

Warning

- Ce guide suppose une connaissance **de l'outil GIT ([git.html#outils-internes-git](#))**, qui s'avère nécessaire pour certaines étapes. Vous en serez notifié par une bulle d'information.

Pour versionner le code déployé chez nos clients, nous utilisons Gitlab. Gitlab est l'équivalent de Github, mais totalement Open Source, et surtout il peut être installé sur un serveur nous appartenant (On-Premise), ce qui est le cas aujourd'hui.

Petit tour d'utilisation de Gitlab.

1. Présentation générale

Vous venez de vous connecter à Gitlab, mais ne l'avez encore jamais utilisé ?

On va commencer ici alors !

Si par ailleurs, vous ne connaissez pas Git, l'outil principal de versionning que nous utilisons avec Gitlab, il est conseillé de passer en premier lieu par la documentation suivante : Les bases de Git ([git.html#outils-internes-git](#))

1.1 Débuter sur Gitlab

Lorsque vous vous connectez pour la première fois sur Gitlab, une première étape importante est nécessaire, à savoir **l'ajout de votre clé SSH publique dans vos réglages Gitlab**, pour pouvoir récupérer les dépôts clients présents dans Gitlab, sans quoi cela vous sera impossible.

Vous arrivez en premier lieu sur cette page :

Projects

Groups

Activity

Milestones

Snippets

Search or jump to...

On retrouve ici tous les projets clients actuellement gérés par Gitlab, sur lesquels Subteno IT travaille. Vous retrouverez à la fois les **dépôts clients**, ainsi que **vos propres dépôts distants**, lorsque vous aurez **forké** votre premier projet.

Tip

Un fork est une copie d'un dépôt client, dans un espace sur Gitlab qui est le vôtre. Lorsque l'on dit que l'on forke un dépôt, en somme, on le copie ... simplement.

Passons maintenant à la première configuration de votre compte.

1.2 Ajouter sa clé SSH

Pour pouvoir utiliser l'outil GIT (<git.html#outils-interne-git>) , il est nécessaire d'importer sa clé SSH publique dans Gitlab, afin qu'il soit capable d'identifier si la personne essayant de cloner (récupérer un dépôt) est vraiment celle attendue (Vous en locurrence).

Pour cela, rendez-vous en haut à droite, **settings**, puis dans le menu de gauche, **SSH Keys**.

User Settings → SSH Keys

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

To add an SSH key you need to [generate one](#) or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/id_rsa.pub' and begins with 'ssh-rsa'. Don't use your private SSH key.

Typically starts with "ssh-rsa ..."

Title

e.g. My MacBook key

Name your individual key via a title

Add key

Ici, tout est expliqué. Vous devez récupérer votre clé SSH publique qui se situe sur votre mac. Pour cela ouvrez un terminal, et utilisez la commande suivante pour afficher votre clé SSH :

```
cat ~/.ssh/id_rsa.pub
```

Copiez ce qui est alors affiché, et collez le dans Gitlab. Nommez la clé et validez. Votre clé est désormais bien enregistrée dans Gitlab, et vous pouvez désormais cloner les dépôts clients, ou surtout, vos forks de ces dépôts !

Warning

- Votre clé SSH privée ne doit JAMAIS sortir de votre mac. Sous aucune raison. Sinon,

n'importe qui sera en capacité d'utiliser votre identité.

- Si vous changez votre clé SSH, il vous faudra alors la mettre à jour sur tous les outils

utilisant le protocole SSH : Serveurs clients, Gitlab, Github, Gitolite ... etc. Un conseil, ne la changez pas.

1.3 Interface de travail

Gitlab vous proposer une interface de travail simple, pour suivre les différents éléments sur lesquels vous agissez. Dans la **top navbar**, vous pouvez notamment retrouver les issues, merges-requests, ou encore les tâches auxquelles vous êtes lié :



Pour voir comment tout fonctionne, nous allons commencer par forker notre premier projet, le récupérer en local, puis effectuer quelques modifications dessus.

Tip

A partir d'ici, il est recommandé, voire nécessaire de connaître l'outil GIT. Sa documentation est disponible à **cet endroit** (git.html#outils-internes-git)

2. Utilisation de Gitlab

Pour voir comment tout fonctionne, nous allons commencer par forker notre premier projet, le récupérer en local, puis effectuer quelques modifications dessus.

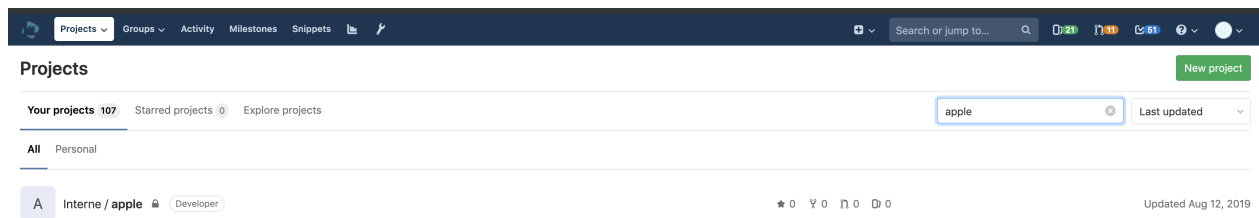
2.1 Récupération d'un projet Gitlab

Tip

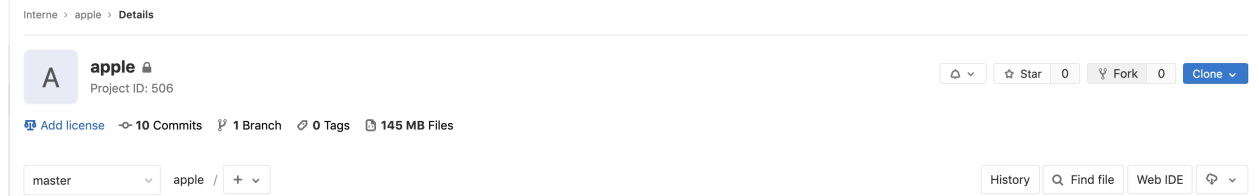
La procédure vue ici s'applique à TOUS les projets clients.

Pour récupérer un projet, rendez-vous sur le projet **Apple**, que nous allons prendre en exemple pendant toute la durée de cette documentation. Ce projet est destiné à vous entraîner, et contient un dépôt Odoo **standard**, en version 12.

Rendez-vous sur le dépôt Gitlab, en le recherchant via la search bar :



Puis forcez le projet afin d'en créer une copie vous appartenant :



Choisissez l'espace qui vous appartient, avec votre nom et prénom.

Gitlab prends un peu de temps pour faire une copie. Une fois la copie réalisée, vous vous retrouverez alors sur **votre fork** gitlab, ce qui vous permettra ainsi de travailler.

Tip

Nous ne travaillerons jamais sur le dépôt client directement. Gitlab est fait de sorte à ce que l'on puisse plutôt proposer des modifications, qui font l'objet d'une phase de validation avant mise en production. D'autre part, cela nous permet de mettre en place un processus de CI (Intégration Continue) qui effectue de nombreuses vérifications pour nous, notamment pour valider la non régression du dépôt.

Clonez ensuite le dépôt en local sur votre mac. Pour cela, cliquez sur **clone**, puis copiez l'URL SSH de votre fork.

Rendez-vous ensuite dans le répertoire dédié aux projets clients sur votre MAC :

```
cd ~/Projects/Clients
```

Puis clonez le dépôt via :

```
git clone <url_ssh_de_votre_fork>
```

Vous venez de récupérer une copie locale de votre dépôt :)

2.2 Première configuration d'un dépôt local

Vous venez de récupérer votre dépôt. Il faut désormais configurer deux choses afin d'avoir un fonctionnement optimal, à savoir :

- Configurer le remote de référence, qui est le dépôt client. On veut se tenir à jour par rapport à ce dépôt.
- Configurer la branche à suivre du remote de référence, depuis notre branche locale **master**

Pour ce faire, il faut configurer sur votre dépôt local un nouveau remote, qui se nommera subteno (ou client si vous préférez), pointant vers l'url SSH du dépôt de référence :

```
git remote add -f subteno git@gitlab.subteno-it.net:interne/apple.git
```

Enfin, comme nous commencerons toujours à travailler depuis notre branche locale **master**, il faut que cette branche **suive** la branche master du dépôt de référence :

```
git branch -u subteno/master master
```

Vous êtes désormais prêts à travailler !

Pour ceux qui préfèrent le format vidéo, voici un récapitulatif de ce qui doit être fait lors de la récupération d'un dépôt client (Ici, nous récupérerons le projet "REMADE_TECH") :

```
bash-5.0$ source ~/.bash_profile
vincentcoffin@MBP-de-Vincent:~$ cd /
vincentcoffin@MBP-de-Vincent:~/Projects/Clients$ Je récupère mon fork !^C
vincentcoffin@MBP-de-Vincent:~/Projects/Clients$ git clone git@gitlab.subteno-it.net:vco-subteno-it/remade_tech.git
Cloning into 'remade_tech'...
remote: Enumerating objects: 43338, done.
remote: Counting objects: 100% (43338/43338), done.
remote: Compressing objects: 100% (27759/27759), done.
remote: Total 43338 (delta 14320), reused 43338 (delta 14320)
Receiving objects: 100% (43338/43338), 172.79 MiB | 15.81 MiB/s, done.
Resolving deltas: 100% (14320/14320), done.
Checking out files: 100% (38257/38257), done.
vincentcoffin@MBP-de-Vincent:~/Projects/Clients$ cd remade_tech/
(master) vincentcoffin@MBP-de-Vincent:~/Projects/Clients/remade_tech$ je rajoute e
```



(<https://asciinema.org/a/f41pisKCrolfGfTGHFwCDaJGL>)

2.3 Envoi d'une Merge Request (Proposer une modification)

Vous avez réalisé une modification en local, et souhaitez proposer une modification du dépôt client (suivant une tâche précédemment assignée).

Tip

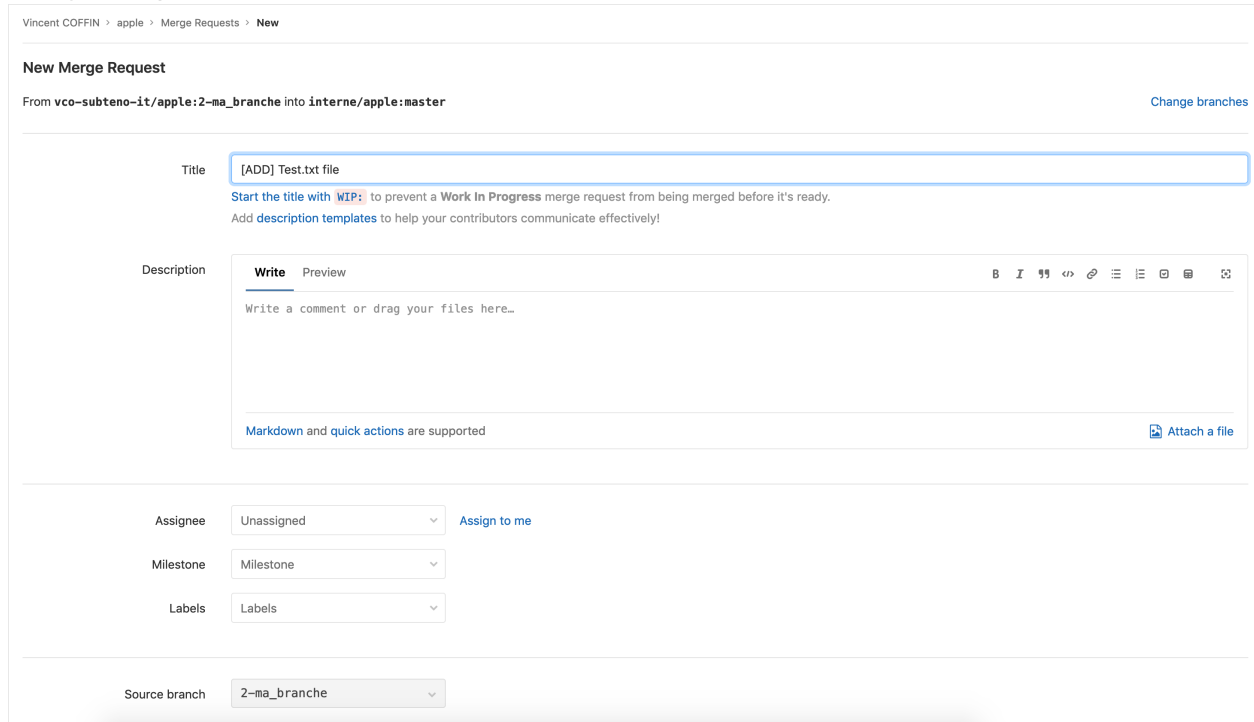
Pour proposer vos modifications, assurez vous d'avoir créé une nouvelle branche en local, nommée de la forme **<N°_issue>-<nom_branche>**. Ce process est établi dans nos processus internes de développement, et est disponible **à cet endroit** (**./fonctionnement_interne/processus_de_developpement.html#fonctionnement-interne-processus-de-developpement**) On ne développe pas sur la branche **master**, elle permet juste de se tenir à jour vis à vis du dépôt de référence.

On pousse alors nos modifications sur notre dépôt distant, sur la nouvelle branche sur laquelle nous travaillons. Gitlab va la créer pour nous sur dépôt distant :

```
{2-ma_branche} vincentcoffin@MBP-de-Vincent:~/Projects/Clients/apple$ git push origin 2-ma_branche
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 286 bytes | 286.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote:
remote: To create a merge request for 2-ma_branche, visit:
remote:   https://gitlab.subteno-it.net/vco-subteno-it/apple/merge_requests/new?merge_request%5Bsource_branch%5D=2-ma_branche
remote:
To gitlab.subteno-it.net:vco-subteno-it/apple.git
 * [new branch]      2-ma_branche -> 2-ma_branche
{2-ma_branche} vincentcoffin@MBP-de-Vincent:~/Projects/Clients/apple$
```

Dans le retour de Gitlab, on voit qu'il nous propose d'accéder à un lien. Ce lien nous permet de faire une proposition de modification, si on le consulte.

Via **Visual Studio Code**, vous avez la possibilité de faire **Cmd+Click** afin d'ouvrir votre navigateur par défaut à cet URL. Voici où l'on arrive :



Plusieurs points importants à cet endroit :

- Sous le titre que vous donnez à la MR (Merge Request), vous avez la possibilité de cliquer sur un lien, qui vous rajoute automatiquement le préfixe **WIP:** devant votre titre. Cette fonctionnalité permet de spécifier que le travail est toujours en cours, et donc que la MR **ne doit pas être mergée en l'état**. Gitlab bloque de toute façon toute tentative de Merge tant que ce statut n'est pas résolu par la suite.
- Vous pouvez ajouter une description pour décrire la modification que vous apportez.
- Une personne peut-être assignée à la MR. Cette personne devient alors celle devant valider les modifications réalisées, afin de vérifier que tout fonctionne. Selon le code envoyé, vous pouvez vous permettre de vous assigner vous même, mais en cas de grosse modification, il est fortement conseillé de passer par une autre personne, qui validera avec vous le code pour être certain que tout est bon (Quality Analysis).
- Enfin, la **target branch** est la branche ciblée dans le dépôt de référence (dépôt client). Vous choisissez ici à quel endroit votre code est envoyé. **ATTENTION:** Par défaut, Gitlab cherchera à merger dans Master. Si vous apporter un développement pour une version d'odoo supérieure, notamment dans le cadre d'une migration, **pensez à changer la branche cible**.

Une fois les vérifications faites, et que tout vous semble correct, vous pouvez cliquer sur le bouton **Submit merge request**.

Une fois la MR envoyée, elle est disponible dans la section **Merge request** du dépôt de référence (client), et un processus d'intégration continue se met en place.

2.4 Intégration continue (CI / CD)

Vous avez envoyé votre première MR. Vous ne l'avez peut-être pas remarqué, mais un processus bien spécifique s'est mis en marche à ce moment là : la CI / CD (Continuous Integration / Continuous Development).

Ce processus est important, pour diverses raisons :

- Il vérifie la qualité du code envoyé. On respecte les normes en place pour l'écriture du code, ce qui le rend plus facilement maintenable.
- Il effectue les tests unitaires, qui vérifient qu'aucune régression n'est amenée via la MR proposée.
- Il certifie ainsi le bon fonctionnement de ce qui est envoyé. (Ou tout du moins, qu'aucun erreur de compilation n'est rencontrée).

Ce fonctionnement est visible à cet endroit, sur la MR que vous avez créé :

The screenshot displays a GitLab Merge Request (MR) interface. At the top, a green checkmark icon indicates that the pipeline is successful. The text reads: "Pipeline #17323 passed for 7872ca2b on vco-subteno-it:2-ma_branche". To the right of this text are two green circular icons: one with a checkmark and another with a double arrow. Below this, a section shows deployment status: "Deployed to coverage/2-ma_branche 3 hours ago" with a "View app" button and a small square icon. Below that, two rows show future deployment plans: "Will deploy to review-demo/2-ma_branche" and "Will deploy to review-real/2-ma_branche", each with a small square icon. The bottom section features a green "Merge" button and a checkbox labeled "Delete source branch". Below this, a summary states: "1 commit and 1 merge commit will be added to master. Modify merge commit". At the very bottom, a note says: "You can merge this merge request manually using the command line".

Ici, on voit que la pipeline s'est correctement déroulée, et que le code **peut** être mergé dans la branche du dépôt de référence.

ATTENTION: Une CI se déroulant correctement ne certifie aucunement que votre développement fonctionne. C'est à vous de valider, ou la personne que vous avez désigné sur la MR le bon fonctionnement de votre développement.

Pour voir tous les processus qui ont été démarrés automatiquement pour valider votre MR, vous pouvez cliquer sur la gauche de **Pipeline**, sur l'icône vert. Vous verrez alors quelque chose de similaire à ceci :



Pipeline #17323 triggered 3 hours ago by Vincent COFFIN

[ADD] Test.txt file

⌚ 6 jobs for `2-ma_branche` in 1 minute and 52 seconds (queued for 2 seconds)

latest



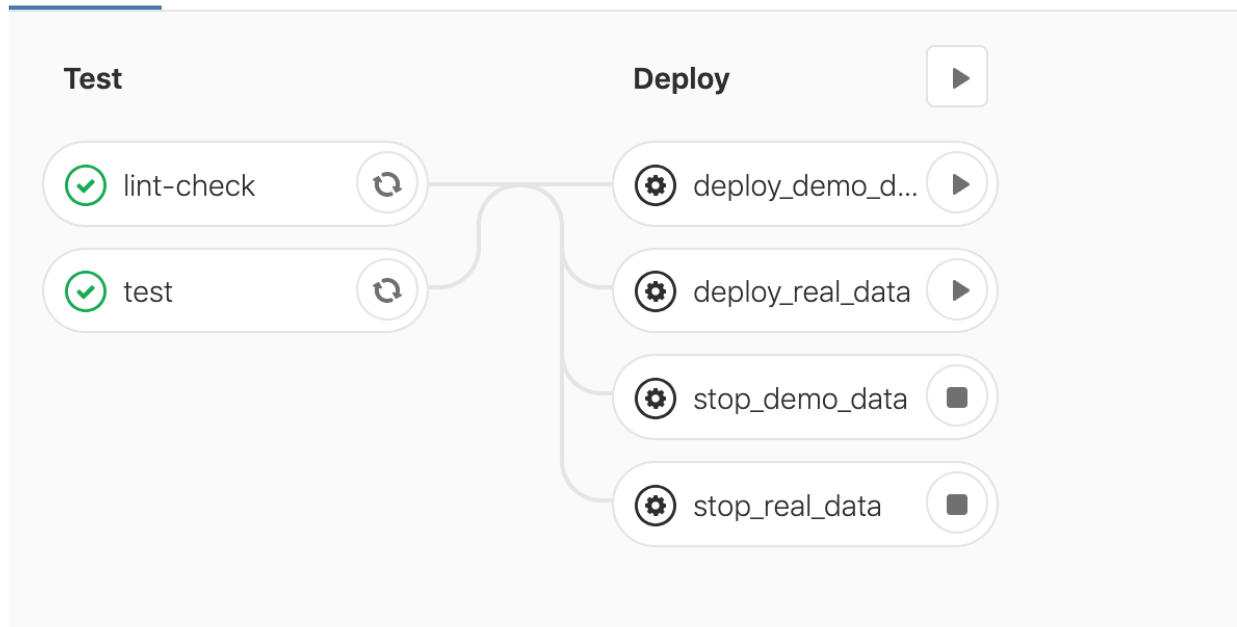
7872ca2b



Pipeline

Jobs

6



Tip

Certaines étapes de validation ne sont pas lancées automatiquement. Lors de la réalisation d'une MR, seules 2 étapes sont déclenchée, à savoir le **lint_check** , qui vérifie la qualité du code, et le **test** , qui effectue les tests unitaires.

Tip

Une autre étape de validation est lancée, pour l'instant uniquement une fois la MR validée, qui vise à créer la base template utilisée par la suite. Cette étape est parfois nécessaire pour valider une MR qui ne passe pas, puisque la base template est utilisée pour effectuer les tests lancés lors de l'étape de **test**.

3 Informations complémentaires

Quelques informations supplémentaires sont actuellement disponibles dans Gitlab.

3.1 Informations projets

Sur la fiche Gitlab de chaque projet client, est disponible **l'URL de production ainsi que l'URL de pré-production** du client. Si ce n'est pas le cas, il faut l'ajouter par le biais des réglages généraux du projet.

3.2 Conventions de nommage

Afin de mieux savoir ce qui est poussé, et pouvoir le lier simplement à une issue Gitlab, elle-même rattachée à une Merge Request encore elle-même rattachée à une tâche Odoo, nous avons mis en place un process de développement qui nous permet de retrouver simplement les développements réalisés, et faire le lien entre MR / Issues / Tâches.

Ainsi, les branches que vous poussez sur Gitlab, pour tous les projets, doivent être nommées de la façon suivante :

`<N°_issue>-<Nom_branche>`

Si le nommage n'est pas correct, un message d'erreur apparaîtra, vous expliquant quels noms sont autorisés lors d'un **push**. Si vous voyez un nom de branche autorisé manquant (Pouvant être utilisé lors de certaines situations), **parlons en au SCRUM !**

4. Gitlab Runners (SYS ADMIN)

TO DO

5. Sécurité Gitlab

Cette partie de la documentation de Gitlab a pour but d'expliquer les aspects sécurité de Gitlab, et la façon dont il doit être utilisé pour un niveau de sécurité le plus optimal possible.

5.1 Signature des commits

Afin de certifier que les commits réalisés le sont bien par les bonnes personnes, les signatures de commits peuvent être utilisées.

Qu'est ce que c'est que ça exactement ?

En fait, signer ses commits, c'est certifier qu'un commit nous appartient, et que c'est bien nous qui avons créé le commit que nous allons signer. C'est un moyen de s'assurer de l'intégrité d'un commit et de son auteur. Sinon, il est tout à fait possible de commiter au nom de quelqu'un, en réalisant un **commit --amend** par exemple sur un commit déjà réalisé.

En somme, signer ses commits, c'est s'assurer que personne ne soit en capacité de commiter à notre nom ! :)

Alors comment fais t-on ?

5.2 Configuration de GPG

Afin de mettre en place une signature de commit, il est nécessaire de configurer GPG dans Gitlab. C'est avec nos clés GPG que nous allons signer nos commits.

Pour cela, il est nécessaire de respecter à la lettre les étapes ci-dessous :

5.2.1 Vérifications préalables

Avant de pouvoir configurer GPG dans gitlab, vous devez au préalable vérifier votre fichier de configuration git pour voir quel e-mail est utilisé pour vos commits.

Cet e-mail doit être le même qu'une de vos **identités GPG**, qui sera vérifiée dans Gitlab. Si ce n'est pas le cas, les commits ne seront jamais vérifiés.

Pour voir la configuration de votre git :

```
cat ~/.gitconfig
```

La partie qui nous intéresse dedans est celle contenant notre email, au début du fichier :

L'email à cet endroit doit être votre e-mail actuel, de la bonne forme.

Une fois cela vérifié, on passe à la partie vérification de la partie "**identités GPG**".

Pour vérifier si votre clé GPG a bien votre e-mail actuel comme identité, lancez la commande suivante dans un terminal : **gpg --list-secret-keys --keyid-format LONG**

Vous devriez avoir un résultat similaire à celui-ci :

```
/Users/vincentcoffin/.gnupg/pubring.kbx
-----
sec    rsa4096/XXXXXXXXXXXXXXXXXXXX 2018-02-26 [SC] # Ceci est un comm
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
uid           [  ultime ] Vincent COFFIN <vco@subteno-it.com>
uid           [  ultime ] Vincent COFFIN <vincent.coffin@subteno
uid           [  ultime ] Vincent COFFIN <vincent.coffin@syleam.
ssb    rsa4096/D8B35AD86DE69D21 2018-02-26 [E]
```

Vous devriez voir ici l'email définie dans votre gitconfig, qui est votre e-mail actuel.

Si ce n'est pas le cas, nous allons alors ajouter une identité à votre clé GPG Si vous voyez votre e-mail, vous pouvez passer cette partie.

Editer votre clé :

Dans le résultat de la commande précédente, copiez le fingerprint de la clé, affiché sous **sec rsa4096/XXXXXXXXXXXX** (Je vous ai mis un petit commentaire :)) Puis utilisez la commande suivante pour éditer votre clé :

```
gpg --edit-key <fingerprint_de_votre_clé>
```

Vous entrerez en mode édition de votre clé. Voici le résultat pour moi :

```
{master} vincentcoffin@MBP-de-Vincent:~/Projects/Config/documentation
gpg (GnuPG) 2.2.16; Copyright (C) 2019 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

La clef secrète est disponible.

```
sec  rsa4096/XXXXXXXXXXXX
    créé : 2018-02-26  expire : jamais      utilisation : SC
    confiance : ultime      validité : ultime
ssb  rsa4096/XXXXXXXXXXXX
    créé : 2018-02-26  expire : jamais      utilisation : E
[  ultime ] (1). Vincent COFFIN <vco@subteno-it.com>
[  ultime ] (2) Vincent COFFIN <vincent.coffin@subteno-it.fr>
[  ultime ] (3) Vincent COFFIN <vincent.coffin@syleam.fr>
```

gpg>

Ensuite, tapez dans la console GPG qui s'est ouverte : **adduid**

Vous devrez renseigner:

- Votre vrai nom et prénom : COFFIN Vincent (pour ma part)
- L'email rattaché : vco@subteno-it.com (mailto:vco@subteno-it.com)
- Commentaire : Rien à mettre ici

Un prompt de check apparaîtra. Vous pouvez valider votre nouvelle identité en tapant "O" puis entrée.

Super, vous avez ajouté une nouvelle identité à votre clé GPG :)

5.2.2 Ajout de la clé sur Gitlab

Pour ajouter votre clé GPG à Gitlab, c'est très simple. Utilisez la commande suivante pour lister vos identités :

```
gpg --list-secret-keys --keyid-format LONG
```

Maintenant nous allons copier le fingerprint de cette clé pour exporter la partie publique de notre clé GPG, qui devra être renseignée dans Gitlab :

```
$ gpg --list-secret-keys --keyid-format LONG
/Users/vincentcoffin/.gnupg/pubring.kbx
-----
sec  rsa4096/COPIEZMOI 2018-02-26 [SC]
    CENUMERONESTPASCELUIQUIDOITETRECOPIE
uid          [  ultime ] Vincent COFFIN <vco@subteno-it.com>
uid          [  ultime ] Vincent COFFIN <vincent.coffin@subteno-it.fr>
uid          [  ultime ] Vincent COFFIN <vincent.coffin@syleam.fr>
ssb  rsa4096/D8B35AD86DE69D21 2018-02-26 [E]
```

pour exporter la clé publique, tapez cette commande en adaptant le fingerprint à votre propre cas :

```
$ gpg --armor --export COPIEZMOI
```

Une fois cette commande réalisée (l'export), un long block de texte devrait s'être affiché (toute votre clé publique).

Copiez là entièrement (copiez TOUT ce qui s'est affiché, même les **BEGIN/END PGP PUBLIC ...** etc)

A partir d'ici, rendez-vous dans Gitlab pour finaliser l'ajout de votre clé dans Gitlab.

5.2.3 Configuration Gitlab

Rendez-vous dans vos réglages personnels (en haut à droite) puis → **Clé GPG**

Collez votre clé dans le champ indiqué, puis cliquez sur **ajouter la clé**.

Votre clé devrait par la suite apparaître sur la page, et Gitlab vous montre les identités (e-mails) qui sont vérifiées et celle qui ne le sont pas. Si votre identité n'est pas vérifiée, pas de panique, il vous reste une dernière étape :

Rendez-vous dans **E-mails** dans le menu de gauche.

Ajoutez votre adresse e-mail à cet endroit.

5.2.4 Signature automatique de commits

Nous allons enfin dire à Git de signer nos commits "par défaut" avec notre clé GPG. Le but est de ne pas avoir à le faire manuellement à chaque commit !

Pour cela, nous allons d'abord dire à Git quelle clé il doit utiliser. Utilisez la commande suivante, en renseignant le fingerprint de votre propre clé (comme auparavant) :

```
git config --global user.signingkey <fingerprint_de_votre_clé>
```

Puis dites simplement à Git de signer chaque commit :

```
git config --global commit.gpgsign true
```

Effectuez ensuite un commit, puis vérifiez à l'aide de la commande `git llog` si votre commit est bien signé par clé GPG :

```
* [G 383ca6d] [2020-03-02] Vincent COFFIN: Ce commit est signé (Note:
* [N 856a380] [2020-02-26] Vincent COFFIN: Ce commit N'EST PAS signé
```



Vous pourrez ensuite vérifier que vos commits apparaissent bien signés dans Gitlab, sur une merge-request par exemple :



Vos commits sont désormais bien signés :)

5.3 Troubleshooting

Si vous rencontrez des problèmes au cours de l'installation qui ne sont pas répertoriés ici, tournez vous vers les personnes qui assurent le support interne.

5.3.1 Echec du déchiffrement

Si lorsque vous utilisez Pass après avoir effectué ces manipulations, vous rencontrez un problème de déchiffrement lié à votre clé GPG, c'est que GPG ne sait plus quelle console utiliser pour vous demander votre mot de passe, saute cette étape, et ne peut donc pas

déchiffrer le contenu que vous souhaitez lire.

Pour corriger le problème, on peut installer **pinentry-mac** sur le MAC et le définir comme programme à utiliser par GPG pour vous demander votre mot de passe :

```
$ brew install pinentry-mac
```

Ensuite, une fois l'installation réalisée, brew vous proposera de créer un fichier **gpg-agent.conf**, et de lui fournir un contenu précis. Ce fichier **gpg-agent.conf** est le fichier de configuration de l'agent gpg (qui est lui même le daemon qui tourne en arrière plan et vous écoute lorsque vous utilisez GPG).

Editez s'il existe, ou créez le fichier **~/ .gnupg/gpg-agent.conf** et fournissez le de ce contenu :

```
pinentry-program /usr/local/bin/pinentry-mac
```

Une fois cela fait, sauvegardez le fichier, puis relancez votre agent gpg à l'aide de la commande suivante :

```
$ killall gpg-agent
```

Lancez ensuite une commande **PASS** ou bien tentez de faire un commit signé, ce qui devrait vous demander votre mot de passe au travers d'une jolie invite de commande :-)

Tip

Il est possible avec cette invite de commande, de sauvegarder votre passphrase dans le Keychain, qui est le gestionnaire de mot de passe de Mac OS, pour ne plus avoir à taper votre passphrase GPG à l'avenir. Plutôt pratique !

5.3.2 Erreur git : Failed to write Objects

Warning

Si une erreur git survient en tentant de signer un commit, liée à l'écriture de l'objet Git, alors le problème est probablement le même que la section au dessus. Effectuez donc les étapes décrites en **3.1 Echec du déchiffrement** pour vous débloquer.

6. Gitlab Runner : Comment ça marche ?

Afin de permettre d'automatiser les tests des développements réalisés, il existe dans Gitlab la possibilité de configurer des **runners**, qui sont au final des serveurs qui attendent d'effectuer automatiquement certaines actions de compilation d'Odo.

En somme, lorsque vous faites une Merge Request, votre code est testé automatiquement sur plusieurs points.

Warning

Actuellement, deux systèmes de runners cohabitent. D'un côté, l'ancien système, qui n'effectue que les tests les plus simples (Tests unitaires, lint check et génération de la base template), et d'un autre côté, un nouveau système qui teste plus profondément vos développements, en testant votre code sur une base vierge, mais aussi sur la base de données réelle du client.

6.1 Ancien système

L'ancien système comporte les actions suivantes :

- **generate_template** : Génération de la base template qui sera utilisée pour effectuer les tests.
- **lint_check** : Vérifie la qualité syntaxique de votre code.
- **test** : Lance les tests unitaires.
- **deploy-demo-data** : Déploie une base de démo avec le code.
- **stop-demo-data** : Stoppe le déploiement des données de démo.

Warning

Pour certains rares clients, il est aussi possible de déployer des instances pour chaque développement. On compte : FPPM et Bobbies

- **deploy-real-data** : Déploie un jeu de données réel du client sur une instance.
- **stop-real-data** : stoppe le déploiement qui a été lancé.

6.2 Nouveau système

Le nouveau système de runner utilise docker, et permet de déployer pour tout client des instances indépendantes de tests.

On retrouve les éléments suivants :

- **build-image** : Construit l'image docker qui sera utilisée pour déployer Odoo. A ce stade, **contactez les admin sys en cas d'erreur**.
- **lint_check** : Vérifie la qualité syntaxique de votre code.
- **code** : Teste si votre code fonctionne avec une base de données vierge (teste la compilation d'Odoo).
- **empty-image** : Vérifie que l'image Odoo peut se lancer. A ce stade, **contactez les admin sys en cas d'erreur**.
- **real-update** : Vérifie qu'Odoo se lance bien avec les données du client.

Enfin, vous avez la possibilité de lancer une instance de test qui pourra être fournie au client avec le dernier job :

- **real** : Lance une instance de test avec les données du client. Cette instance est isolée des autres développements, et n'est pas donc impactée par autre chose que votre développement.

Tip

Une fois la dernière CI passée, le lien sera disponible à la toute fin du log de CI. Vous verrez une ligne comme celle-ci :

```
$ You can access to real instance through this URL in some minutes. I
$ echo http://${CI_COMMIT_BRANCH}.${CI_PROJECT_NAME}.${TESTING_URL}:50353
$ http://146-gitlab-ci-docker.bohin.runner-2.subteno-it.net:50353
```

7. Spécificités système en place

Voici la liste des actions spécifiques réalisées en lien avec le serveur Gitlab :

- Le serveur Gitlab Runner 1 est installé sous Ubuntu, pour des raison de compatibilité de paquets Gitlab
- Docker est installé sur le serveur Gitlab Runner 1, pour faire tourner des CI Docker.
- Un cron est en place sur l'utilisateur **ubuntu** du serveur Gitlab Runner 1, et vide toutes les heures les volumes docker non utilisés (pour l'espace disque).
- Des volumes ont été montés pour permettre de redimensionner aisément l'espace disque du runner.
- Grafana a été implémenté manuellement sur le serveur Gitlab Runner 1