mail.umich.edu
  smtpencryption = tls
  smtpserverport = 587
[ui]
  editor = babi
[user]
  name = Anthony Sottile
  email = asottile@umich.edu
```

```
git config --global alias.st status
git config --global alias.lg "log --online --graph --decorate"
```

**Explication :**

- alias.st crée un raccourci pour git status.
- alias.lg crée un raccourci visuel pour l'historique des commits (en une ligne avec un graphique).

**Comment tester :**

Tape :

```
git st
git lg
```

Tu obtiens respectivement l'état du dépôt et un joli graphe de commits.

**Astuce :** ces alias te feront gagner du temps toute l'année.

# Chapitre 2

## Git Attributes : des règles par type de fichier

`.gitattributes` permet de dire à Git : "Traite ce type de fichier de telle manière." C'est pratique quand ton projet contient du code, des images, des PDF, etc.

### Exemple de fichier `.gitattributes`

```
# Forcer les fins de ligne LF sur les fichiers texte
*.txt text eol=lf

# Dire que les images sont binaires (pas de diff)
*.png binary

# Utiliser un diff spécial pour les fichiers Markdown
*.md diff=markdown
```

#### Explication :

- `text eol=lf` → évite les problèmes de retours à la ligne entre Windows et Linux.
- `binary` → empêche Git d'essayer de "comparer" des images.
- `diff=markdown` → affiche les différences de manière plus lisible pour les fichiers `.md`.

#### Comment tester :

1. Crée un fichier `.gitattributes` à la racine du projet.
2. Ajoute les lignes ci-dessus.
3. Fais un commit et observe : Git saura quel type de diff afficher selon l'extension.



# Chapitre 3

## Git Hooks : automatiser ton workflow

### 1. Introduction : pourquoi parler des "hooks" ?

Quand on débute avec Git, on apprend vite à faire des **commits**, des **branches**, des **merge**... Mais au fur et à mesure, on se rend compte d'une chose :

Les développeurs sont humains. Et les humains font des oublis.

Combien de fois as-tu déjà :

- oublié de lancer tes tests avant de pousser ton code ?
- fait un commit avec un message du style "test truc" ?
- laissé un `console.log()` ou un `fmt.Println()` dans le code avant de livrer ?

Ces petites erreurs sont normales, mais dans une vraie équipe, elles peuvent coûter du temps, voire casser une version. C'est là qu'interviennent les **Git Hooks**.

### 2. D'où vient l'idée des Hooks ?

Le mot **"hook"** signifie littéralement **"crochet"**. Dans l'univers Git, c'est une **porte d'entrée cachée** dans le système : tu peux y "accrocher" du code pour que Git  **fasse des actions automatiques à ta place**.

Historiquement, les hooks existent **depuis les toutes premières versions de Git (2005)**. À l'époque, Git a été conçu pour gérer le code du noyau Linux. Des centaines de développeurs y travaillaient en parallèle, et il fallait un moyen :

- de **vérifier la qualité du code** avant qu'il soit accepté,
- d'**imposer certaines conventions**,
- et d'**automatiser des tâches** sans dépendre de la mémoire humaine.

📌 **En résumé** : les hooks sont nés pour fiabiliser le travail d'équipe dans les gros projets open source.

### 3. C'est quoi concrètement un hook ?

Un **Git hook**, c'est un **script** (souvent en bash, Python ou Node.js) que Git exécute **automatiquement** avant ou après certaines actions.

Par exemple :

- juste **avant un commit** → pour vérifier ton code,
- juste **avant un push** → pour s'assurer que tout est propre,
- juste **après un merge** → pour mettre à jour ton environnement.

📌 Tu n'as plus besoin d'y penser : Git s'en occupe à ta place.

### 4. Où vivent les hooks ?

Chaque projet Git a un dossier caché nommé :

```
.git/hooks/
```

C'est là que vivent tous les scripts de hooks. Si tu fais :

```
ls .git/hooks
```

Tu verras des fichiers comme :

```
pre-commit.sample
commit-msg.sample
pre-push.sample
post-merge.sample
```

Les fichiers en `.sample` sont juste des **exemples** fournis par Git. Pour activer un hook :

1. Supprime le `.sample` à la fin du nom.
2. Ajoute ton propre code.
3. Rends le script exécutable :

```
chmod +x .git/hooks/pre-commit
```


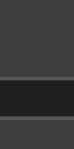



### 5. Pourquoi les hooks sont utiles ?

Les hooks sont là pour :

- **Automatiser** les vérifications (tests, syntaxe, conventions).
- **Éviter les erreurs humaines**.
- **Uniformiser le travail** entre les membres d'une équipe.
- **Gagner du temps** : plus besoin de tout faire à la main.

C'est une petite dose d'intelligence dans ton workflow. Git devient ton assistant personnel qui veille sur ton code.

### 6. Les hooks les plus courants

	<b>pre-commit</b> S'exécute <i>avant que le commit soit validé</i> . Sert à vérifier ton code ou bloquer un commit si quelque chose ne va pas (exemple : code non formaté, erreurs, fichiers interdits).
	<b>commit-msg</b> S'exécute <i>juste après que tu aies écrit le message de commit</i> . Permet de vérifier que ton message respecte un format précis (ex : "feat: ...", "fix: ..."), ou d'empêcher les messages vides.
	<b>pre-push</b> S'exécute <i>avant que ton code parte sur GitHub (ou un dépôt distant)</i> . Sert souvent à lancer les tests unitaires, ou à s'assurer que la branche est à jour avant d'être poussée.
	<b>post-merge</b> S'exécute <i>après une fusion réussie (merge)</i> . Utile pour relancer une installation (npm install ou go mod tidy) si des fichiers comme package.json ont changé.
	<b>post-checkout</b> S'exécute <i>après avoir changé de branche</i> . Peut servir à relancer des scripts, reconfigurer l'environnement, ou charger les bonnes variables selon la branche.

### 7. Exemple 1 — Un hook "pre-commit"

**But** : empêcher les commits si le code contient un "console.log".

```
#!/bin/bash

echo "Vérification du code avant commit..."

if grep -q "console.log" *.js; then
  echo "Enlève tes console.log avant de commit !"
  exit 1
fi

echo "Tout est propre, commit accepté."
```

Ce script est exécuté **automatiquement** à chaque `git commit`. Si une erreur est trouvée → le commit est bloqué.

C'est comme une barrière de sécurité automatique.

### 8. Exemple 2 — Un hook "commit-msg"

**But** : forcer les développeurs à écrire un message clair.

```
#!/bin/bash

MESSAGE=$(cat "$1")

if [[ ! "$MESSAGE" =~ ^feat|fix|docs|style|test: ]]; then
  echo "Message invalide : il doit commencer par 'feat:', 'fix:', etc."
  exit 1
fi
```

**Résultat** :

- "feat: ajout du bouton contact" → accepté
- "update truc" → refusé

Les hooks aident à garder un historique Git lisible et professionnel.

### 9. Exemple 3 — Un hook "pre-push"

**But** : vérifier que tous les tests passent avant d'envoyer sur GitHub.

```
#!/bin/bash

echo "Lancement des tests avant push..."

npm test

if [ $? -ne 0 ]; then
  echo "Tests échoués, push annulé."
  exit 1
fi
```

C'est le garde-fou ultime : tant que ton code n'est pas stable, il ne partira pas dans le dépôt distant.

### 10. Les hooks d'équipe (modernes)

Les hooks dans `.git/hooks` sont **locaux** : ils ne sont pas partagés quand tu pousses ton code sur GitHub.

Pour que toute l'équipe ait les mêmes vérifications et règles, on utilise des outils modernes comme **Husky** (JavaScript), **pre-commit** (Python) ou **Lefthook** (multi-langage). Ils permettent de lancer automatiquement les mêmes tests, vérifs ou conventions sur tous les postes.

📌 **À retenir** : Les *hooks* servent à automatiser et vérifier ton code. Ils sont locaux par défaut, mais grâce à Husky, pre-commit ou Lefthook, ils deviennent **collaboratifs et pro**.

Moderniser ses hooks, c'est garantir que toute l'équipe code avec les mêmes bonnes pratiques.

### 11. Une philosophie, pas juste un outil

Les Git Hooks ne sont pas seulement un truc technique. C'est une **façon de travailler mieux**, de faire confiance à Git pour gérer les détails, et de se concentrer sur l'essentiel : le **contenu du code**.

Git Hooks, c'est ton coéquipier silencieux. Il ne parle pas, mais il veille toujours sur ton dépôt.

# Chapitre 4

## Exemple : une politique d'équipe Git

Les hooks et configurations peuvent devenir des règles partagées pour tout le groupe.

**Exemple d'équipe** :

- Les messages de commit doivent commencer par feat: ou fix:
- Les tests doivent passer avant un push
- Les fichiers texte doivent tous être en LF

Ces règles peuvent être imposées avec un mélange de :

- `.gitconfig` partagé,
- `.gitattributes`,
- et `hooks/` personnalisés.

📌 **Résultat** : moins d'erreurs, un code plus propre, et des habitudes professionnelles.

### Chapitre 5 – En résumé

Élément	Rôle	Exemple
<code>.gitconfig</code>	Tes préférences Git	alias, nom, couleur, éditeur
<code>.gitattributes</code>	Règles par fichier	EOL, diff, merge
<code>hooks/</code>	Scripts automatiques	vérifier, tester, nettoyer

### Conclusion

Personnaliser Git, c'est comme personnaliser ton bureau ou ton IDE : plus tu le rends à ton goût, plus tu gagnes du temps. Git ne doit pas être une contrainte, mais un outil qui travaille avec toi.

Commence simple, teste tes réglages, et découvre ce qui te fait gagner en confort et en rigueur. Ton futur "toi" te remerciera.