

# Pipeline CI/CD avec GitHub CI


# C'est quoi GitHub ?

- 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 GitHub

## Sur votre poste

Il n'est plus possible d'utiliser un login/pass, vous devez 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 GitHub ?

## Sur GitHub

- Dans les menus GitHub :
  - **Settings > SSH and GPG Keys > New SSH Key**
  - Copier coller le contenu de votre fichier **\*.pub** généré à l'étape précédente

⚠ Gardez **précieusement** vos deux fichiers de clés (publique et privée), nous en aurons besoin plus tard !

# Création de notre repository GitHub

Nous allons procéder en **trois** étapes :

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

# Création du projet Démo


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

```
symfony new --demo symfony-github
```

Et enfin, entrons dans le repertoire du projet:

```
cd symfony-github
```

# Création du repository chez GitHub

- Vous devez ouvrir un compte, c'est **gratuit** !
- Cliquez sur **New repository**
- Repository Name : **Symfony GitHub**
- **Create repository** 

# Push de votre projet sur GitHub

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

```
git remote add origin git@github.com:johndoe/symfony-github.git  
git branch -M main  
git push -u origin main
```

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



# La gestion des branches sur GitHub

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

- Créons une branche :
  - Cliquer sur la branche **main**
  - Saisir dans le champ de recherche : `production`
  - **Create branche**


# Supprimons le pipeline par défaut

Le projet **demo** de Symfony arrive avec un pipeline (qui peut être en échec) lors de sa création, nous allons supprimer le pipeline !

- Supprimer le repertoire `.github`
- Et faite un **commit + push**

# Le repertoire .github

C'est **LE** repertoire qui gère tout votre pipeline avec GitHub.

- Il contient un repertoire `workflow`
- C'est ce repertoire qui contiendra nos fichiers de pipeline
- Ce sont des fichiers au format **YAML**
-  Pour nous simplifier l'apprentissage, nous utiliserons **un fichier** pour la CI et **un autre fichier** pour la CD.

# Un pipeline pour découvrir !

- Testons le principe, et créons un pipeline **ULTRA** simple **directement** sur GitHub. **Actions > New workflow > set up a ...**

```
# cd.yaml
name: Pipeline-CI
on:
  push:
    branches: [ main ]
jobs:
  ci:
    runs-on: ubuntu-latest
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
      - uses: actions/checkout@v2
      - name: LS
        run: ls
```

# Ajoutons les outils du monde PHP

```
name: Pipeline-CI-CD

on:
  push:
    branches: [ main ]

jobs:
  ci:
    runs-on: ubuntu-latest
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
      - uses: actions/checkout@v2

      # https://github.com/shivammathur/setup-php (community)
      - name: Setup PHP, extensions and composer with shivammathur/setup-php
        uses: shivammathur/setup-php@v2
        with:
          extensions: mbstring, xml, ctype, iconv, intl, pdo, pdo_mysql, dom, filter, gd, iconv, json, mbstring, pdo

      # Runs a single command using the runners shell
      - name: LS
        run: ls
```

# La gestion du cache

Histoire d'acceller nos pipelines, mettons en cache notre repertoire

vendor

- name: Cache multiple paths  
uses: actions/cache@v2  
with:  
  path: |  
    vendor  
  key: \${ runner.os }-\${ hashFiles('composer.lock') }
- name: "Composer install"  
run: composer install --no-interaction --no-progress

# 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

```
# Install Symfony CLI
```

```
- name: Symfony CLI
```

```
run: |
```

```
    curl -sS https://get.symfony.com/cli/installer | bash
```

```
    mv /home/runner/.symfony/bin/symfony /usr/local/bin/symfony
```

```
# Check vulnerabilities
```

```
- name: Symfony Check Vulnerabilities
```

```
run: symfony check:security
```



# PHP CS FIXER

Bien qu'il existe des **recettes**, je préfère utiliser un *bon vieux script*, c'est plus simple et lisible !

```
# PHP CS FIXER
- name: PHP-CS-Fixer
  run: |
    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!

```
# PHP STAN
- name: PHP Stan
  run: |
    ./vendor/bin/simple-phpunit install
    composer require phpstan/phpstan
    composer require phpstan/extension-installer
    composer require phpstan/phpstan-symfony
    ./vendor/bin/phpstan analyse src --memory-limit 1G
```

# Linteurs Symfony

- name: Lint YAML files  
run: `./bin/console lint:yaml config --parse-tags`
- name: Lint Twig templates  
run: `./bin/console lint:twig templates --env=prod`
- name: Lint Parameters and Services  
run: `./bin/console lint:container --no-debug`
- name: Lint Doctrine entities  
run: `./bin/console doctrine:schema:validate --skip-sync -vvv --no-interaction`

# Tests PHP UNIT

Logiquement, maintenant plus de surprise pour la marche à suivre !

```
- name: PHP Unit  
  run: ./bin/phpunit
```

-  en voila un CI qui envoie non ?

# Gestion des erreurs

En fonction des cas et de vos exigences, vous allez peut être vouloir décider que faire en cas d'échec d'une étape de votre pipeline.

C'est **hyper simple** :

```
jobs:  
  ci:  
    runs-on: ubuntu-latest  
    continue-on-error: false # or false
```

# Un badge sur la home du projet ?

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

Dans GitHub :

- **Actions > Pipeline-CI > ... > Create status badge**
- Laissez toutes les options par défaut
- **Copy status badge Markdown**
- Collez le bout de code dans votre `README.md`
- C'est beau ! 🙄

# Le déploiement continu avec GitHub

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 !

# 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 le repository GitHub :

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

# Git Clone KO 🤔

- Copier les clés de votre poste vers le serveur
- Sur le serveur

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

# Git Clone KO 🤔

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

```
Host github.com
  HostName github.com
  IdentityFile ~/.ssh/mykey
  User git
```

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

# La gestion des secrets

Dans GitHub **Settings** > **Secrets** > **New repository secret**.

- Name : `SSH_PRIVATE_KEY`
  - Value : Le contenu de votre clé **privée**
- Name : `SSH_HOST`
  - Value : @ip(ip de votre serveur)
- Name : `SSH_USER`
  - Value : Votre user sur le serveur
- Name : `SSH_PORT`
  - Value : Le port SSH de votre serveur

# Création du Pipeline CD

```
# .github/workflows/cd.yml
name: Pipeline-CD
on:
  push:
    branches: [ main ]
jobs:
  cd:
    runs-on: ubuntu-latest
    continue-on-error: false
    steps:
      - uses: actions/checkout@v2
      - name: Deploy Over SSH
        uses: appleboy/ssh-action@master
        with:
          host: ${ secrets.SSH_HOST }
          username: ${ secrets.SSH_USER }
          key: ${ secrets.SSH_PRIVATE_KEY }
          port: ${ secrets.SSH_PORT }
          script: ls
```

# Déployer "simplement" un projet Symfony

```
- name: Deploy Over SSH
  uses: appleboy/ssh-action@master
  with:
    host: ${ secrets.SSH_HOST }
    username: ${ secrets.SSH_USER }
    key: ${ secrets.SSH_PRIVATE_KEY }
    port: ${ secrets.SSH_PORT }
    script: |
      cd /var/www
      git pull
      composer install
      symfony console d:m:m -n
      APP_ENV=prod APP_DEBUG=0 php bin/console cache:clear
```

# 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**

```
name: Pipeline-CD

on:
  push:
    branches: [ production ]
```

- Sur GitHub: **merger** sur **Production**