Introduction aux hooks

Temps de lecture : 4 minutes

Qu'est-ce qu'un `hook` ?

Un `hook` en programmation permet d'exécuter du code à un moment précis lorsque certains événements surviennent.

`Git` permet l'utilisation de `hooks` pour exécuter des `script` avant et après quasiment toutes les actions possibles.

Par exemple, grâce au `hooks` vous pouvez exécuter un `script` avant ou après un `commit`.

Ces `hooks` sont stockés dans le dossier caché de `Git` et plus précisément dans `.git/hooks`.

Par défaut, `Git` place quelques `scripts` mais qui ne sont pas activés :

```
ls .git/hooks
```

Donne :

```
applypatch-msg.sample
fsmonitor-watchman.sample
pre-applypatch.sample
prepare-commit-msg.sample
pre-rebase.sample
update.sample
commit-msg.sample
post-update.sample
pre-commit.sample
pre-push.sample
pre-receive.sample
```

Les `scripts` sont par défaut des `scripts shell` exécutables donc par `bash` par exemple.

Vous pouvez cependant les écrire dans n'importe quel langage, par exemple en `Node.js`.

L'extension `.sample` permet de ne pas les lancer par défaut. Il faut retirer l'extension pour les activer.

Exemple de `hook pre-commit`

Un `hook pre-commit` s'exécute avant d'effectuer le `commit`, avant même l'ouverture de l'éditeur de texte pour le message de validation si vous n'utilisez pas l'option `-m`.

Si vous ne retournez pas `null` ou `0` dans le `script` exécuté dans ce `hook`, le `commit` sera annulé.

Voici un exemple de `hook pre-commit` que nous utilisons chez `Dyma`.

Son objectif est d'empêcher d'effectuer un `commit` contenant un `console.log` :

```
#!/bin/bash

exec < /dev/tty
if test $(git diff --cached | grep ^+.*console.log | wc -l) != 0
then
    read -p "Au moins un console.log est ajouté dans votre commit ! Ê
tes-vous certain de continuer ? [o/n]: " choix
    if [[ $choix == 'o' || $choix == 'O' ]]
    then
        exit 0;
    else
        exit 1;
    fi
fi
```

Il est utile pour ne pas se retrouver en production avec des `console.log` utilisée parfois en développement.

Vous pourrez retrouver notre cours dédié sur les `shells` mais voici quelques explications.

`#!/bin/bash` permet de dire que c'est un `script shell` devant être exécuté avec `bash`.

`exec < /dev/tty` permet d'activer la possibilité pour les utilisateurs de rentrer du texte.

`test` permet de tester une expression et de renvoyer `0` si elle est vraie et `1` si elle est fausse.

`git diff --cached` permet de lister les changements qui sont indexés par rapport au dernier `commit`.

`|` est un `pipe` : c'est-à-dire qu'il permet de rediriger la sortie de l'expression précédente en entrée de l'expression suivante.

`grep` permet de filtrer. Ici nous utilisons une expression régulière (`^+.*console\.log`) qui signifie : commence par un `+` (pour un ajout dans l'index) suivi de tout caractère un nombre indéfini de fois, suivi de `console.log`.

`wc -l` permet d'afficher le nombre de lignes d'une expression.

`if test $(git diff --cached | grep ^+.*console\.log | wc -l) != 0` signifie donc : si il y a au moins une ligne qui ajoute un `console.log` et qui est indexée alors passe au `then`.

`read -p` permet d'afficher du texte et de permettre à l'utilisateur de rentrer du texte directement après celui-ci.

`"Au moins un console.log est ajouté dans votre commit ! Êtes-vous certain de continuer ? [o/n]: " choix` : permet d'afficher la phrase et de stocker ce que rentre l'utilisateur dans la variable `choix`.

`if [[ $choix == 'o' || $choix == 'O' ]]` : Les doubles crochets permettent l'utilisation de plusieurs conditions en `bash`. Ici, si la variable `choix` vaut `o` ou `O` nous retournons `0` sinon `1`.

Si nous retournons `0`, le `commit` est effectué, si nous retournons `1`, le `commit` n'est pas fait.

Nous verrons dans la leçon suivante qu'il existe des librairies facilitant l'utilisation des `hooks`.