Engineer

hook Git

⚓ script que Git exécute lorsqu'un événement survient

Ce que c’est

Un hook Git est un programme que Git lance lui-même à un moment précis de son travail : avant d’enregistrer un commit, une fois son message rédigé, avant qu’un push quitte la machine ou lorsqu’un serveur en reçoit un. Chaque hook est un fichier sans extension, nommé d’après ce moment, par exemple `pre-commit`, `commit-msg` ou `pre-push`. Sans configuration contraire, Git les cherche dans `.git/hooks`, à l’intérieur du dépôt ; le paramètre `core.hooksPath` permet de les placer ailleurs. Le fichier doit être exécutable, car Git ignore un fichier sans bit d’exécution, mais il peut être écrit dans n’importe quel langage que la machine sait exécuter.

Les hooks côté client s’exécutent dans le clone du développeur et se déclenchent lors d’opérations comme la création de commits et les fusions. Les hooks côté serveur s’exécutent sur la machine qui reçoit les pushs, telle qu’une forge, et se déclenchent à l’arrivée des commits poussés.

Ce qui transforme un hook de notification en garde-fou est son code de sortie. Le hook `pre-commit` ne reçoit aucun argument, et un code non nul annule le commit. Le hook `commit-msg` reçoit le chemin d’un fichier contenant le message proposé et peut le refuser ou le modifier. Le hook `pre-push` reçoit en arguments le nom et l’adresse du dépôt distant, et sur l’entrée standard une ligne par référence ; un code non nul arrête le push avant tout envoi. Sur le serveur, `pre-receive` s’exécute une fois par push, lit pour chaque référence l’ancien nom d’objet, le nouveau et le nom de la référence. S’il se termine avec un code non nul, aucune référence n’est mise à jour. Les hooks exécutés après l’événement, comme `post-commit`, servent à notifier et ne peuvent pas en changer le résultat.

Lorsque `git init` crée un dépôt, il remplit le répertoire des hooks de scripts d'exemple dont les noms se terminent par `.sample` . Git n'en exécute un qu'après son renommage sans ce suffixe.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un hook Git ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des changements testés, publiés et réversibles, pour livrer aux clients un mardi après-midi plutôt qu'un vendredi soir tard.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Déploiement SaaS GIS automatisé : GitHub Actions — déploiement, données et reporting automatisés

Déploiement SaaS GIS automatisé : GitHub Actions Contactez-nous →

Pourquoi c'est important

Un hook transforme un contrôle auquel il faut penser en un contrôle qui se produit tout seul, au moment où il coûte le moins cher. Un outil de formatage ou un détecteur de secrets qui s'exécute à la création du commit repère le problème alors que l'auteur du changement l'a encore en tête, plutôt que lors d'une revue plusieurs jours plus tard ou d'une exécution de CI après le partage du travail.

Les hooks font aussi d'une règle une propriété du dépôt plutôt que de la personne qui crée le commit. Une convention inscrite dans un guide de contribution ne tient que tant que les gens lisent le guide ; la même convention dans un hook `commit-msg` s'applique à chaque commit.

Les deux groupes diffèrent par les garanties qu'ils peuvent offrir. Le livre Pro Git rappelle que cloner un dépôt ne copie pas ses hooks côté client et recommande de faire respecter les règles côté serveur. Un hook côté serveur comme `pre-receive` s'applique à chaque push de chaque personne, quelle que soit son installation locale. Le coût est du temps : un hook s'exécute à chaque commit ou à chaque push, et un contrôle qui ralentit la création des commits incite à le contourner.

À quoi cela ressemble en pratique

Le dépôt de ce site installe ses hooks par le framework pre-commit, configuré dans `.pre-commit-config.yaml` et installé par `task setup` , qui exécute `pre-commit install --install-hooks` . Le fichier installe trois hooks Git. À l'étape pre-commit, il exécute Prettier, markdownlint, stylelint, yamllint, codespell, les contrôles de lint

et de formatage de Ruff, une détection de secrets par gitleaks, un contrôle des émojis et un contrôle du ton. À l'étape commit-msg, un script impose aux messages une forme conventionnelle sans émoji. À l'étape pre-push, il exécute tous les contrôles de qualité, `task check`, ce qui prend des minutes plutôt que des secondes. Le commentaire de ce hook en accepte le prix, car les pushs sont rares. L'en-tête du fichier interdit de contourner les hooks avec `--no-verify`, et le commentaire maintient cette position : un push qui ne peut pas attendre n'est pas prêt.

Un hook de message de commit impose aussi une norme d'historique commune à quatre dépôts : un préfixe et un périmètre conventionnels, une ligne d'objet de longueur inférieure à une limite fixée, aucun émoji ni mention d'attribution en fin de message. Le hook refuse tout ce qui n'est pas conforme dans chaque dépôt contenant du travail ; la norme appartient ainsi au dépôt, pas à la personne qui crée le commit.

Un hook côté serveur pour la forge de l'entreprise a fait l'objet d'un essai avant qu'un vrai push ne lui parvienne. Un tel hook s'exécute dans un dépôt nu sans arbre de travail, dans un environnement défini par la forge, et lit les références poussées sur l'entrée standard : l'exécuter à la main sur un ordinateur portable prouve donc peu de choses. Le test crée un dépôt nu, installe le hook, clone, crée des commits, pousse et vérifie les chemins d'acceptation et de refus. Il a révélé deux défauts invisibles à la lecture du fichier, tous deux décrits plus bas.

Sur un projet, le hook de commit et les contrôles de qualité exécutaient en fait des vérifications différentes, alors qu'un document affirmait que le hook exécutait tous ces contrôles. Un test détermine désormais les vérifications réellement invoquées par chacun et échoue lorsque les ensembles divergent. Les cinq qui ne s'exécutaient que dans le contrôle global étaient celles qui lisaient les textes du CV : le code était protégé à chaque commit et la rédaction seulement quand quelqu'un y pensait.

Un hook côté client n'a besoin d'aucun framework. Celui-ci, enregistré dans `.git/hooks/commit-msg` et rendu exécutable, refuse un commit dont l'objet ne commence pas par un type conventionnel :

```
#!/bin/sh
# git passes the path of the proposed commit message as $1
pattern='^(feat|fix|docs|chore)(\([a-z0-9-]+\))?: .+'
if head -n 1 "$1" | grep -Eq "$pattern"; then
```

```
    exit 0
fi
echo "commit-msg: start the subject with a type, as in 'fix(nav): ...'" >&2
exit 1
```

Où les problèmes surviennent

- **Un nouveau clone arrive sans les hooks.** Chaque clone doit les installer ; c'est le rôle de `task setup` ici. Un commentaire du `Taskfile.yml` du site décrit la version silencieuse de l'échec : sans `pre-commit` sur la machine, `task setup` n'installe rien, aucun hook ne s'exécute, sans avertissement, et le résultat ressemble exactement à celui d'un dépôt dont les contrôles réussissent.
- `--no-verify` **passe tout droit.** `git commit --no-verify` contourne les hooks `pre-commit` et `commit-msg`, `git push --no-verify` contourne `pre-push`, et définir `core.hooksPath` sur `/dev/null` désactive tous les hooks à la fois. Une règle qui doit s'appliquer doit aussi figurer dans un hook côté serveur.
- **L'environnement de Git suit le hook dans les autres dépôts.** Git exporte des variables comme `GIT_DIR` pour que les commandes Git d'un hook trouvent le bon dépôt. githooks(5) conseille de les supprimer lorsqu'un hook travaille dans un autre dépôt. Le test de la forge a rencontré précisément ce cas : le contrôle global extrayait la révision poussée et y exécutait les tests, qui héritaient du pointeur de Git vers le dépôt nu. Dix-neuf échouaient sur du code correct.
- **L'entrée standard transporte les cas limites.** L'autre défaut du test était un mécanisme de repli qui, faute de référence à traiter, utilisait un substitut que Git ne pouvait pas résoudre. Comme la valeur arrive sur l'entrée standard, l'échec apparaissait bien plus loin sous la forme d'une erreur sans rapport apparent. La page précise aussi que la forge recevra des mises à jour forcées et des suppressions de branches, que le test ne reproduit pas. Dans githooks(5), une référence supprimée arrive à `pre-push` sous la forme `(delete)`, avec un nom d'objet entièrement composé de zéros.
- **Un hook lent est contourné.** Intégrer les cinq contrôles de texte au hook de commit a nécessité d'en accélérer deux pour respecter le budget d'un hook pre-commit. La page précise que le temps d'exécution augmente avec le contenu et qu'à partir d'un certain point, un hook lent est contourné.
- **Le format passe et le sens reste sans contrôle.** Comme le dit la page consacrée aux quatre dépôts, le format est vérifié, pas le contenu. Un objet conforme qui ne décrit rien passe donc aussi bien qu'un objet qui explique le changement.

Pour aller plus loin

- Documentation de githooks — tous les hooks pris en charge par Git, leur moment d'exécution, leurs entrées et les décisions que détermine leur code de sortie.
- Pro Git : les hooks Git — les hooks côté client et côté serveur, leur installation et la raison pour laquelle un clone ne les emporte pas.
- Pro Git : exemple de règles imposées par Git — un hook `update` côté serveur qui impose un format de message et des règles d'accès, avec des hooks côté client qui avertissent avant un push.
- git config — le paramètre `core.hooksPath`, pour déplacer les hooks hors de `.git/hooks` ou tous les désactiver.
- git commit — l'option `--no-verify` et les deux hooks qu'elle contourne.
- git push — le hook `pre-push` et l'option `--no-verify` qui le contourne.



Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un hook Git ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/git-hook/