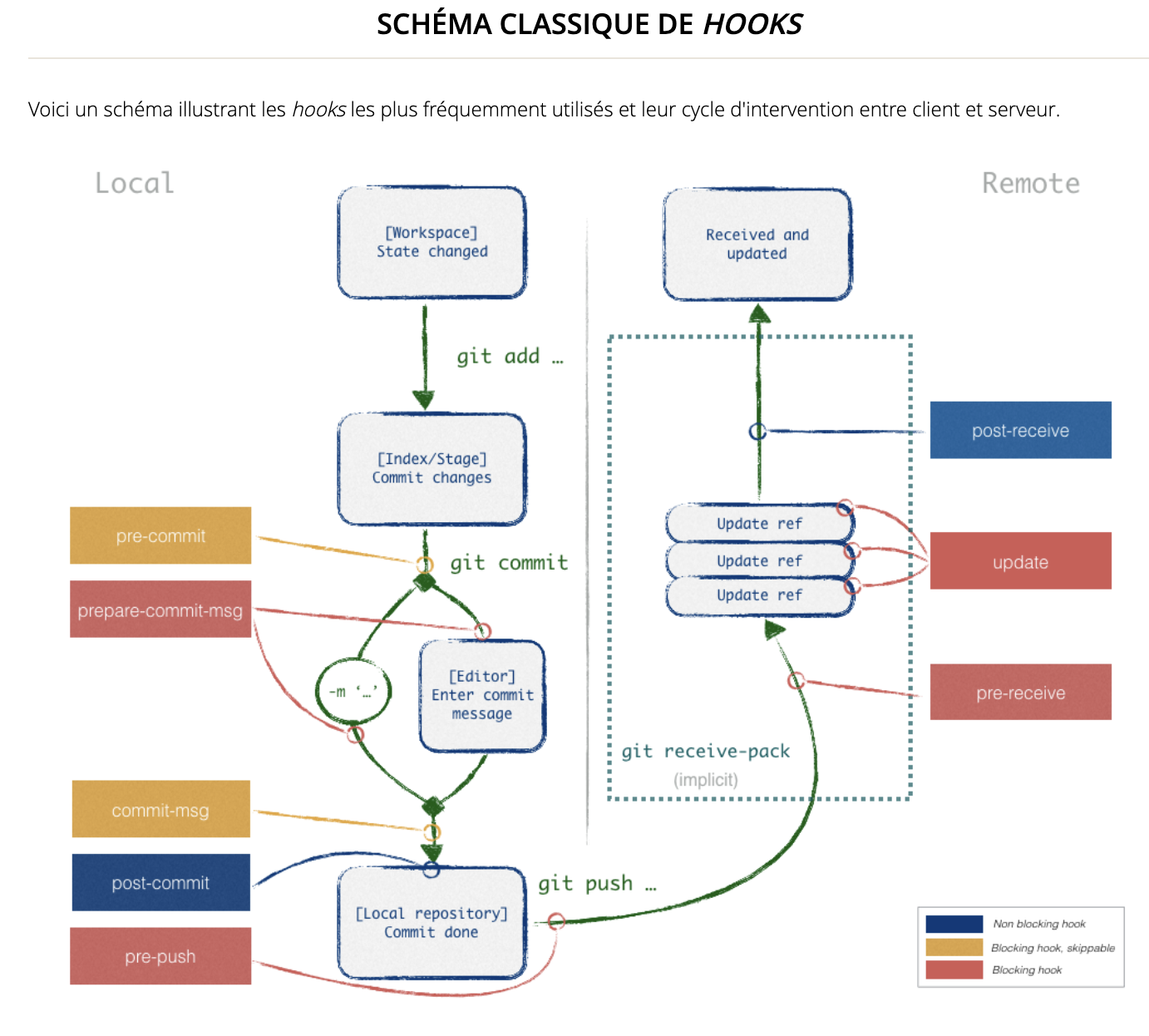
# Git Hooks intro :

* Permet d’automatiser des actions grâce à des scripts
* Utilisable coté serveur (serveur Git) et en local (client)
* Dossier : .git/hooks (dans le dossier du projet Git local)
* Scripts langage : Python, Ruby …
* Empaqueter scripts : .sample
* Rendre script fonctionnel : placer script dans dossier hooks, sans ext

# Schéma classique de Hooks :



# Initialisation du dépôt via un gabarit de projet :

Lorsque vous initialisez votre dépôt projet, vous pouvez dire à Git d’utiliser un gabarit d'arborescence de projet type grâce à l'option --template=<template directory> :

* Lors de la création du projet : git init --template=<template directory>
* Lors du *clone* : git clone --template=<template directory>

Cette approche vous permet de définir plusieurs gabarits et de charger celui qui vous intéresse à l’initialisation ou à la récupération d’un projet.

# Hooks disponibles coté client :

* Autour des commits :
  + pre-commit : avant la création du commit, avant même l’édition du message (ex. : linting, tests unitaires courts) ;
  + prepare-commit-msg : avant la création du commit, au moment où l’on va avoir la main sur l’édition du message (ex. : permet l’injection d’un message pré-calculé) ;
  + commit-msg : juste avant la création du commit, mais après l’édition du message (ex. : contrôle du contenu du message et réécriture à la volée) ;
  + post-commit : une fois le commit créé (ex. : notification) ;
* Autour de l’application d’un patch (*hooks* lancés autour de la commande git am) :
  + applypatch-msg : avant l’application du patch (vérification du message de patch) ;
  + pre-applypatch : après l’application du patch, mais avant la création du commit (ex. : validation du contenu du patch appliqué) ;
  + post-applypatch : une fois le patch appliqué et le commit créé (ex. : notification de l’auteur du patch) ;
* Autres opérations :
  + pre-rebase : avant le démarrage de la commande git rebase (ex. : interdire le *rebase* de la branche *master* ou de commits déjà poussés sur le serveur) ;
  + post-checkout : après l’exécution d’un git checkout et ses appels implicites, par exemple lors d'un rebase ou à la fin d’un git clone (ex. : paramétrage de l’environnement de travail associé, énoncé de la branche lors du déplacement sur une branche) ;
  + post-merge : après l’exécuton réussie de git merge (ex. : vérification de la présence de marqueurs de conflits suite à une mauvaise fusion) ;
  + post-rewrite : invoqué par des commande de "ré-écriture" des révisions (git commit --amend, git rebase) ;
  + pre-auto-gc : lors de l’appel automatique au garbage collector (permet donc d’intervenir avant la suppression de certains objets et certaines références) ;
  + pre-push : intervient juste avant l’envoi de nouvelles révisions et/ou objets vers un serveur (ex. : lancement des tests unitaires et validation/invalidation de l’envoi).

# Hooks disponibles coté serveur :

* pre-receive : avant réception des références/objets (ex. : vérification des droits de l’utilisateurs sur le projet) ;
* update : avant réception des références/objets par branche (ex. : vérification des droits par branche) ;
* post-update : après réception des références par branche (ex. : notification d’utilisateurs tiers pour effectuer une revue de code) ;
* post-receive : après réception de l’ensemble des références (ex. : lancement de l’intégration continues/tests d’intégration, déploiement automatique).

Préférez post-receive, plus complet que post-update : il possède les anciennes références et options liées au push (notamment utilisé chez GitHub pour les notifications).

Une image contenant texte

Description générée automatiquement

Figure , création script vérification d'email client pre-push

Une image contenant texte

Description générée automatiquement

Figure , résultat du script au moment du push, fonctionne