CI/CD

Solutions de déploiements continus

🇫🇷 ESGI Lyon - M2-IW-ALT-S1 2023

- Présentation disponible à l'adresse: https://dduportal.github.io/dduportal/esgi-courses/2023-S1-M2-IW
- Version PDF de la présentation : Cliquez ici
- Contenu sous licence Creative Commons Attribution 4.0 International License
  - Une partie du contenu provient de https://github.com/cicd-lectures/slides (auteurs: Julien Levesy et moi-même)
- Code source de la présentation: https://github.com/dduportal/dduportal/esgi-courses

Comment utiliser cette présentation ?

- Pour naviguer, utilisez les flèches en bas à droite (ou celles de votre clavier)
  - Gauche/Droite: changer de chapitre
  - Haut/Bas: naviguer dans un chapitre
- Pour avoir une vue globale : utiliser la touche "o" (pour "**Overview**")

Bonjour !

La suite: vers le bas ⬇

Damien DUPORTAL

- Staff Software Engineer chez CloudBees pour le projet Jenkins 👨🏻‍⚖️
- Freelancer
- Me contacter :
  - damien.duportal @ gmail.com
  - dduportal
  - Damien Duportal

Et vous ?

A propos du cours

- Alternance de théorie et de pratique pour être le plus interactif possible
- Compatible Distanciel / Présentiel
- Contenu entièrement libre, open-source sous licence Creative Commons
    - N'hésitez pas ouvrir des Pull Request si vous voyez des améliorations ou problèmes: sur cette page (😉 wink wink)

Calendrier

- **Présentiel** 🎓 Jeudi 14 septembre 2023 - 9h45 → 19h
- **Présentiel** 🎓 Vendredi 15 septembre 2023 - 9h45 → 19h

Evaluation

- **Pourquoi ?** s'assurer que vous avez acquis un minimum de concepts
- **Quoi ?** Deux notes sur 20 (25% Contrôle Continu, 75% Projet)
- **Comment ?**
  - Contrôle Continu: Participation + attention pendant le cours, respect des échéance et des autres, éventuellement un QCM
  - Un projet GitHub (public) évalué selon une liste de critères *pré-déterminés*

Plan

- Introduction
- Rappels (Ligne de commande, Git)
- Intégration Continue (CI)
- Rappels (Docker)
- Déploiement Continu (CD)
- Livraison Continue (CD aussi!)

La suite: vers la droite ➡

Introduction

💡 Vous avez dit "Continu" ?

Avant : le cycle en V

What could go right?

Copyright © https://www.projectcartoon.com under the Creative Commons Attribution 3.0 Unported License

Et dans la vraie vie c'est pire !

Copyright © https://www.projectcartoon.com under the Creative Commons Attribution 3.0 Unported License

Problématique

- Travail basé sur:
  - Hypothèses initiales (capture du besoin)
  - Temps écoulé entre hypothèse et usage en production

⚠ Le besoin a forcément évolué dans ce laps de temps

⚠ Erreur de capture du besoin == coût de l'erreur décuplé avec le temps

Agile

- Travailler par petites itérations **complètes**
  - Commencer petit
  - Confronter le logiciel au plus tôt aux utilisateurs.
  - Refaire des hypothèses basées sur ce que l'on à appris, et recommencer !
- "Embrasser" le changement : Votre logiciel va changer en **continu**

Source

Manifeste Agile

- https://agilemanifesto.org/iso/fr/manifesto.html

Source

La clé : gérer le changement!

- Le changement ne doit pas être un événement, ça doit être la **norme**
- Faire en sorte que changer quelque chose soit:
  - Facile
  - Rapide
  - Stable
  - Sûr

Say Hello to DevOps

Heureusement, vous avez des outils à disposition !

Préparer votre environnement de travail

Outils Nécessaires 🛠️

- Un navigateur web récent (et décent)
- Un compte sur GitHub

GitPod

GitPod.io : Environnement de développement dans le ☁ "nuage"

- **But:** reproductible
- Puissance de calcul sur un serveur distant
- Éditeur de code VSCode dans le navigateur
- Gratuit pour 50h par mois (⚠)

Démarrer avec GitPod 🚀

- Rendez vous sur https://gitpod.io
- Authentifiez vous en utilisant votre compte GitHub:
  - Bouton "Login" en haut à droite
  - Puis choisissez le lien "Continue with GitHub"

⚠ Pour les "autorisations", passez sur la slide suivante

Autorisations demandées par GitPod 🔓

Lors de votre première connexion, GitPod va vous demander l'accès (à accepter) à votre email public configuré dans GitHub :

⚠ Passez à la slide suivante avant d'aller plus loin

Validation du Compte GitPod 📱

GitPod vous demande votre numéro de téléphone mobile afin d'éviter les abus (service gratuit).
Saisissez un numéro de téléphone valide pour recevoir par SMS un code de déblocage :

⚠ Passez à la slide suivante avant d'aller plus loin

Choix de l'Éditeur de Code ✎

Choisissez l'éditeur "VSCode Browser" (la première tuile) :

⚠ Passez à la slide suivante avant d'aller plus loin

Workspaces GitPod 🗂️

- Vous arrivez sur la page listant les "workspaces" GitPod :
- Un workspace est une instance d'un environnement de travail virtuel (C'est un ordinateur distant)
- ⚠ Faites attention à réutiliser le même workspace tout au long de ce cours⚠

Permissions GitPod <→ GitHub 🔐

- Pour les besoins de ce cours, vous devez autoriser GitPod à pouvoir effectuer certaines modification dans vos dépôts GitHub
- Rendez-vous sur la page des intégrations avec GitPod
- Éditez les permissions de la ligne "GitHub" (les 3 petits points à droits) et sélectionnez uniquement :
  - `user:email`
  - `public_repo`
  - `workflow`

Démarrer l'environnement GitPod

Cliquez sur le bouton ci-dessous pour démarrer un environnement GitPod personnalisé:

Open in Gitpod

Après quelques secondes (minutes?), vous avez accès à l'environnement:

- Gauche: navigateur de fichiers ("Workspace")
- Haut: éditeur de texte ("Get Started")
- Bas: Terminal interactif
- À droite en bas: plein de popups à ignorer (ou pas?)

Source disponible dans : https://github.com/dduportal/esgi-gitpod

Checkpoint 🎯

- Vous devriez pouvoir taper la commande `whoami` dans le terminal de GitPod:
  - Retour attendu: `gitpod`
- Vous devriez pouvoir fermer le fichier "Get Started"…
  - … et ouvrir le fichier `.gitpod.yml`

On peut commencer !

Ligne de commande

Guide de survie

🤔 Problématique

- Communication Humain <→ Machine
- Base commune de TOUS les outils

CLI

- 🇬🇧 CLI == "Command Line Interface"
- 🇫🇷 "Interface de Ligne de Commande"

REPL

Pour les théoriciens et curieux :

- 🇬🇧 REPL == "Read–eval–print loop"

https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

Anatomie d'une commande

```
ls --color=always -l /bin
```

Copy

- Séparateur : l'espace
- Premier élément (`ls`) : c'est la commande
- Les éléments commençant par un tiret − sont des "options" et/ou drapeaux ("flags")
  - "Option" == "Optionnel"
- Les autres éléments sont des arguments (`/bin`)
  - Nécessaire (par opposition)

Manuel des commande

- Afficher le manuel de `<commande>` :

```
# Commande 'man' avec comme argument le nom de ladite comman
man <commande>
```

  - Navigation avec les flèches haut et bas
    - Tapez `/` puis une chaîne de texte pour chercher
    - Touche `n` pour sauter d'occurrence en occurrence
  - Touche `q` pour quitter

🎓 Essayez avec `ls`, chercher le mot `color`

- 💡 La majorité des commandes fournit également une option (`--help`), un flag (`-h`) ou un argument (`help`)
- Google c'est pratique aussi hein !

Raccourcis

Dans un terminal Unix/Linux/WSL :

- `CTRL + C` : Annuler le process ou prompt en cours
- `CTRL + L` : Nettoyer le terminal
- `CTRL + A` : Positionner le curseur au début de la ligne
- `CTRL + E` : Positionner le curseur à la fin de la ligne
- `CTRL + R` : Rechercher dans l'historique de commandes

🎓 Essayez-les !

Commandes de base 1/2

- `pwd` : Afficher le répertoire courant
  - 🎓 Option `-P` ?
- `ls` : Lister le contenu du répertoire courant
  - 🎓 Options `-a` et `-l` ?
- `cd` : Changer de répertoire
  - 🎓 Sans argument : que se passe t'il ?
- `cat` : Afficher le contenu d'un fichier
  - 🎓 Essayez avec plusieurs arguments
- `mkdir` : créer un répertoire
  - 🎓 Option `-p` ?

Commandes de base 2/2

- `echo` : Afficher un (des) message(s)
- `rm` : Supprimer un fichier ou dossier
- `touch` : Créer un fichier
- `grep` : Chercher un motif de texte

Arborescence de fichiers 1/2

- Le système de fichier a une structure d'arbre
  - La racine du disque dur c'est `/` : 🎓 `ls -l /`
  - Le séparateur c'est également `/` : 🎓 `ls -l /usr/bin`
- Deux types de chemins :
  - Absolu (depuis la racine): Commence par `/` (Ex. `/usr/bin`)
  - Sinon c'est relatif (e.g. depuis le dossier courant) (Ex `./bin` ou `local/bin/`)

Source

Arborescence de fichiers 2/2

- Le dossier "courant" c'est `.` : 🎓 `ls -l ./bin # Dans le dossier /usr`
- Le dossier "parent" c'est `..` : 🎓 `ls -l ../ # Dans le dossier /usr`
- ~ (tilde) c'est un raccourci vers le dossier de l'utilisateur courant : 🎓 `ls -l ~`
- Sensible à la casse (majuscules/minuscules) et aux espaces : 🎓

```
ls -l /bin
ls -l /Bin
mkdir ~/"Accent tué"
ls -d ~/Accent\ tué
```

Copy

Un language (?)

- Variables interpolées avec le caractère "dollar" `$` :

```
echo $MA_VARIABLE
echo "$MA_VARIABLE"
echo ${MA_VARIABLE}

# Recommendation
echo "${MA_VARIABLE}"

MA_VARIABLE="Salut tout le monde"

echo "${MA_VARIABLE}"
```

- Sous commandes avec `$(<command>)` :

```
echo ">> Contenu de /tmp :\n$(ls /tmp)"
```

- Des `if`, des `for` et plein d'autres trucs (https://tldp.org/LDP/abs/html/)

Codes de sortie

- Chaque exécution de commande renvoie un code de retour (🇬🇧 "exit code")
  - Nombre entier entre 0 et 255 (en POSIX)
- Code accessible dans la variable **éphémère** `$?` :

```
ls /tmp
echo $?

ls /do_not_exist
echo $?

# Une seconde fois. Que se passe-t'il ?
echo $?
```

Entrée, sortie standard et d'erreur

```
ls -l /tmp
echo "Hello" > /tmp/hello.txt
ls -l /tmp
ls -l /tmp >/dev/null
ls -l /tmp 1>/dev/null

ls -l /do_not_exist
ls -l /do_not_exist 1>/dev/null
ls -l /do_not_exist 2>/dev/null

ls -l /tmp /do_not_exist
ls -l /tmp /do_not_exist 1>/dev/null 2>&1
```

Copy

Pipelines

- Le caractère "pipe" `|` permet de chaîner des commandes
  - Le "stdout" de la première commande est branchée sur le "stdin" de la seconde
    - Exemple : Afficher les fichiers/dossiers contenant le lettre `d` dans le dossier `/bin` :

```
ls -l /bin

ls -l /bin | grep "d" --color=auto
```

Exécution 1/2

- Les commandes sont des fichier binaires exécutables sur le système :

```
command -v cat # équivalent de "which cat"

ls -l "$(command -v cat)"
```

- La variable d'environnement `$PATH` liste les dossiers dans lesquels chercher les binaires
  - 💡 Utiliser cette variable quand une commande fraîchement installée n'est pas trouvée

Exécution 2/2

- Exécution de scripts :
  - Soit appel direct avec l'interprétateur : `sh ~/monscript.txt`
  - Soit droit d'exécution avec un "shebang" (e.g. `#!/bin/bash`)

```
$ chmod +x ./monscript.sh

$ head -n1 ./monscript.sh
#!/bin/bash

$ ./monscript.sh
# Exécution
```

Copy

Git

Guide de survie

🤔 Problématique

- Support de communication
  - Humain → Machine
  - Humain <→ Humain
- Besoins de traçabilité, de définition explicite et de gestion de conflits
  - Collaboration requise pour chaque changement (revue, responsabilités)

Tracer le changement dans le code

avec un **VCS** : 🇬🇧 Version Control System

également connu sous le nom de SCM (🇬🇧 Source Code Management)

Pourquoi un VCS ?

- Pour conserver une trace de **tous** les changements dans un historique
  - "Source unique de vérité" (🇬🇧 *Single Source of Truth*)
- Pour **collaborer** efficacement sur un même référentiel

Concepts des VCS

Source : https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Quel VCS utiliser ?

Nous allons utiliser **Git**

Git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

https://git-scm.com/

Les 3 états avec Git

- L'historique ("Version Database") : dossier `.git`
- Dossier de votre projet ("Working Directory") - Commande
- La zone d'index ("Staging Area")

Source : https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Rudiments-de-Git#*les_trois*%C3%A9tats

🎓 Exercice avec Git - 1.1

- Dans le terminal de votre environnement GitPod:
  - Créez un dossier vide nommé `projet-vcs-1` dans le répertoire `/workspace`, puis positionnez-vous dans ce dossier

```
mkdir -p /workspace/projet-vcs-1/
cd /workspace/projet-vcs-1/
```

  - Est-ce qu'il y a un dossier `.git/` ?
  - Essayez la commande `git status` ?
- Initialisez le dépôt git avec `git init`
  - Est-ce qu'il y a un dossier `.git/` ?
  - Essayez la commande `git status` ?

✅ Solution de l'exercice avec Git - 1.1

```
mkdir -p /workspace/projet-vcs-1/
cd /workspace/projet-vcs-1/
ls -la # Pas de dossier .git
git status # Erreur "fatal: not a git repository"
git init ./
ls -la # On a un dossier .git
git status # Succès avec un message "On branch main No commits yet"
```

🎓 Exercice avec Git - 1.2

- Créez un fichier `README.md` dedans avec un titre et vos nom et prénoms
  - Essayez la commande `git status` ?
- Ajoutez le fichier à la zone d'indexation à l'aide de la commande `git add (...)`
  - Essayez la commande `git status` ?
- Créez un commit qui ajoute le fichier `README.md` avec un message, à l'aide de la commande `git commit -m <message>`
  - Essayez la commande `git status` ?

✅ Solution de l'exercice avec Git - 1.2

```
echo "# Read Me\n\nObi Wan" > ./README.md
git status # Message "Untracked file"

git add ./README.md
git status # Message "Changes to be committed"
git commit -m "Ajout du README au projet"
git status  # Message "nothing to commit, working tree clean"
```

Terminologie de Git - Diff et changeset

diff: un ensemble de lignes "changées" sur un fichier donné

10 cluster/addons/node-problem-detector/npd.yaml

```diff
@@ -26,28 +26,28 @@ subjects:
  apiVersion: apps/v1
  kind: DaemonSet
  metadata:
-   name: npd-v0.8.0
+   name: npd-v0.8.5
    namespace: kube-system
    labels:
      k8s-app: node-problem-detector
-     version: v0.8.0
+     version: v0.8.5
      kubernetes.io/cluster-service: "true"
      addonmanager.kubernetes.io/mode: Reconcile
  spec:
    selector:
      matchLabels:
        k8s-app: node-problem-detector
-       version: v0.8.0
+       version: v0.8.5
    template:
      metadata:
        labels:
          k8s-app: node-problem-detector
-         version: v0.8.0
+         version: v0.8.5
          kubernetes.io/cluster-service: "true"
```

changeset: un ensemble de "diff" (donc peut couvrir plusieurs fichiers)

Showing 12 changed files with **314 additions** and **200 deletions**.

- 3 Jenkinsfile
- 10 make.ps1
- 456 tests/plugins-cli.Tests.ps1
- 2 tests/plugins-cli.bats
- 2 tests/plugins-cli/Dockerfile-windows
- 16 tests/plugins-cli/custom-war/Dockerfile-windows

Terminologie de Git - Commit

commit: un changeset qui possède un (commit) parent, associé à un message

"HEAD": C'est le dernier commit dans l'historique

🎓 Exercice avec Git - 2

- Afficher la liste des commits
- Afficher le changeset associé à un commit
- Modifier du contenu dans `README.md` et afficher le diff
- Annulez ce changement sur `README.md`

✅ Solution de l’exercice avec Git - 2

```
git log

git show # Show the "HEAD" commit
echo "# Read Me\n\nObi Wan Kenobi" > ./README.md

git diff
git status

git checkout -- README.md
git status
```

Terminologie de Git - Branche

- Abstraction d’une version "isolée" du code
- Concrètement, une **branche** est un alias pointant vers un "commit"

🎓 Exercice avec Git - 3

- Créer une branche nommée `feature/html`
- Ajouter un nouveau commit contenant un nouveau fichier `index.html` sur cette branche
- Afficher le graphe correspondant à cette branche avec `git log --graph`

✅ Solution de l'exercice avec Git - 3

```
git switch --create feature/html
# Ou alors: git branch feature/html && git switch feature/html
echo '<h1>Hello</h1>' > ./index.html
git add ./index.html && git commit --message="Ajout d'une page HTML par défaut" # -m / --message

git log
git log --graph
git lg # cat ~/.gitconfig => regardez la section section [alias], cette commande est déjà définie!
```

Terminologie de Git - Merge

- On intègre une branche dans une autre en effectuant un **merge**
  - Un nouveau commit est créé, fruit de la combinaison de 2 autres commits

🎓 Exercice avec Git - 4

- Merger la branche `feature/html` dans la branche principale
  - ⚠ Pensez à utiliser l'option `--no-ff`
- Afficher le graphe correspondant à cette branche avec `git log --graph`

✅ Solution de l’exercice avec Git - 4

```
git switch main
git merge --no-ff feature/html # Enregistrer puis fermer le fichier 'MERGE_MSG' qui a été ouvert
git log --graph

# git lg
```

Feature Branch Flow

- **Une seule** branche **par** fonctionnalité

Exemple d’usages de VCS

- "Infrastructure as Code" :
  - Besoins de traçabilité, de définition explicite et de gestion de conflits
  - Collaboration requise pour chaque changement (revue, responsabilités)
- Code Civil:
  - https://github.com/steeve/france.code-civil
  - https://github.com/steeve/france.code-civil/pull/40
  - https://github.com/steeve/france.code-civil/commit/b805ecf05a86162d149d3d182e04074ecf72c066

Checkpoint 🎯

- `git` est un des (plus populaires) de système de contrôle de versions
- Cet outil vous permet:
  - D'avoir un historique auditable de votre code source
  - De collaborer efficacement sur le code source (conflit `git` == "PARLEZ-VOUS")

  ⇒ C'est une ligne de commande (trop?) complète qui nécessite de pratiquer

Intégration Continue (CI)

> *Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.*

— Martin Fowler

Pourquoi la CI ?

But : Détecter les fautes au plus tôt pour en limiter le coût

Source : http://cartoontester.blogspot.be/2010/01/big-bugs.html

Qu’est ce que l’Intégration Continue ?

Objectif : que l’intégration de code soit un *non-évènement*

- Construire et intégrer le code **en continu**
- Le code est intégré **souvent** (au moins quotidiennement)
- Chaque intégration est validée par une exécution **automatisée**

Et concrètement ?

- Un•e développeu•se•r ajoute du code/branche/PR :
  - une requête HTTP est envoyée au système de "CI"
- Le système de CI compile et teste le code
- On ferme la boucle : Le résultat est renvoyé au développeu•se•r•s

Quelques moteurs de CI connus

- A héberger soit-même : Jenkins, GitLab, Drone CI, CDS…
- Hébergés en ligne : Travis CI, Semaphore CI, Circle CI, Codefresh, GitHub Actions

GitHub Actions

GitHub Actions est un moteur de CI/CD intégré à GitHub

- ✅ : Très facile à mettre en place, gratuit et intégré complètement
- ❌ : Utilisable uniquement avec GitHub, et DANS la plateforme GitHub

Anatomie de déclenchement de GitHub Actions

Concepts de GitHub Actions

Concepts de GitHub Actions - Step 1/2

Concepts de GitHub Actions - Step 2/2

Une **Step** (étape) est une tâche individuelle à faire effectuer par le CI :

- Par défaut c'est une commande à exécuter - mot clef `run`
- Ou une "action" (quel est le nom du produit déjà ?) - mot clef `uses`
  - Réutilisables et partageables

```yaml
steps: # Liste de steps
  # Exemple de step 1 (commande)
  - name: Say Hello
    run: echo "Hello ESGI"
  # Exemple de step 2 (une action)
  - name: 'Login to DockerHub'
    uses: docker/login-action@v2 # https://github.com/marketplace/actions/docker-login
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}
```

Copy

Concepts de GitHub Actions - Job 1/2

Concepts de GitHub Actions - Job 2/2

Un **Job** est un groupe logique de tâches :

- Enchaînement *séquentiel* de tâches
- Regroupement logique : "qui a un sens"
  - Exemple : "compiler puis tester le résultat de la compilation"

```yaml
jobs: # Map de jobs
  build: # 1er job, identifié comme 'build'
    name: 'Build Slides'
    runs-on: ubuntu-22.04 # cf. prochaine slide "Concepts de GitHub Actions - Runner"
    steps: # Collection de steps du job
      - name: 'Build the JAR'
        run: mvn package
      - name: 'Run Tests on the JAR file'
        run: mvn verify
  deploy: # 2nd job, identifié comme 'deploy'
    # ...
```

Copy

Concepts de GitHub Actions - Runner

Un **Runner** est un serveur distant sur lequel s'exécute un job.

- Mot clef `runs-on` dans la définition d'un job
- Défaut : machine virtuelle Ubuntu dans le cloud utilisé par GitHub
- D'autres types sont disponibles (macOS, Windows, etc.)
- Possibilité de fournir son propre serveur

Concepts de GitHub Actions - Workflow 1/2

Concepts de GitHub Actions - Workflow 2/2

Un **Workflow** est une procédure automatisée composée de plusieurs jobs, décrite par un fichier YAML.

- On parle de "Workflow/Pipeline as Code"
- Chemin : `.github/workflows/<nom du workflow>.yml`
- On peut avoir *plusieurs* fichiers donc *plusieurs* workflows

```
.github/workflows
├── ci-cd.yaml
├── bump-dependency.yml
└── nightly-tests.yaml
```

Copy

Concepts de GitHub Actions - Évènement 1/2

Concepts de GitHub Actions - Évènement 2/2

Un **évènement** du projet GitHub (push, merge, nouvelle issue, etc.) déclenche l'exécution du workflow

- Plein de type d'évènements : push, issue, alarme régulière, favori, fork, etc.
  - Exemple : "Nouveau commit poussé", "chaque dimanche à 07:00", "une issue a été ouverte" …
- Un workflow spécifie le(s) évènement(s) qui déclenche(nt) son exécution
  - Exemple : "exécuter le workflow lorsque un nouveau commit est poussé ou chaque jour à 05:00 par défaut"

Concepts de GitHub Actions : Exemple Complet

Workflow File :

Copy

```yaml
name: Node.js CI
on: # Évènements déclencheurs
  - push:
      branch: main # Lorsqu'un nouveau commit est poussé sur la branche "main"
  - schedule:
    - cron: "*/15 * * * *" # Toutes les 15 minutes
jobs:
  test-linux:
    runs-on: ubuntu-22.04
    steps:
    - uses: actions/checkout@v3
    - run: npm install
    - run: npm test
  test-mac:
    runs-on: macos-12
    steps:
    - uses: actions/checkout@v3
    - run: npm install
    - run: npm test
```

Essayons GitHub Actions

- **But** : nous allons créer notre premier workflow dans GitHub Actions
- N'hésitez pas à utiliser la documentation de GitHub Actions:
  - Accueil
  - Quickstart
  - Référence

🎓 Exercice: Créez un dépôt (git dans) GitHub

- En étant authentifié dans GitHub,
- Créez un nouveau dépôt nommé `esgi-ci-cd`
  - Pas de "template" (modèle)
  - Visibilité publique
  - Initialisation avec un fichier `README.md`

🎓 Exercice: Récupérez le dépôt dans Gitpod

- Obtenez l'URL (HTTPS) du dépôt GitHub fraîchement créé
  - 💡 Depuis la page du dépôt, cliquez sur le bouton vert intitulé "**Code**"
- Dans Gitpod :
  - Depuis un terminal, positionnez-vous dans le dossier `/workspace`,
  - Clonez le dépôt avec la commande

```
git clone https://github.com/xxx/{student_gh_repository_name}
```

  - 💡 Si le dépôt n'apparaît pas dans l'"Explorer" à gauche :

```
code -a /workspace/{student_gh_repository_name}
```

🎓 Exercice: Exemple simple avec GitHub Actions

- Dans le dépôt `esgi-ci-cd`, sur la branch `main`,
  - Créez le fichier `.github/workflows/bonjour.yml` avec le contenu suivant :

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - run: echo "Bonjour 👋"
```

- Commitez puis poussez le fichier sur GitHub:

```bash
git add .github/workflows/bonjour.yml
git commit -m 'Create GitHub workflow Bonjour'
git push origin main
```

🎓 Exercice: Exemple simple avec GitHub Actions : Récapépète

- Revenez sur la page GitHub de votre projet et naviguez dans l'onglet "Actions" :
  - Voyez-vous un workflow ? Et un Job ? Et le message affiché par la commande `echo` ?

Exemple GitHub Actions : Checkout

- Supposons que l'on souhaite utiliser le code du dépôt…
  - Essayez: modifiez le fichier `bonjour.yml` pour afficher le contenu de `README.md` :

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - run: ls -l # Liste les fichier du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

- Est-ce que l'étape "cat README.md" se passe bien ? (SPOILER: non ✖)

🎓 Exercice GitHub Actions : Checkout

- **But** : On souhaite récupérer ("checkout") le code du dépôt dans le job
- 👷🏽‍♀️ C’est à vous d’essayer de *réparer* 🛠️ le job :
  - L’étape “cat README.md” doit être conservée et doit fonctionner
  - Utilisez l’action "checkout" (Documentation) du marketplace GitHub Action
  - Vous pouvez vous inspirer du Quickstart de GitHub Actions

✅ Solution GitHub Actions : Checkout

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: ls -l # Liste les fichier du répertoire courant
      - run: cat README.md # Affiche le contenu du fichier `README.md` à la base du dépôt
```

Exemple : Environnement d'exécution

- Notre workflow doit s'assurer que "la vache" 🐮 doit nous lire 💬 le contenu du fichier `README.md`
  - WAT 🤪 ?
- Essayez la commande `cat README.md | cowsay` dans GitPod
  - Modifiez l'étape "cat README.md" du workflow pour faire la même chose dans GitHub Actions
  - SPOILER: ❌ (la commande `cowsay` n'est pas disponible dans le runner GitHub Actions)

🎓 Exercice : Personnalisation dans le workflow

- **But** : exécuter la commande `cat README.md | cowsay` dans le workflow comme dans GitPod
- 👷🏾‍♀️ C'est à vous de mettre à jour le workflow pour personnaliser l'environnement :
  - 💡 Cherchez comment installer `cowsay` dans le runner GitHub (`runs-on`…)

✅ Solution : Personnalisation dans le workflow

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: |
          sudo apt-get update
          sudo apt-get install --yes cowsay
      - run: cat README.md | cowsay
```

Checkpoint 🎯

- L’intégration Continue est un ensemble de pratiques pour s’assurer que le code est **continuellement** "vérifié"
- GitHub Actions est un des nombreux systèmes permettant de faire de l’intégration continue

⇒ 🤔 Problème : comment gérer les différences entre l’environnement d’IC et les machines de travail ?

Unresolved directive in index.adoc - include::./chapitres/ci-nodejs.adoc[leveloffset=1]

Plus de Git !

aka. Mettre son code en sécurité

🤔 Une petite histoire

Votre dépôt est actuellement sur votre ordinateur.

- Que se passe t'il si :
  - Votre disque dur tombe en panne ?
  - On vous vole votre ordinateur ?
  - Vous échappez votre tasse de thé / café sur votre ordinateur ?
  - Une météorite tombe sur votre bureau et fracasse votre ordinateur ?

Testé, pas approuvé.

Comment éviter ça ?

- Répliquer votre dépôt sur une ou plusieurs machines !
- Git est pensé pour gérer ce de problème

Gestion de version décentralisée

- Chaque utilisateur maintient une version du dépôt *local* qu'il peut changer à souhait
- Indépendant du commit, ils peuvent "pousser" une version sur un dépôt **distant**
- Un dépôt *local* peut avoir plusieurs dépôts **distants**.

Centralisé vs Décentralisé

Source Geek for Geeks

Cela rends la manipulation un peu plus complexe, allons-y pas à pas :-)

🎓 Consulter l'historique de commits

- Dans votre workspace GitPod,
- positionnez-vous dans le dépôt `esgi-ci-cd`,
- modifiez le fichier `README.md`, puis commitez,
- et enfin, affichez l'historique de commits.

✅ Consulter l'historique de commits

```
cd /workspace/esgi-devops-2013
echo "A new line" >> ./README.md
git add ./README.md
git commit -m "add a new line to the documentation"

# Liste tous les commits présent sur la branche main.
git log --graph
```

Associer un dépôt distant (1/2)

Git permet de manipuler des "remotes"

- Image "distante" (sur un autre ordinateur) de votre dépôt local.
- Permet de publier et de rapatrier des branches.
- Le serveur maintient sa propre arborescence de commits, tout comme votre dépôt local.
- Un dépôt peut posséder N remotes.

Associer un dépôt distant (2/2)

```
# Liste les remotes associés a votre dépôt
git remote -v
# Vous devriez voir l'URL de votre dépôt en tant que 'origin'

## La commande 'git clone' a effectué l'action ci-dessous pour vous :
# git remote add origin https://<URL de votre dépôt>
```

Copy

Publier une branche dans sur dépôt distant

Maintenant qu'on a un dépôt, il faut publier notre code dessus !

```
# git push <remote> <votre_branche_courante>
git push origin main
```

Copy

Que s’est il passé ?

- `git` a envoyé la branche `main` sur le remote `origin`
- … qui à accepté le changement et mis à jour sa propre branche main.
- `git` a créé localement une branche distante `origin/main` qui suis l'état de `main` sur le remote.
- Vous pouvez constater que la page github de votre dépôt affiche le code source

Refaisons un commit !

```
git commit --allow-empty -m "Yet another commit"
git push origin main
```

Copy

remotes/origin/main
main
Push
main
Local
Origin

Branche distante

Dans votre dépôt local, une branche "distante" est automatiquement maintenue par git

C’est une image du dernier état connu de la branche sur le remote.

Pour mettre a jour les branches distantes depuis le remote il faut utiliser :

```
git fetch <nom_du_remote>
```

```
# Lister toutes les branches y compris les branches distances
git branch -a


# Notez qu'est listé remotes/origin/main


# Mets a jour les branches distantes du remote origin
git fetch origin


# Rien ne se passe, votre dépôt est tout neuf, changeons ça!
```

🎓 Créez un commit depuis GitHub directement

- Cliquez sur le bouton éditer en haut à droite du "README"
- Changez le contenu de votre README
- Dans la section "Commit changes"
  - Ajoutez un titre de commit et une description
  - Cochez "Commit directly to the main branch"
  - Validez

GitHub crée directement un commit sur la branche main sur le dépôt distant

✅ Rapatrier les changements distants

```
# Mets à jour les branches distantes du dépôt origin
git fetch origin

# La branche distante main a avancé sur le remote origin
# => La branche remotes/origin/main est donc mise a jour

# Ouvrez votre README
code ./README.md

# Mystère, le fichier README ne contient pas vos derniers changements?
git log

# Votre nouveau commit n'est pas présent, AHA !
```

remotes/origin/main
main
Fetch
main
Local
Origin

Branche Distante VS Branche Locale

Le changement à été rapatrié, cependant il n'est pas encore présent sur votre branche main locale

```
# Merge la branch distante dans la branche locale.
git merge origin/main
```

Copy

Vu que votre branche main n'a pas divergé (== partage le même historique) de la branche distante, `git merge` effectue automatiquement un "fast forward".

```
Updating 1919673..b712a8e
Fast-forward
 README.md | 1 +

1 file changed, 1 insertion(+)
```

Cela signifie qu'il fait "avancer" la branche `main` sur le même commit que la branche `origin/main`

remotes/origin/main
main
Fetch
main
Local
Origin

```
# Liste l'historique de commit
git log

# Votre nouveau commit est présent sur la branche main !
# Juste au dessus de votre commit initial !
```

Copy

Et vous devriez voir votre changement dans le ficher README.md

Git(Hub|Lab|tea|…)

- Un dépôt distant peut être hébergé par n'importe quel serveur sans besoin autre qu'un accès SSH ou HTTPS.
- Une multitudes de services facilitent et enrichissent encore git: (GitHub, Gitlab, Gitea, Bitbucket…)

Checkpoint 🎯

⇒ git + Git(Hub|Lab|tea|…) = superpowers ! 🦄

- GUI de navigation dans le code
- Plateforme de gestion et suivi d'issues
- Plateforme de revue de code
- Integration aux moteurs de CI/CD
- And so much more…

Git à plusieurs

Limites de travailler seul

- Capacité finie de travail
- Victime de propres biais
- On ne sait pas tout

Travailler en équipe ? Une si bonne idée ?

- … Mais il faut communiquer ?
- … Mais tout le monde n'a pas les mêmes compétences ?
- … Mais tout le monde y code pas pareil ?

Collaborer c’est pas évident, mais il existe des outils et des méthodes pour vous aider.

Cela reste des outils, ça ne résous pas tout non plus.

Git multijoueur

- Git permet de collaborer assez aisément
- Chaque développeur crée et publie des commits…
- … et rapatrie ceux de de ses camarades !
- C'est un outil très flexible… chacun peut faire ce qu'il lui semble bon !

… et (souvent) ça finit comme ça !

Un Example de Git Flow

(Attachez vous aux idées générales… les détails varient d'un projet à l'autre!)

Gestion des branches

- Les "versions" du logiciel sont maintenues sur des branches principales (main, staging)
- Ces branches reflètent l'état du logiciel
  - **main**: version actuelle en production
  - **staging**: prochaine version

staging
main
Origin

Gestion des branches

- Chaque groupe de travail (développeur, binôme…)
  - Crée une branche de travail à partir de la branche staging
  - Une branche de travail correspond à **une chose à la fois**
  - Pousse des commits dessus qui implémentent le changement

feat/add-checkbox
staging
main
Origin

Quand le travail est fini, la branche de travail est "mergée" dans staging

v1.9.3
main
staging
Origin

Gestion des remotes

Où vivent ces branches ?

Plusieurs modèles possibles

- Un remote pour les gouverner tous !
- Chacun son propre remote (et les commits seront bien gardés)
- … whatever floats your boat!

Un remote pour les gouverner tous

Tous les développeurs envoient leur commits et branches sur le même remote

- Simple a gérer …
- … mais nécessite que tous les contributeurs aient accès au dépôt
  - Adapté a l'entreprise, peu adapté au monde de l'open source

https://github.com/torvalds/linux.git
upstream

Chacun son propre remote

- La motivation: le contrôle d'accès
  - Tout le monde peut lire le dépôt principal. Personne ne peut écrire dessus.
  - Tout le monde peut dupliquer le dépôt public et écrire sur sa copie.
  - Toute modification du dépôt principal passe par une procédure de revue.
  - Si la revue est validée, alors la branche est "mergée" dans la branche cible
- C'est le modèle poussé par GitHub !

https://github.com/torvalds/linux.git
upstream
Pull Request
https://github.com/jlevesy/linux.git
upstream
Torvalds
jlevesy

Forks ! Forks everywhere !

Dans la terminologie GitHub:

- Un fork est un remote copié d'un dépôt principal
  - C'est la où les contributeurs poussent leur branche de travail.
- Les branches de version (main, staging…) vivent sur le dépôt principal
- La procédure de ramener un changement d'un fork à un dépôt principal s'appelle la Pull Request (PR)

🎓 Exercice: Créez un fork

- Nous allons vous faire forker vos dépôts respectifs
- Trouvez vous un binôme dans le groupe et échangez vos URL de dépôts
- Depuis la page du dépôt de votre binôme, cliquez en haut à droite sur le bouton **Fork**.

🎓 À vous de jouer : dans le dépôt de votre binôme, vous allez ajouter un étape de "lint"

🎓 Exercice: Contribuez au projet de votre binôme (1/5)

Première étape: on clone le fork dans son environnement de développement

```
cd /workspace/

# Clonez votre fork
git clone <url_de_votre_fork>

# Se placer dans le fork
cd <nom_du_fork>

# Créez votre feature branch
git switch --create feat/hadolint
```

Copy

🎓 Exercice: Contribuez au projet de votre binôme (2/5)

Maintenant voici la liste des choses à faire:

- Ajoutez une étape de "lint" dans le workflow, avant de fabriquer l'image
  - La ligne de commande `hadolint` est pré-installée dans votre GitPod
  - GitHub action hadolint
- Corrigez /ignorez les erreurs du lint afin d'avoir un build qui passe

🎓 Exercice: Contribuez au projet de votre binôme (3/5)

Pour tester votre changement :

```
hadolint ./Dockerfile
```

Copy

🎓 Exercice: Contribuez au projet de votre binôme (4/5)

Une fois que vous êtes satisfaits de votre changement il vous faut maintenant créer un commit et pousser votre nouvelle branche sur votre fork.

🎓 Exercice: Contribuez au projet de votre binôme (5/5)

Dernière étape: ouvrir une pull request!

- Rendez vous sur la page de votre projet
- Sélectionnez votre branche dans le menu déroulant "branches" en haut a gauche.
- Cliquez ensuite sur le bouton ouvrir une pull request
- Remplissez le contenu de votre PR (titre, description, labels) et validez.

La procédure de Pull Request

Objectif : Valider les changements d'un contributeur

- Technique : est-ce que ça marche ? est-ce maintenable ?
- Fonctionnel : est-ce que le code fait ce que l'on veux ?
- Humain : Propager la connaissance par la revue de code.
- Méthode : Tracer les changements.

Revue de code ?

- Validation par un ou plusieurs pairs (technique et non technique) des changements
- Relecture depuis github (ou depuis le poste du développeur)
- Chaque relecteur émet des commentaires // suggestions de changement
- Quand un relecteur est satisfait d'un changement, il l'approuve

- La revue de code est un **exercice difficile** et **potentiellement frustrant** pour les deux parties.
  - Comme sur Twitter, on est bien à l'abri derrière son écran ;=)
- En tant que contributeur, **soyez respectueux** de vos relecteurs : votre changement peut être refusé et c'est quelque chose de normal.
- En tant que relecteur, **soyez respectueux** du travail effectué, même si celui ci comporte des erreurs ou ne correspond pas à vos attentes.

> 💡 Astuce: Proposez des solutions plutôt que simplement pointer les problèmes.

🎓 Exercice: Relisez votre PR reçue !

- Vous devriez avoir reçu une PR de votre binôme :-)
- Relisez le changement de la PR
- Effectuez quelques commentaires (bonus: utilisez la suggestion de changements), si c’est nécessaire
- Si elle vous convient, approuvez la!
- En revanche ne la "mergez" pas, car il manque quelque chose…

Validation automatisée

Objectif: Valider que le changement n'introduit pas de régressions dans le projet

- A chaque fois qu'un nouveau commit est créé dans une PR, une succession de validations ("checks") sont déclenchés par GitHub
- Effectue des vérifications automatisées sur un commit de merge entre votre branche cible et la branche de PR

Quelques exemples

- Analyse syntaxique du code (lint), pour détecter les erreurs potentielles ou les violations du guide de style
- Compilation du projet
- Execution des tests automatisés du projet
- Déploiement du projet dans un environnement de test…

Ces "checks" peuvent êtres exécutés par votre moteur de CI ou des outils externes.

🎓 Exercice: Déclencher un Workflow de CI sur une PR

- Votre PR n'a pas déclenché le workflow de CI de votre binôme 🤔
- Il faut modifier votre workflow pour qu'il se déclenche aussi sur une PR
- "Une chose à la fois" : faites une PR dédiée au déclenchement, puis mettez à jour la PR initial
- La documentation se trouve par ici

Checkpoint 🎯

Règle d'or: Si le CI est rouge, on ne merge pas la pull request !

Même si le linter "ilécon", même si on a la flemme et "sépanou" qui avons cassé le CI.

Versions

Pourquoi faire des versions ?

- Un changement visible d'un logiciel peut nécessiter une adaptation de ses utilisateurs
- Un humain ça s'adapte, mais un logiciel il faut l'adapter!
- Cela permet de contrôler le problème de la compatibilité entre deux logiciels.

Une petite histoire

Le logiciel que vous développez utilise des données d'une API d'un site de vente.

```
// Corps de la réponse d'une requête GET https://supersite.com/api/item
[
  {
    "identifier": 1343,
    // ...
  }
]
```

Copy

Voici comment est représenté un item vendu dans votre code.

```
public class Item {
  // Identifiant de l'item représenté sous forme d'entier.
  private int identifier;
  // ...
}
```

Copy

Le site décide tout d'un coup de changer le format de l'identifiant de son objet en chaîne de caractères.

```
// Corps de la réponse d'une requête GET https://supersite.com/api/item
[
  {
    "identifier": "lolilol3843",
    // ...
  }
]
```

Copy

Que se passe t'il du côté de votre application ?

com.fasterxml.jackson.databind.JsonMappingException

Que s’est-il passé ?

- Votre application ne s’attendait pas à un identifiant sous forme de chaîne de caractères !
- Le fournisseur de l’API à "changé le contrat" de son API d’une façon non rétrocompatible avec votre l’existant.
  - Cela s’appelle un 🇬🇧 **Breaking Change**

Comment éviter cela ?

- Laisser aux utilisateurs une marge de manœuvre pour "accepter" votre changement.
  - Donner une garantie de maintien des contrats existants.
  - Informer vos utilisateurs d’un changement non rétrocompatible.
  - Anticiper les changements non rétrocompatibles à l’aide de stratégies (dépréciation).

Bonjour versions !

- Une version cristallise un contrat respecté par votre application.
- C’est un jalon dans l’histoire de votre logiciel

Quoi versionner ?

Le problème de la compatibilité existe dès qu'une dépendance entre deux bouts de code existe.

- Une API
- Une librairie
- Un langage de programmation
- Le noyau linux

Version sémantique

La norme est l'utilisation du format vX.Y.Z (Majeur.Mineur.Patch)

(source betterprograming)

Un changement **ne changeant pas le périmètre fonctionnel** incrémente le numéro de version **patch**.

Un changement changeant le périmètre fonctionnel de façon **rétrocompatible** incrémente le numéro de version **mineure**.

Un changement changeant le périmètre fonctionnel de façon **non rétrocompatible** incrémente le numéro de version **majeure**.

En résumé

- Changer de version mineure ne devrait avoir aucun d'impact sur votre code.
- Changer de version majeure peut nécessiter des adaptations.

Concrètement avec une API

- Offrir a l'utilisateur un moyen d'indiquer la version de l'API a laquelle il souhaite parler
  - Via un préfixe dans le chemin de la requête:
    - `https://monsupersite.com/api/v2.3/item`
  - Via un en-tête HTTP:
    - `Accept-version: v2.3`

Version VS Git

- Un identifiant de commit est de granularité trop faible pour un l'utilisateur externe.
- Utilisation de **tags** git pour définir des versions.
- Un **tag** git est une référence sur un commit.

🎓 🐳 Docker Hub

Si vous n’avez pas déjà un compte sur le DockerHub, créez-en un maintenant (nécessite une validation)

🎓 "Taguez" et déployez la version 1.0.0 de `cowsay`

- Depuis GitPod, créez un tag git local `1.0.0`
  - 💡 `git tag 1.0.0 -a -m "Première release 1.0.0, mode manuel"`
- Fabriquez l'image Docker de `cowsay` avec le tag (Docker) 1.0.0
  - 💡 `docker image build --tag=XXXX/cowsay:1.0.0 #`... avec `XXXX` étant votre username DockerHub
- Déployez l'image sur le DockerHub
  - 💡 `docker login`, `docker image push`
- Déployez le tag sur GitHub
  - 💡 `git push origin <tagname>`

✅ "Taguez" et déployez la version 1.0.0 de `cowsay`

```
DOCKER_USERNAME=dduportal
TAG=1.0.0

git tag "${TAG}" -a -m "Première release ${TAG}, mode manuel"

docker image build --tag="${DOCKER_USERNAME}"/cowsay:"${TAG}" ./
docker login --username="${DOCKER_USERNAME}"
# ...

docker image push "${DOCKER_USERNAME}"/cowsay:"${TAG}"

git push origin "${TAG}"

# Check DockerHub et GitHub après ça
```

Checkpoint 🎯

- La notion de "version" est un outil de communication aux consommateurs de nos produits logiciels
- Le "semantic versioning" est une des façon les plus usitées pour gérer les politiques de version
- Nous avons déployé manuellement notre première image Docker, avec synchronisation code source <→ image Docker

⇒ 🤔 C'était fort manuel. Et si on regardait à automatiser tout ça ?

"Continuous Everything"

Livraison Continue

Continuous Delivery (CD)

Pourquoi la Livraison Continue ?

- Diminuer les risque liés au déploiement
- Permettre de récolter des retours utilisateurs plus souvent
- Rendre l'avancement visible par **tous**

> *How long would it take to your organization to deploy a change that involves just one single line of code?*

— Mary and Tom Poppendieck

Qu’est ce que la Livraison Continue ?

- Suite logique de l’intégration continue:
  - Chaque changement est **potentiellement** déployable en production
  - Le déploiement peut donc être effectué à **tout** moment

> *Your team prioritizes keeping the software **deployable** over working on new features*

— Martin Fowler

La livraison continue est l'exercice de **mettre à disposition automatiquement** le produit logiciel pour qu'il soit prêt à être déployé à tout moment.

Livraison Continue avec GitHub Actions

Pré-requis: exécution conditionnelle des jobs

- Il est possible d'exécuter conditionnellement un job ou un step à l'aide du mot clé `if` (documentation de `if`)

```yaml
jobs:
  release:
    # Lance le job release uniquement si la branche est main.
    if: contains('refs/heads/main', github.ref)
    steps:
      # ...
```

Copy

🎓 Secret GitHub / DockerHub Token

- Créez un personal access token dans le DockerHub avec les droits "Read,Write"
  - ⚠ ne partagez pas ce token, ne le mettez PAS dans GitPod, ne l'enregistrez pas sur votre disque dur
  - 💡 Documentation "Manage access tokens"
- Insérez le token DockerHub comme secret dans votre dépôt GitHub
  - 💡 Creating encrypted secrets for a repository

🎓 Livraison Continue sur le DockerHub

- **But :** Automatiser le déploiement de l'image dans le DockerHub lorsqu'un tag est poussé
- Changez votre workflow de CI de façon à ce que, sur un push de tag, les tâches suivantes soient effectuées :
  - Comme avant: Lint, Build, Test de l'image (en affichant le `README`)
  - SI c'est un tag, alors il faut pousser (et éventuellement reconstruire avec le bon nom) l'image sur le DockerHub
- 💡 Utilisez les GitHub Action suivantes :
  - docker-login
  - build-and-push-docker-images

✅ Livraison Continue sur le DockerHub

```
Unresolved directive in chapitres/continuous-delivery.adoc - include::../code-samples/gh-actions/ci-docker-push-ta
```

Copy

Déploiement Continu

🇬🇧 Continuous Deployment / "CD"

Qu’est ce que le Déploiement Continu ?

- Version "avancée" de la livraison continue:
  - Chaque changement **est** déployé en production, de manière **automatique**

Continuous Delivery versus Deployment

Source : http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

Bénéfices du Déploiement Continu

- Rends triviales les procédures de mise en production et de rollback
  - Encourage à mettre en production le plus souvent possible
  - Encourage à faire des mises en production incrémentales
- Limite les risques d'erreur lors de la mise en production
- Fonctionne de 1 à 1000 serveurs et plus encore…

Qu’est ce que "La production" ?

- Un (ou plusieurs) ordinateur ou votre / vos applications sont exécutées
- Ce sont là où vos utilisateurs "utilisent" votre code
  - Que ce soit un serveur web pour une application web
  - Ou un téléphone pour une application mobile
- Certaines plateformes sont plus ou moins outillées pour la mise en production automatique

🎓 Déploiement Continu sur le DockerHub

- **But :** Déployer votre image `cowsay` continuellement sur le DockerHub
- Changez votre workflow de CI de façon à ce que, sur un push sur la branch `main`, les tâches suivantes soient effectuées :
  - Comme avant: Lint, Build, Test de l'image (en affichant le `README`)
  - SI c'est la branche `main`, alors il faut pousser l'image avec le tag `latest` sur le DockerHub
  - Conservez le cas avec les tags

✅ Déploiement Continu sur le DockerHub

```
Unresolved directive in chapitres/continuous-delivery.adoc - include::../code-samples/gh-actions/ci-docker-push-ma
```

Copy

Checkpoint 🎯

- La livraison continue et le déploiement continu étendent les concepts du CI
- Les 2 sont automatisées, mais un être humain est nécessaire comme déclencheur pour la 1ère
- Le choix dépends des risques et de la "production"
- On a vu comment automatiser le déploiement dans GitHub Actions
  - Conditions dans le workflow
  - Gestion de secrets

Bibliographie

Ligne de commande

- https://tldp.org
- https://en.wikipedia.org/wiki/POSIX
- https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop
- https://linuxhandbook.com/linux-directory-structure/

Git / VCS

- https://docs.github.com
- https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
- http://martinfowler.com/bliki/VersionControlTools.html
- http://martinfowler.com/bliki/FeatureBranch.html
- https://about.gitlab.com/2014/09/29/gitlab-flow/
- https://www.atlassian.com/git/tutorials/comparing-workflows
- http://nvie.com/posts/a-successful-git-branching-model/

Intégration Continue

- http://martinfowler.com/articles/continuousIntegration.html
- http://martinfowler.com/bliki/ContinuousDelivery.html
- https://jaxenter.com/implementing-continuous-delivery-117916.html
- https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/
- http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model
- http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

Docker

- https://dduportal.github.io/cours/cnam-docker-2018
- https://kodekloud.com/blog/docker-for-beginners/
- https://www.slideshare.net/dotCloud/why-docker
- https://docs.docker.com/engine/reference/builder/
- https://www.r-bloggers.com/2021/05/best-practices-for-r-with-docker/
- https://github.com/wagoodman/dive
- https://docs.docker.com/engine/tutorials/networkingcontainers/
- https://towardsdatascience.com/docker-networking-919461b7f498

Livraison/Déploiement Continu

- http://martinfowler.com/articles/continuousIntegration.html
- http://martinfowler.com/bliki/ContinuousDelivery.html
- https://jaxenter.com/implementing-continuous-delivery-117916.html
- https://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/
- http://blog.arungupta.me/continuous-integration-delivery-deployment-maturity-model
- http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

Autre

- https://cicd-lectures.github.io/slides/
- https://dduportal.github.io/cours/

Merci !

damien.duportal @ gmail.com

Slides: https://dduportal.github.io/dduportal/esgi-courses/2023-S1-M2-IW

Source on : https://github.com/dduportal/dduportal/esgi-courses