

Pipeline CI/CD avec GitLab CI


C'est quoi GitLab ?

- C'est une forge logiciel base sur **GIT**
- Nombreuses fonctionnalités :
 - Gestion de bugs/projet
 - Wiki
 - **Intégration Continue et déploiement continu**
 - et bien d'autres choses encore !

Une clé SSH pour GitLab ?

Sur votre poste

Bien qu'il soit possible d'utiliser un Login/Pass pour utiliser GitLab, il est plus fiable et simple d'utiliser une paire de clés SSH.

- Sur votre poste :
 - `ssh-keygen -t ed25519 -C "Yoan PC Bureau"`
 - Répertoire : idéalement `~/.ssh/` sur les systèmes Unix
 -  **Ne pas mettre de passphrase**

Une clé SSH pour GitLab ?

Sur GitLab

- Dans les menus GitLab :
 - Préférences > SSH Keys
 - Copier coller le contenu de votre fichier *.**pub** généré à l'étape précédente

⚠ Gardez **précieusement** vos deux fichiers de clés (publique et privée) !

Création de notre repository GitLab

Nous allons procéder en **trois** étapes :

- Création du projet **Démo** en local
- Création du repository sur GitLab
- Push de notre projet sur notre repo GitLab

Création du projet Démo

Positionnez vous dans notre repertoire de travail `formation-ci-cd`, puis initialisons un nouveau projet **demo** :

```
symfony new --demo symfony-gitlab
```

Et enfin, entrons dans le repertoire du projet:

```
cd symfony-gitlab
```

Création du repository chez GitLab

- Vous devez ouvrir un compte, c'est **gratuit** !
- Cliquez sur **New project**
- Puis **Create blank project**
- Project Name : **Symfony GitLab**
- ⚠ Décocher l'option **Initialize repository with a README**
- **Create project** 🚀

Push de votre projet sur GitLab

Changer le nom de la branche de **Master** à **Main**
(avec VSCode par exemple)

```
git remote add origin git@gitlab.com:formation-ci-cd-pour-les-d-veloppeurs-php/symfony-gitlab-ok.git
git push -u origin --all
git push -u origin --tags
```

Le projet **demo** devrait être désormais disponible sur GitLab.

La gestion des branches sur GitLab

Dans cette formation nous n'abordons pas la notion de Workflow Git, mais le sujet est important !

- Créons une branche **protégée** :
 - Dans **Repository > Branches**, créons une branche **production**
 - Dans **Settings > Repository > Protected branches**
 - Allowed to merge : **Maintener**
 - Allowed to push : **No one**

Les runners GitLab

- Les pipeline GitLab CI sont *exécutés* sur des **Runners** :
 - Par défaut vous disposez de 400 minutes/mois gratuites
 - Vous pouvez installer des Runners sur votre poste :
 - Installation de GitLab Runner
 - `sudo gitlab-runner register`
 - Enter an executor : `docker`
 - Enter the default Docker image : `php`

Le fichier .gitlab-ci.yml

C'est **LE** fichier qui gère tout votre pipeline avec GitLab. Un exemple :

```
image: php:8.1
cache:
  paths:
    - vendor/
before_script:
  - docker-php-ext-install pdo_mysql
stages:
  - Foo
bar:
  stage: foo
  script:
    - /bin/console lint:twig templates --env=prod
```

Un pipeline pour découvrir !

- Testons le principe, et créons un pipeline **ULTRA** simple !

```
image: php:8.1
```

```
stages:
```


```
  - ls
```

```
test:
```

```
  stage: ls
```

```
  script:
```

```
    - ls
```

-
- Commit + Git Push
- Le pipeline s'exécute 

Le principe du Before Script

Même portion de scripts pour chaque étapes ? **before_script** !

```
image: php:8.1
before_script:
  - apt-get update && apt-get install -y git libzip-dev
  - curl -sSk https://getcomposer.org/installer | php -- --disable-tls && mv composer.phar /usr/local/bin/composer
  - docker-php-ext-install zip
  - composer install
stages:
  - ls
test:
  stage: ls
  script:
    - ls
```

La gestion du cache

Accélérez l'exécution de vos pipelines avec du cache !

```
image: php:8.1
cache:
  paths:
    - vendor/
before_script:
  - apt-get update && apt-get install -y git libzip-dev
  - curl -sSk https://getcomposer.org/installer | php -- --disable-tls && mv composer.phar /usr/local/bin/composer
  - docker-php-ext-install zip
  - composer install
stages:
  - ls
test:
  stage: ls
  script:
    - ls
```

Mettons en place un Pipeline CI plus complexe !

- Check des vulnérabilités
- PHP CS FIXER
- PHP STAN
- Linteurs Symfony
- Tests PHP Unit

Ready ?

GO ! 

Check des vulnérabilités

A partir de maintenant, je simplifie les slides pour une meilleure lisibilité 😊

```
# ...
before_script:
  # ...
  - curl -sS https://get.symfony.com/cli/installer | bash && mv /root/.symfony/bin/symfony /usr/local/bin/symfony
stages:
  - security
check_vulnerabilities:
  stage: security
  script:
    - symfony check:security
```


PHP CS FIXER

```
# ...  
stages:  
    - security  
    - qa  
  
php_cs_fixer:  
    stage: qa  
    script:  
        - composer require friendsofphp/php-cs-fixer  
        - ./vendor/bin/php-cs-fixer fix --dry-run
```

PHP STAN

⚠ Pensez à utiliser la configuration **phpstan.neon** élaborée plus tôt!

```
# ...
stages:
    - security
    - qa

php_stan:
    stage: qa
    script:
        - composer require phpstan/phpstan
        - composer require phpstan/extension-installer
        - composer require phpstan/phpstan-symfony
        - ./vendor/bin/phpstan analyse src --memory-limit 1G
```

Linteurs Symfony

```
# ...
stages:
    - qa

lint:
    stage: qa
    script:
        - ./bin/console lint:yaml config --parse-tags
        - ./bin/console lint:twig templates --env=prod
        - ./bin/console lint:container --no-debug
        - ./bin/console doctrine:schema:validate --skip-sync -vvv --no-interaction
```

Tests PHP UNIT

```
stages:
```

```
- tests
```

```
php_unit:
```

```
  stage: tests
```

```
  script:
```

```
    - ./bin/phpunit
```

Création d'un Artefact



Résultats de nos tests PHP Unit directement dans GitLab ?

- PHP Unit doit générer un fichier report.xml au format **junit**
- On doit demander à GitLab de **stocker** le fichier

```
php_unit:  
  stage: tests  
  script:  
    - ./bin/phpunit --log-junit report.xml  
  artifacts:  
    when: always  
    reports:  
      junit: report.xml
```

Gestion des échecs

Dans la vraie vie, vous allez avoir des étapes en echec (c'est le but du pipeline de le detecter justement) !

- En l'etat, notre pipeline laisse tout passer, que les étapes réussissent... ou non 
- Ajoutons une gestion des échecs 

```
foo:
  stage: bar
  script:
    - ls
  allow_failure: false # or true
```

Un badge sur la home du projet ?

*Avant de basculer dans la partie **CD**, une petite douceur 🍫*

Dans GitLab :

- **Settings > General > badges**
- Name : Pipeline
- Link : `https://gitlab.com/{project_path}`
- Badge :
`https://gitlab.com/{project_path}/badges/{default_branch}/pipeline.svg`

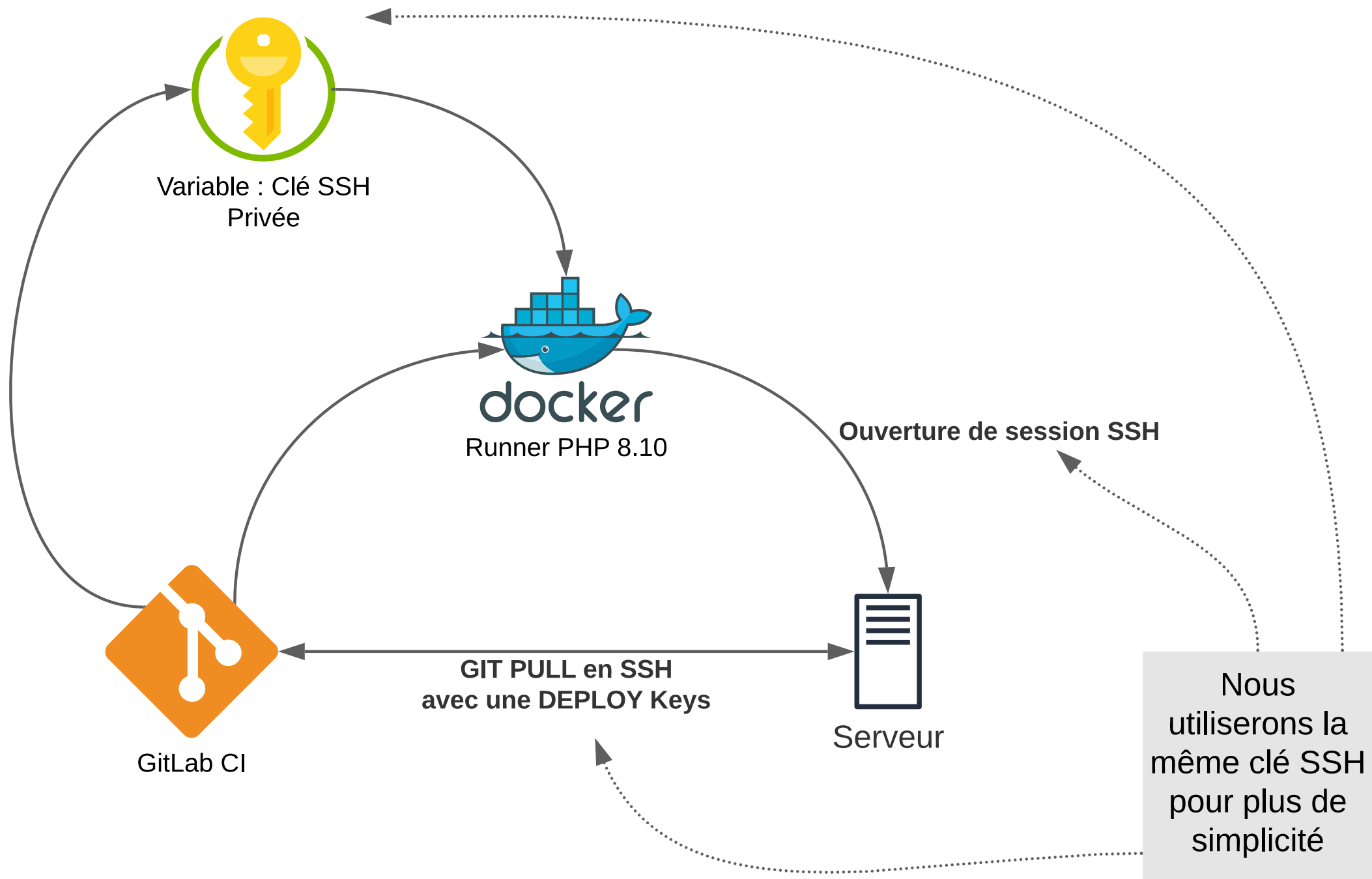
Le déploiement continu avec GitLab

Nous allons prendre un **cas d'utilisation fréquent** pour un hébergement d'un projet **PHP/Symfony** :

- Un serveur (VPS, Mutualisé ou n'importe quoi d'autre)
- Un accès via SSH **obligatoire**
 - via l'utilisation d'une **clé SSH**
- Une première installation du projet **à la main**

 Cette approche est simple **MAIS** très efficace et **RAPIDE** à mettre à oeuvre !

Un schéma avant de plonger ?



Clé SSH et ouverture de session

⚠ En fonction de votre hébergeur :

- Vous devrez générer une clé depuis le serveur (et lui permettre d'ouvrir une session)
- **OU** Ajouter une clé existante via une console d'administration
- ☒ Si l'ouverture de session est possible, vous pourrez poursuivre sans encombres !

Git Clone KO 🤔

En théorie, vous ne devriez pas pouvoir faire un `git clone` du projet (sauf si le projet est **public**).

Dans GitLab :

- **Settings > Repository > Deploy Keys**
 - Copier-coller la clé publique qui permet l'ouverture de session sur votre serveur
 - Ne cocher pas **grant write permission**

Git Clone KO 🤔

- Copier les clés de votre poste vers le serveur

```
scp --p ~/.ssh/id_gitlab.pub user@serveur:~/.ssh/id_gilab.pub  
scp --p ~/.ssh/id_gitlab user@serveur:~/.ssh/id_gilab
```

- Sur le serveur (en fonction des cas)

```
cd ~/.ssh/  
chmod 400 id_gitlab.pub  
chmod 400 id_gitlab
```

Git Clone KO 🤔


- Créer un fichier `config` dans `~.ssh/`
- Expliquer la clé à utiliser pour GitLab :

```
Host gitlab.com
  HostName gitlab.com
  IdentityFile ~/.ssh/id_gitlab
  User git
```

- 🚀 Le **GIT CLONE** fonctionne avec notre **Deploy Key** !

La gestion des secrets

Dans GitLab :

- **Settings > CI/CD > Variables > Add variables**
 - Key : `SSH_PRIVATE_KEY`
 - Value : Le contenu de votre clé **privée**
 - Key : `SSH_HOST`
 - Value : `@ip -p {port}` (ip+port de votre serveur)
- Désormais, nous pourrons utiliser dans notre pipeline cette clé SSH pour ouvrir une session sur notre serveur 

Utiliser la clé SSH dans la pipeline

```
stages:
  - deploy

production:
  stage: deploy
  script:
    - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client git -y )'
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add - > /dev/null
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" >> ~/.ssh/config'
    - |
      ssh $SSH_HOST << EOF
      cd /var/www
      git pull
      EOF
```


Déployer "simplement" un projet Symfony

```
production:
  stage: deploy
  script:
    (..)
    - |
      ssh $SSH_HOST << EOF
      cd /var/www
      git pull
      composer install --no-dev --optimize-autoloader
      symfony console d:m:m -n
      APP_ENV=prod APP_DEBUG=0 php bin/console cache:clear
      EOF
```

Adpatation !

- Chaques projets à des spécificités
- Vous pourriez avoir besoin de:
 - `npm install` et `npm run build`
 - Ou d'autre commandes spécifiques à votre projet
- Vous pourriez avoir besoin de sauvegarder la base de données juste avant la migration ...
- Faites jouer votre imagination 🪐

Conditionner certaines étapes de la pipeline

- Ne déployons que lors d'une Merge sur la branche **Production**

```
stages:  
  - deploy  
  
production:  
  only:  
    - production  
  stage: deploy
```

- Sur GitLab : **merger** sur **Production**