Introduction au CI/CD

ENSG - Décembre 2023

- Présentation disponible à l'adresse: https://cicd-lectures.github.io/slides/2023
- Version PDF de la présentation : Cliquez ici
- This work is licensed under a Creative Commons Attribution 4.0 International License
- Code source de la présentation: https://github.com/cicd-lectures/slides

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 : utilisez la touche "o" (pour "**Overview**")
- Pour voir les notes de l’auteur : utilisez la touche "s" (pour "**Speaker** notes")

Bonjour !

Damien DUPORTAL

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

Julien LEVESY

- Senior Platform Engineer @ Voi 🛴
- Me contacter :
  - jlevesy <chez> gmail.com
  - Julien Levesy
  - @jlevesy

Et vous ?

A propos du cours

- Alternance de théorie et de pratique pour être le plus interactif possible
- Reproductible à la maison, pensé dans le contexte du "Covid à la maison"
- Contenu entièrement libre et open-source
  - N'hésitez pas ouvrir des Pull Request si vous voyez des améliorations ou problèmes: sur cette page (😉 wink wink)

Une petite histoire du génie logiciel

Comment mener un projet logiciel?

Avant : le cycle en V

Que peut-il mal se passer?

- On spécifie et l'on engage un volume conséquent de travail sur des hypothèses
  - … et si les hypothèses sont fausses?
  - … et si les besoins changent?
- Cycle trèèèèès long
  - Aucune validation à court terme
  - Coût de l'erreur décuplé

Comment éviter ça?

- Valider les hypothèses au plus tôt, et étendre petit à petit le périmètre fonctionnel.
  - Réduire le périmètre fonctionnel au minimum.
  - 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**

La clé : gérer le changement!

- Le changement ne doit pas être un événement, ça doit être la norme.
- Notre objectif : minimiser le coût du changement.
- Faire en sorte que:
  - Changer quelque chose soit facile
  - Changer quelque chose soit rapide
  - Changer quelque chose ne casse pas tout

Heureusement, vous avez des outils à disposition!

Et c’est ce que l’on va voir ensemble pendant les trois prochains jours!

Préparer votre environnement de travail

Outils Nécessaires 🛠️

- Un navigateur récent (et décent)
- Un compte sur GitHub
- Un compte sur Google
- On va vous demander de travailler en binôme, commencez à réfléchir avec qui vous souhaitez travailler !
- Enregistrez vous par ici!

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 (⚠)
- Open-Source : vous pouvez l'héberger chez vous

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

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

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/cicd-lectures/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 !

Guide de survie en ligne de commande

Remise à niveau / Rappels

CLI

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

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>` :

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

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 !

- `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

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

- 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/`)

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 $?
```

Copy

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
```

Copy

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)"
```

Copy

- 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

Les fondamentaux de git

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
- Pour **collaborer** efficacement sur un même référentiel de code source

Local Computer
Checkout
File
Version Database
Version 3
Version 2
Version 1

Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Computer B
File
Version Database

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/
    ```

    Copy

  - 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 : 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 master No commits yet"
```

Copy

🎓 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 : 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"
```

Copy

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

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 : 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 : avec Git - 3

```
git branch feature/html && git switch feature/html
# Ou git switch --create 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!
```

Copy

- On intègre une branche dans une autre en effectuant un **merge**
- Plusieurs strategies sont possibles pour merger:
  - Quand l'historique de commit n'a pas diverge: git fait avancer la branche directement, c'est un **fast-forward**
  - Dans le cas contraire, 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` (no fast forward) pour forcer git a créer un commit de merge.
- Afficher le graphe correspondant à cette branche avec `git log --graph`

✅ Solution : 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
```

Copy

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

On a vu :

- A quoi sert `git` et sa nomenclature de base (diff, changest, commit, branch)
- A quoi reconnaître un dépôt initialisé local et l'espace de travail associé
- Comment utiliser git localement (ajouter au staging, commiter)
- l'historique et un merge avec git (localement)

Présentation de votre projet

Contexte (1/2)

- Dans un effort d'optimisation de ses processus métiers, la cantina de Mos-Estafette s'est lancée dans un grand projet de digitalisation de la gestion de sa carte
- Le projet est divisé en deux lots:
  - Une partie serveur qui stocke et expose la données de la carte via une API REST
  - Une partie client qui permet de consulter et de gérer les menus de la cantina via l'API du serveur

Contexte (2/2)

- Après une longue et épique bataille de chefs de projets et autres décideurs fonctionnels, la société **Bananes Services d'Entreprises**, spécialiste national des technologies du minitel, à remporté le lot de la partie serveur…
- …Cependant, suite à d'un **grave accident de brainstorming** entre les 18 chefs de projet travaillant sur le dit projet et 6 mois de retard sur la première livraison, l'entreprise **Bananes** s'est retrouvée dans l'incapacité de mener le projet à son terme
- Désemparés face à cette terrible nouvelle, le conseil d'administration de la cantina à décidé de se tourner vers l'ENSG pour tenter de sauver leur transition numérique

Prise en Main du Projet (1/2)

- Une équipe technique de **Bananes** avait commencé l'implémentation du serveur, et à fourni une archive téléchargeable ICI, contenant le code source du projet
- **Bananes** vous assure qu'ils ont suivi toutes les "best practices" du développement logiciel sur minitel
  - Il y à un `LISEZMOI.txt` à la racine du projet :tada:
  - … et pas grand chose d'autre?

Prise en Main du Projet (2/2)

```
# Création du répertoire menu-server
mkdir -p /workspace/menu-server && cd /workspace/menu-server

# Téléchargez le projet sur votre environnement de développement
curl -sSLO https://cicd-lectures.github.io/slides/2023/media/menu-server.tar.gz

# Décompresser et extraire l'archive téléchargée
tar xvzf ./menu-server.tar.gz
```

Copy

A partir de la vous pouvez ouvrir le fichier `LISEZMOI.txt` et commencer à suivre ses instructions.

Comment accéder à un serveur démarré dans un Gitpod ?

- Vous avez probablement démarré un serveur web qui écoute sur le port 8080
- Si vous souhaitez accéder par `curl` depuis terminal de votre gitpod, rien à faire
- Si vous souhaitez y accéder depuis l'extérieur de votre gitpod il vous faut une URL publique:
  - `gp url 8080` vous indique l'URL du port 8080 de votre instance gitpod!
  - `gp preview "$(gp url 8080)"` ouvrir votre navigateur.

Qu’est-ce qui va / ne va pas dans ce projet d’après vous?

Triste Rencontre avec la Réalité

- Pas de gestion de version…
- Le projet ne fonctionne pas, tous les menus retournés s’appellent "TODO" :sob:
- Le correctif ne semble pas compliqué à faire…
- … sauf que vous ne pouvez pas compiler le projet!

Il va falloir remédier à ça d’une façon ou d’une autre, sinon vous n’allez pas aller bien loin!

Architecture du projet

HTTP Layer

- Fait la transition entre une requête HTTP et une action logique de l’application
- Accepte en entrée une requête HTTP (headers, query parameters, verb, (JSON) body)
- Réponds un code de statut, des headers et optionnellement un body
- Appelle la couche service en passant des DTOs (data transfer objects)

Business Layer

- Implémente la logique métier de l’application
- Accepte en entrée des DTOs et en réponds en sortie
- Appelle d’autre services ou les repositories de l’application en leur passant des objets model

Storage Layer

- Fait la transition entre le modèle objet java et le modèle de la base de données (ici relationnel // SQL)
- Accepte en entrée des objets du modèle, et peut en répondre aussi
- Chaque méthode implémentée par un repository est en réalité une requête à votre base de données
- Implémentation ici déléguée à un outil d'ORM (object relational mapping), ici hibernate
- On utilisera dans le cadre de ce cours une base de donnée in-memory appelée h2

Architecture: resume

Modèle de données

- Une entité `Menu`, portant un identifiant et un titre
- Une entité `Dish`, portant un identifiant et un titre
- Un `Menu` est composé de 0 à N `Dish`

Organisation du Code Source

Toutes les classes sont créées dans un sous répertoire de
`src/main/java/com/cicdlectures/menuserver`

Le découpage est le suivant:

- `controller`: Couche d'interconnection entre HTTP et le domaine métier
- `dto` (Data Transfer Objects): Représentation intermédiaire de la donnée
- `service`: Couche métier
- `repository`: Couche d'accès a la base de données
- `model`: Représentation de votre modèle de donnée, configuration de l'ORM.

🎓 Exercice : Initialisez un dépôt git

- Nettoyez le contenu superflu du projet et initialisez un dépôt git dans le répertoire, puis créez un premier commit
- Par contenu superflu, nous entendons:
  - Tout ce qui est potentiellement généré
  - Les scripts de lancement obsolètes et inutiles
  - Un petit renommage du LISEZMOI.txt en README.md et un coup de nettoyage de son contenu

💡 Pour chaque "popup" de l'éditeur (en bas à droite), choisissez "Oui" ou "Reload"

✅ Solution Exercice

Copy

```
# On évacue le contenu inutile
rm -rf dist/ menu-server.tar.gz
rm executer.sh
# On renomme LISEZMOI.txt en README.md
mv LISEZMOI.txt README.md
# On nettoie son contenu
code README.md

# On initialise un nouveau dépôt git
git init

# On ajoute tous les fichiers contenus a la zone de staging.
git add .

# On crée un nouveau commit
git commit -m "Add initial menu-server project files"
```

Checkpoint 🎯

- Vous avez récupéré un projet Java qui semble fonctionner…
  - ..mais pas vraiment à l'état de l'art.
- Application du chapitre précédent : vous avez initialisé un projet `git` local

Cycle de vie de votre projet

🤔 Quel est le problème ?

On a du code. C’est un bon début. MAIS:

- Qu’est ce qu’on "fabrique" à partir du code ?
- Comment faire pour "fabriquer" de la même manière pour tout•e•s (💻 | 🖥️) ?

Que "fabrique" t’on à partir du code ? 📦

Un **livrable** :

- C’est ce que vos utilisateurs vont utiliser: un binaire à télécharger ? L’application de production ?
- C’est versionné
- C’est *reproductible*

Reproduire la fabrication 🏭

Comment fabriquer et tester de manière reproductible ?

- Dimension technique: type de language, différents OS, différentes machines
- Dimension temporelle: version de dépendances qui changent dans le temps
- Dimension humaine: habitudes, connaissances, documentation (ou pas)

(Mauvais) Exemple

Que pensez-vous de :

```
# Compilation
javac -d ./target \
    ./src/main/java/com/cicdlectures/menuserver/MenuServerApplication.java \
    ./src/main/java/com/cicdlectures/menuserver/*.java \
    ./src/main/java/com/cicdlectures/menuserver/controller/*.java \
    ./src/main/java/com/cicdlectures/menuserver/dto/*.java \
    ./src/main/java/com/cicdlectures/menuserver/model/*.java \
    ./src/main/java/com/cicdlectures/menuserver/repository/*.java \
    ./src/main/java/com/cicdlectures/menuserver/service/*.java
```

```
# Exécution
java -cp target/classes/com/cicdlectures/menuserver/ \
  -cp target/classes/com/cicdlectures/menuserver/controller/ \
  -cp target/classes/com/cicdlectures/menuserver/dto/ \
  -cp target/classes/com/cicdlectures/menuserver/model/ \
  -cp target/classes/com/cicdlectures/menuserver/repository/ \
  -cp target/classes/com/cicdlectures/menuserver/service/ \
MenuServerApplication
```

Résultat(s) :

```
./src/main/java/com/cicdlectures/menuserver/dto/DishDto.java:16: error: cannot find symbol
@NoArgsConstructor(access = AccessLevel.PRIVATE)
```

Copy

```
Error: Could not find or load main class MenuServerApplication
Caused by: java.lang.ClassNotFoundException: MenuServerApplication
```

Copy

Maven

Say Hello to "Maven":

- Idée de Maven : **Cycles de vie** standardisés
- "Convention over configuration" : fichier `pom.xml` décrivant le projet
- Ligne de commande `mvn` qui lit le `pom.xml` et exécute les **phases**

Maven : Cycles de vie

- 3 cycles de vie :
  - 🏗️ `default` : pour construire les applications dans `target/`
  - 📝 `site` : pour construire un site web de documentation dans `target/`
  - 🧹 `clean` : nettoyer `target/`
- Chaque cycle de vie est composé d'une série d'étapes appelées "phases"

Maven : Phases

Phases du cycle de vie 🏗️ `default` :

- `validate` - Validation du projet (syntaxe du `pom.xml` et du Java, etc.)
- `compile` - Fabrication des fichiers `.class` depuis le code java
- `test` - Exécuter les tests unitaires
- `package` - Préparer le livrable finale (`.jar` par exemple)
- `verify` - Exécuter les tests d'intégration
- `install` - Mettre à disposition le livrable localement pour d'autres projets Maven
- `deploy` - Copier le livrable dans un système de stockage de dépendance distant

Maven : `pom.xml`

- "POM" signifie "Project Object Model"
- Contenu en XML : language de type "markup", avec un **schéma**, donc strict
- "Convention au lieu de configuration" pour limiter la complexité
  - code source attendu dans `src/main/java/`
  - résultats dans `target/` (transient), etc.
  - `pom.xml` à la racine de votre projet

Maven : Ligne de commande `mvn`

Ligne de commande `mvn` :

- Lit le fichier `pom.xml` pour comprendre le projet
- Attend une (ou plusieurs) phases en argument
- Accepte des options (formes courtes `-X` ou longues `--debug`)

```
# Exemples :
mvn clean # Appelle la phase "clean"
mvn compile # Appelle les phases "validate" puis "compile"
mvn clean compile -X # On peut appeler plusieurs phases et passer des options
```

Copy

🎓 Exercice Maven : C'est à vous !

But : fabriquer l'application menuserver avec Maven

Open in Gitpod

Commençons par valider le projet en utilisant la phase `validate` de Maven:

```
mvn validate
```

Copy

```
# ...
[ERROR] The goal you specified requires a project to execute but there is no POM in this directory (/workspace/menu-serve
# ...
```

Copy

✖ Il manque un fichier `pom.xml` !

🎓 Exercice Maven : fichier `pom.xml`

- Commençons par créer un fichier `pom.xml` avec le contenu ci-dessous :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!-- Insert content here  -->
</project>
```

- Puis ré-essayons de valider le projet avec Maven :

```
mvn validate
```

```
# ...
[ERROR] [ERROR] Some problems were encountered while processing the POMs:
[FATAL] 'groupId' is missing. @ line 2, column 102
[FATAL] 'artifactId' is missing. @ line 2, column 102
[FATAL] 'version' is missing. @ line 2, column 102
# ...
```

Maven : identité d'un projet

Maven identifie un projet avec les 3 éléments **obligatoires** suivants :

- **groupId** : Identifiant unique de votre projet suivant les règles Java de nommage de paquets
- **artifactId** : Identifiant du projet (paquet de la classe principale)
- **version** : Version de l'artefact

```
<!-- Exemple avec un paquet Java `com.cicdlectures.menuserver` dans `src/main/java/com/cicdlectures/menuserver` -->
<groupId>com.cicdlectures</groupId>
<artifactId>menuserver</artifactId>
<version>1.0-SNAPSHOT</version>
```

Copy

🎓 Exercice Maven : identifiez votre projet

⇒ C'est à vous

- Identifiez votre projet en remplissant le fichier `pom.xml`
  - `groupId` et `artifactId`: utilisez le nom de package de votre classe principale `MenuServerApplication.java`
  - `version:1.0-SNAPSHOT`
- **Objectif :** Maven doit valider le projet avec succès :

```
mvn validate
```

Copy

```
# ...
[INFO] BUILD SUCCESS
# ...
```

Copy

Checkpoint 🎯

- On a pu créer un fichier `pom.xml` valide ✅
- Pensez à commiter ce changement 💡
- Il est temps de compiler l'application avec Maven 🏗️

🎓 Exercice Maven : Compiler 🏗️

- Essayez de compiler l'application à l'aide de la phase `compile` de Maven:

```
mvn compile
```

Copy

- Résultat attendu : Message `[INFO] BUILD FAILURE` ❌

Analyse des erreurs de compilation 🏗️

Que s'est il passé ?

1. ⇒ Maven a téléchargé plein de dépendances depuis https://repo.maven.apache.org
2. ⇒ La compilation a échoué avec plein d'erreurs et quelques "warning" :

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile (default-compile)
      on project my-app: Compilation failure: Compilation failure:
[ERROR] <...>/src/main/java/com/cicdlectures/menuserver/repository/MenuRepository.java:[3,43]
      package org.springframework.data.repository does not exist
```

Copy

Maven et Dépendances Externes

- Maven propose 2 types de dépendances externes :
  - **Plugin** : c’est un artefact qui sera utilisé par Maven durant son cycle de vie
    - "Build-time dependency"
  - **Dépendance** (🇬🇧 "dependency") : c’est un artefact qui sera utilisé par votre application, *en dehors de Maven*
    - "Run-time dependency"

Maven et Plugins

Quand on regarde sous le capot, Maven est un framework d'exécution de plugins.

⇒ Tout est plugin :

- Effacer le dossier `target` ? Un plugin ! (si si essayez `mvn clean` une première fois…)
- Compiler du Java ? Un plugin !
- Pas de plugin qui fait ce que vous voulez ? Écrivez un autre plugin !

C’est bien gentil mais comment corriger notre erreur ?

💡 Il manque des dépendances pour compiler :

- ✗ `package org.springframework.data.repository does not exist`
- ✗ `package jakarta.persistence does not exist`
- ✗ `package lombok does not exist`

Dépendances Externes

Hypothèse : on a besoin de code et d'outils externes (e.g. écrits par quelqu'un d'autre)

- Comment faire si le code externe est mis à jour ?
- Que se passe t'il si le code externe est supprimé de l'internet ?
  - https://github.blog/2020-11-16-standing-up-for-developers-youtube-dl-is-back/
- Acceptez-vous d'exécuter le code de quelqu'un d'autre sur votre machine ?
- Et si quelqu'un injecte du code malicieux dans le code externe ?
  - https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/

TOUS les languages…

... sont concernés

Maven : Dépôts d'Artefacts

Maven récupère les dépendances (et plugins) dans des dépôts d'artefacts

(🇬🇧 Artifacts Repositories) qui sont de 3 types :

- **Central** : un dépôt géré par la communauté - https://repo.maven.apache.org
  - Avec une interface web de recherche
- **Remote** : des dépôts privés de votre organisation
- **Local** : un dossier sur la machine où la commande `mvn` est exécuté, généralement dans `${HOME}/.m2`

? 💡 `mvn install` cible ce dépôt "local"

Dépendances Maven

💡 "Introduction au mécanisme de dépendances - documentation Maven

- Pour spécifier les dépendances (dans votre `pom.xml`):
  - Il faut utiliser la balise `<dependencies>`,
  - … qui est une collection de dépendances (balise `<dependency>` - quelle surprise !),
  - .. chaque dépendance étant défini par un trio `<groupId>`, `<artifactId>` et `<version>` (que de surprises…)
- Pour les plugins c'est la même idée (`<plugins>` → `<plugin>` → `<groupId>`, `<artifactId>`, `<version>`)

Exemple de Dépendance : Spring

- **Idée** : Nous avons besoin d'ajouter le framework Spring en dépendance.

Voilà ce que ça donne dans le fichier `pom.xml` :

Copy

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>3.0.2</version>
  </dependency>
</dependencies>
```

🎓 Exercice avec les dépendances Spring 1/2

⇒ C'est à vous.

- Ajoutez le bloc `<dependencies>` de la slide précédente dans votre `pom.xml`
  - 💡 org.springframework.boot.spring-boot-starter-web 3.0.2 sur Maven Central
- Exécutez la commande `mvn compile`
- Résultat attendu :
  - ✅ L'erreur `package org.springframework.data.repository does not exist` a disparu: la dépendance est présente
  - ❌ Encore d'autre dépendances manquantes (`jakarta.persistence`, `lombok`, etc.)

🎓 Exercice avec les dépendances Spring 2/2

- **But:** Compiler l'application complète
- Continuez de modifier le fichier `pom.xml` afin d'ajouter les 2 dépendances suivantes :
  - Lombok: org.projectlombok.lombok 1.18.26 sur Maven Central
  - Jakarta persistence: org.springframework.boot.spring-boot-starter-data-jpa 3.0.2 sur Maven Central
- Résultat attendu : ✅ `[INFO] BUILD SUCCESS`

✅ Solution avec les dépendances Spring

Copy

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
    <groupId>com.cicdlectures</groupId>
    <artifactId>menuserver</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
        <version>3.0.2</version>
      </dependency>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>3.0.2</version>
      </dependency>
      <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.26</version>
      </dependency>
    </dependencies>
</project>
```

Exécution de l'application Spring : Tentative 1

- Quel est le contenu de `target/` ? Et de `target/classes` ?

```
# 💡 Chercher tous les fichier dans le sous-dossier ./target/classes
find ./target/classes -type f
```

- Vous avez un état stable : pensez à `git` :
  - Dites à `git` d'ignorer le dossier `target/` (fichier `.gitignore`)
  - Ajoutez et commitez dans `git`

```
echo 'target/' >> .gitignore
git add .gitignore pom.xml
git commit -m "Ajout de la configuration pour compiler l'application"
```

Checkpoint 🎯

- C'est la galère pour trouver les bonnes dépendances 🤔
- `mvn compile` a produit des fichiers dans `target/classes/**` ✅
- Il faut encore pouvoir exécuter l'application

⇒ reprenons en lisant le documentation

- Spring Boot est bien plus simple à utiliser que ce que l'on a vu 🧸 !
  - On l'a abordé ainsi pour mieux comprendre
- Une documentation très complète :
  - "Get Started" pour bien démarrer
  - Spring Initialzr pour générer son `pom.xml` en ligne
  - Une documentation de référence
- Un plugin Maven est fourni par le projet Spring Boot pour se simplifier la vie:
  - Pas besoin de répéter les versions
  - Plein de fonctionnalités de développement

Maven Plugins

Un plugin Maven implémente les tâches à effectuer durant les différentes phases, et peut appartenir à l'un ou l'autre de ces 2 types :

- **"Build"** : Implémente une action durant les phase du cycle de vie `default`, et est configuré dans la balise `<build>`
- **"Reporting"** Implémente une action durant les phases du cycle de vie `site`, et est configuré dans la balise `<reporting>` (à votre grande surprise)

C'est un fichier `*.jar` identifié par… groupId, artifactId et version.

Plugin Maven Spring Boot

- On vous fournit le contenu du `pom.xml` (slide suivante) généré (et adapté) depuis Spring Initialzr, avec les changements suivants :
  - Ajout d'un POM "parent" (dont on hérite) venant de Spring Boot (Éviter la répétition)
  - Configuration avec des properties (clef/valeurs)
  - Mise à jour des dépendances (ajouts et simplification des versions, déléguées au "POM parent")
  - Activation du plugin Spring Boot lors des phases de "build"

Copy

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.cicdlectures</groupId>
  <artifactId>menuserver</artifactId>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
  </parent>

  <properties>
    <java.version>17</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
```

🎓 Exercice : Démarrer l'application - 1/2

- **But**: Exécuter l'application à l'aide du plugin Spring Boot
- Sprint Boot fournit une phase Maven nommée `spring-boot:run` qui exécute l'application en mode "développement" sur le port `8080` local
  - Essayez d'appeler cette phase avec Maven
- Résultat attendu (une jolie bannière ASCIIArt):

Copy

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.2)
```

🎓 Exercice : Démarrer l'application - 2/2

- Dans un second terminal de Gitpod, affichez la page web de l'application avec les commandes suivantes :
  - `gp url 8080` pour afficher l'URL publique de l'application correspondant au port `8080` local
  - `gp preview "$(gp url 8080)/"` pour prévisualiser le "endpoint" `/` dans un navigateur local
- La page d'accueil doit afficher HTTP/404
- Trouvez la page des menus qui doit répondre `[]` (liste vide en JSON)

? 💡 `src/*/controller/*.java`

Checkpoint 🎯

- Spring Boot (et toutes ses dépendances) est configuré ✅
- Le plugin Maven Spring Boot permet de compiler et d'exécuter l'application avec la commande `mvn spring-boot:run` ✅
- Il faut encore fabriquer un fichier JAR 🏺 pour la production 🤔

⇒ reprenons avec Maven

🎓 Exercice : Maven JAR Plugin

- **But**: Produire l'artefact JAR distribuable
- La génération du JAR est déclenchée lors de l'appel à `mvn package` :

```
ls -ltra ./target
mvn clean # Nettoyez tout !
ls -ltra ./target
mvn package
ls -ltra ./target # Est-ce que vous voyez un fichier JAR ?
```

- Exécution de l'application :

```
java -jar <chemin vers le fichier JAR>
```

  - Même fonctionnement que précédemment (bannière, port 8080, endpoint `/menus`…)

🎓 Exercice : Changer le nom de l'artefact final

- **But**: Produire un artefact JAR dont le nom est `menu-server.jar`
- Quel est le nom de l'artefact généré ? Est-il constant ?
  - (SPOILER: 🙅🏽‍♀️)
- En utilisant la documentation de référence https://maven.apache.org/pom.html#the-basebuild-element-set, adaptez votre `pom.xml` afin que le fichier généré se nomme **toujours** `menu-server.jar`.

✅ Solution : Changer le nom de l'artefact final

```
<build>
  <finalName>menu-server</finalName>
  <!-- ... <plugins> ... -->
</build>
```

Copy

💡 Pensez à ajouter / commiter quand c’est fonctionnel pour vous !

Copy

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.cicdlectures</groupId>
  <artifactId>menuserver</artifactId>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
  </parent>

  <properties>
    <java.version>17</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
```

9.43

Checkpoint 🎯

- Le projet Menu Server est configuré avec Maven (`pom.xml`) ✅
- On peut vérifier l'application localement avec la commande `mvn spring-boot:run` ✅
- L'application est fabriquée avec la commande `mvn package` qui produit le délivrable `./target/menu-server.jar` ✅

Mettre son code en sécurité

Une autre 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**.

Centralise vs Decentralise

Source Geek for Geeks

Créer un dépôt distant

- Rendez vous sur GitHub
  - Créez un nouveau dépôt distant en cliquant sur "New" en haut à gauche
  - Appelez le `menu-server`
  - Une fois créé, mémorisez l'URL (https://github.com/...) de votre dépôt :-)
  - Inscrivez l'URL de votre depot ici.

Consulter l'historique de commits

Dans votre workspace

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

Copy

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

# Ajoute votre dépôt comme remote appelé `origin`
git remote add origin https://<URL de votre dépôt>

# Vérifiez que votre nouveau remote `origin` est bien listé a la bonne adresse
git remote -v
```

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 que 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!
```

Copy

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 !
```

Copy

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(+)
```

Copy

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…)

⇒ Dans le cadre du cours, nous allons utiliser **GitHub**.

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…

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 ? 1/2

- 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

Et concrètement ? 2/2

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

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

Copy

```yaml
steps: # Liste de steps
  # Exemple de step 1 (commande)
  - name: Say Hello
    run: echo "Hello ENSG"
  # Exemple de step 2 (une action)
  - name: 'Login to DockerHub'
    uses: docker/login-action@v1 # https://github.com/marketplace/actions/docker-login
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}
```

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 :

```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
```

Copy

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
- Retournez dans le dépôt créé précédemment dans votre environnement GitPod

Exemple simple avec GitHub Actions

- Dans le projet "menu-server", 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
- 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 simple avec GitHub Actions : Récapépète

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 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 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

Copy

```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 du workflow pour faire la même chose dans GitHub Actions
    - SPOILER: ❌ (la commande `cowsay` n'est pas disponible dans le runner GitHub Actions)

Problème : Environnement d'exécution

- **Problème** : On souhaite utiliser les mêmes outils dans notre workflow ainsi que dans nos environnement de développement
- Plusieurs solutions existent pour personnaliser l'outillage, chacune avec ses avantages / inconvénients :
  - Personnaliser l'environnement dans votre workflow: (⚠ sensible aux mises à jour, ✅ facile à mettre en place)
  - Spécifier un environnement préfabriqué pour le workflow (⚠ complexe, ✅ portable)
  - Utiliser les fonctionnalités de votre outil de CI (⚠ spécifique au moteur de CI, ✅ efficacité)

🎓 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`, paquet `cowsay` dans Ubuntu 22.04)

✅ 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 -y cowsay
      - run: cat README.md | cowsay
```

Copy

🎓 Exercice : Environnement préfabriqué

- **But** : exécuter la commande `cat README.md | cowsay` dans le workflow comme dans GitPod
  - En utilisant le même environnement que GitPod (même version de cowsay, java, etc.)
- 👷‍♀️ C'est à vous de mettre à jour le workflow pour exécuter les étapes dans la même image Docker que GitPod :
  - 💡 Image utilisée dans GitPod
  - 💡 Utilisation d'un container comme runner GitHub Actions
  - 💡 Contraintes d'exécution de container dans GitHub Actions (`--user=root`)

✅ Solution : Environnement préfabriqué

Copy

```yaml
name: Bonjour
on:
  - push
jobs:
  dire_bonjour:
    runs-on: ubuntu-22.04
    container:
      image: ghcr.io/cicd-lectures/gitpod:latest
      options: --user=root
    steps:
      - uses: actions/checkout@v3 # Récupère le contenu du dépôt correspondant au commit du workflow en cours
      - run: cat README.md | cowsay
```

Checkpoint 🎯

- Quel est l'impact en terme de temps d'exécution du changement précédent ?
- **Problème :** Le temps entre une modification et le retour est crucial

du moteur de CI

- **But** : s'assurer que GitHub actions install et utilise `cowsay` le plus efficacement possible
- C'est à vous de mettre à jour le workflow pour:
  - Lire le contenu du fichier `README.md` dans un "output" (une variable temporaire de GitHub Actions)
  - Passer le contenu (via l'output) à une version de cowsay gérée par GitHub Actions
- 💡 Utilisez les GitHub Actions et documentations suivantes :
  - GitHub Action pour cowsay
  - GitHub Action pour lire un fichier dans une variable `output`

✅ Solution : Optimiser avec les fonctionnalités du moteur de CI

Copy

```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
      - uses: juliangruber/read-file-action@v1
        id: readfile
        with:
          path: ./README.md
      - uses: Code-Hex/neo-cowsay-action@v1
        with:
          message: "${{ steps.readfile.outputs.content }}"
```

11.33

"menu-server"

👷🏾‍♀️ C’est à vous de modifier le projet "menu-server" pour faire l’intégration continue, afin qu’à chaque commit poussé sur votre dépôt, un workflow GitHub Actions va :

- Récupérer le code de l’application depuis GitHub
- S’assurer d’utiliser les **même** versions de Java et Maven que dans Gitpod (💡 `mvn -v`)
  - 💡 https://github.com/marketplace/actions/setup-java-jdk
  - 💡 https://github.com/marketplace/actions/setup-maven
- L’application est compilée
- Le `jar` de l’application est fabriqué

"menu-server"

Copy

```yaml
name: Menu Server CI
on:
  - push
jobs:
  menu_server:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3
      - name: Checkout Code
        uses: actions/checkout@v3
      - name: Setup JDK Zulu 17
        uses: actions/setup-java@v3
        with:
          distribution: 'zulu'
          java-version: '17'
      - name: Setup Maven 3.9.0
        uses: stCarolas/setup-maven@v4.5
        with:
          maven-version: 3.9.0
      - name: Check Maven tooling
        run: mvn -v
      - name: Build application
```

🎯 Checkpoint

- Pour chaque commit poussé dans la branche `main` de Menu Server,
- GitHub action vérifie que l'application est compilable et fabriquée,
- Avec un feedback (notification GitHub).

⇒ On peut modifier notre code avec plus de confiance !

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 !

BLOING!
BLOING!

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

Origin

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

main
staging

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
origin
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)

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

A vous de jouer: Ajoutez la fonctionnalité "suppression d’un menu" au projet de votre binôme

🎓 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>

# Créez votre feature branch
git switch --create implement-delete
# Équivalent de git checkout -b <...>
```

Copy

binôme (2/5)

Maintenant voici la liste des choses à faire:

- Rajouter le `MenuRepository` comme dépendance du MenuController
- Implémenter une nouvelle méthode `deleteMenu`
  - Gère les requêtes `DELETE /menus/{id}`
  - Appelle la méthode `deleteById` du `menuRepository`
  - Réponds 200 si la suppression est réussie
- Bonus si vous arrivez à faire en sorte que le serveur réponde 404 si un menu à supprimer n'existe pas.

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

Pour tester votre changement

```
# D'abord on crée un menu
curl -H "Content-Type: application/json" --data-raw '{"name": "Menu spécial du chef", "dishes": [{"name": "Bananes aux fr
```

```
# Puis on le supprime
curl -XDELETE localhost:8080/menus/4
```

```
# Et on vérifie que le menu est bien supprimé
curl localhost:8080/menus
```

🎓 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.

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 changer la spec de votre workflow pour qu'il se déclenche aussi sur une PR
- Vous pouvez changer la spec du workflow directement dans votre PR
- 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.

Tests Automatisés

Qu’est ce qu’un test ?

C’est du code qui vérifie que votre code fait ce qu’il est supposé faire.

Pourquoi faire des tests ?

- Prouve que le logiciel se comporte comme attendu a tout moment.
- Détecte les impacts non anticipés des changements introduits
- Evite l'introduction de régressions
- Écrire des tests est un acte préventif et non curatif.

Qu’est ce que l’on teste ?

- Une fonction
- Une combinaison de classes
- Un serveur applicatif et une base de données

On parle de **SUT**, System Under Test.

Différents systèmes, Différentes Techniques de Tests

- Test unitaire
- Test d'integration
- Test de bout en bout
- Smoke tests
- Test de performance

(La terminologie varie d'un développeur / langage / entreprise / écosystème à l'autre)

Test unitaire

- Test validant le bon comportement une unité de code.
- Prouve que l'unité de code interagit correctement avec les autres unités.
- Par exemple :
  - Retourne les bonnes valeur en fonction des paramètres donnés
  - Appelle la bonne méthode du bon attribut avec les bons paramètres

Mise en place de l'exercice

- Depuis votre environnement de développement, dans le repertoire du **fork** de votre binôme
- Créez une feature branch `add-tests`.

Ajout des Outils de Tests Unitaires au Projet (1/3)

L'execution de tests nécessite un outillage non ajouté au projet

- Framework d'écriture et d'execution de tests: `JUnit`
- Librairie de création de mocks: `Mockito`
- Plugin maven de lancement de tests unitaires: `surefire`

Cycle de Vie Maven et Tests

- Rappel : Maven définit un cycle de vie de votre logiciel par phases:
- `surefire` exécute les tests contenus dans tous les fichiers ayant le suffixe `Tests.java` (`MaClasseTests.java`)
- Le plugin `surefire` s'exécute (par défaut) lors de la phase `tests`
- Lancer `mvn test` va exécuter les phases `validate` → `compile` → `test`

Ajout des Outils de Tests Unitaires au Projet (2/3)

- Ajoutez la dépendance suivante dans votre `pom.xml` (section `<dependencies>`):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

- Puis configurez le plugin Maven Spring Boot pour les tests unitaires (section `<build>` `<plugins>`):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <skipTests>${skipUnitTests}</skipTests>
  </configuration>
```

Ajout des Outils de Tests Automatisés au Projet (3/3)

- Exécutez les tests unitaires avec la commande `mvn test`
  - Spoiler : `No tests to run`...
  - Pourquoi ca ?

🎓 Exercice : Corriger un Bug (1/11)

- La classe `ListMenuService` semble être "buggée"…
  - Tous les noms des menus sont **TEST TODO** 😱
- Quand on regarde l'implémentation, on se rends compte que le problème provient de la méthode statique `fromModel` de la classe `MenuDto`
- Même si la correction est aisée, on va d'abord écrire un test unitaire qui valide le comportement du service.
- Notre SUT: `ListMenuService` + `DTO` + `Model`

🎓 Exercice : Corriger un Bug (2/11)

Mise en place du test

```java
// src/test/java/com/cicdlectures/menuserver/service/ListMenuServiceTests.java

public class ListMenuServiceTests {

  private ListMenuService subject;

  @BeforeEach
  public void init() {
    subject = new ListMenuService(null);
  }

  @Test
  @DisplayName("lists all known menus")
  public void listsKnownMenus() {
     List<MenuDto> got = subject.listMenus();
  }
}
```

Copy

🎓 Exercice : Corriger un Bug (3/11)

- Super on à un test, il ne reste plus qu'à le lancer avec `mvn test` 🎉
- Spoiler `java.lang.NullPointerException`

🎓 Exercice : Corriger un Bug (4/11)

- Le `ListMenuService` à besoin d'un `MenuRepository` pour fonctionner.
- Cependant :
  - On ne veut pas valider le comportement du `MenuRepository`, il est en dehors de notre SUT.
  - Pire, on ne veut pas se connecter à une base de donnée pendant un test unitaire.

🎓 Exercice : Corriger un Bug (5/11)

Solution : On fournit une "fausse implémentation" au service, un mock.

Copy

```
// src/test/java/com/cicdlectures/menuserver/service/ListMenuServiceTests.java

private MenuRepository menuRepository;

private ListMenuService subject;

@BeforeEach
public void init() {
  this.menuRepository = mock(MenuRepository.class);
  this.subject = new ListMenuService(this.menuRepository);
}
```

🎓 Exercice : Corriger un Bug (6/11)

Ce "mock" peut être piloté dans les tests!

```
@Test
@DisplayName("lists all known menus")
public void listsKnownMenus() {
  // Quand le repository reçoit l'appel findAll
  // Alors il retourne la valeur null.
  when(menuRepository.findAll()).thenReturn(null);
}
```

Copy

🎓 Exercice : Corriger un Bug (7/11)

- Super on a un test unitaire, il ne reste plus qu'à le lancer avec `mvn test` 🎉
- Spoiler: ✅

Sauf qu’on avait pas un bug à corriger au fait?

🎓 Exercice : Corriger un Bug (8/11)

Objectif: Vérifier que les valeurs retournées par le `ListMenuService` sont cohérentes avec les données en base, pour cela il nous faut:

- Préparer un jeu de données de test et configurer le mock du repository pour qu'il le retourne
- Appeler notre service
- Comparer le résultat obtenu du service avec des valeurs attendues.

```java
@Test
@DisplayName("lists all known menus")
public void listsKnownMenus() {
  // Défini une liste de menus avec un menus.
  Iterable<Menu> existingMenus = Arrays.asList(
    new Menu(
      Long.valueOf(1),
      "Christmas menu",
      new HashSet<>(
        Arrays.asList(
          new Dish(Long.valueOf(1), "Turkey", null),
          new Dish(Long.valueOf(2), "Pecan Pie", null)
        )
      )
    )
  );

  // On configure le menuRepository pour qu'il retourne notre liste de menus.
  when(menuRepository.findAll()).thenReturn(existingMenus);

  // On appelle notre sujet
  List<MenuDto> gotMenus = subject.listMenus();

  // On défini wantMenus, les résultats attendus
  Iterable<MenuDto> wantMenus = Arrays.asList(
      new MenuDto(
        Long.valueOf(1),
```

Copy

🎓 Exercice : Corriger un Bug (10/11)

- Super on a un test unitaire (qui teste!), il ne reste plus qu'à le lancer avec `mvn test` 🎉
- Spoiler:

```
[ERROR] Failures:
[ERROR]   ListMenuServiceTests.listsKnownMenus:66
expected:
  <[MenuDto(id=1, name=Christmas menu, dishes=[DishDto(id=2, name=Pecan Pie), DishDto(id=1, name=Turkey)])]>
but was:
  <[MenuDto(id=1, name=TEST TODO, dishes=[DishDto(id=2, name=Pecan Pie), DishDto(id=1, name=Turkey)])]>
```

Copy

- Il ne reste plus qu’a faire la correction et le tour est joué!

Test Unitaire : Quelques Règles

- Un test unitaire teste un et un seul comportement
- Faites attention a ce que votre test teste vraiment quelque chose!
  - Avec les mocks, c'est facile de se faire piéger.
- Essayez, dans la mesure du possible, d'écrire vos tests (qui échouent) avant d'écrire votre code.
- Il n'y a pas de définition ferme du SUT
  - Attention à garder une taille raisonnable (quelques classes).
  - Privilégiez les tests de méthodes publiques.

Checkpoint 🎯

On a vu :

- 🔍 Qu'il faut tester son code
- 🌍 Qu'il existe différents type de tests en fonction de ce que l'on veut tester
- 🧩 Comment faire des tests unitaires

Test Unitaire : Pro / Cons

- ✅ Super rapides (<1s) et légers a executer
- ✅ Pousse à avoir un bon design de code
- ✅ Efficaces pour tester des cas limites
- ❌ Peu réalistes

Tester des composants indépendamment ne prouve pas que le système fonctionne une fois intégré!

✅ Solution: Tests d'intégration

- Test validant qu'un assemblage d'unités se comportent comme prévu.
- Teste votre application au travers de toutes ses couches
- Par exemple avec menu server:
  - Prouve que GET /menus retourne la liste des menus enregistrés en base
  - Prouve que POST /menus enregistre un nouveau menu en base avec ses plats.

Définition du SUT (1/2)

Définition du SUT (2/2)

Une suite de tests d'intégration doit:

- Démarrer et provisionner un environnement d'exécution (une DB, Elasticsearch, un autre service…)
- Démarrer votre application
- Jouer un scénario de test
- Éteindre et nettoyer son environnement d'exécution pour garantir l'isolation des tests

Ce sont des tests plus lents et plus complexes que des tests unitaires. Comment gérer ça?

Exécuter Les Tests d'Intégration: Cycle de Vie Maven

- Les tests d'intégration sont une autre partie du cycle de vie de l'application: la phase `verify`.
- `verify` est une méta-phase composée de 3 sous-phases :
  - `pre-integration-test`: prépare l'environnement des tests d'intégration
  - `integration-test`: execute la suite de tests d'intégration
  - `post-integration-test`: nettoie l'environnement des tests d'intégration

⚠ Il faut toujours appeler `verify` et non pas `integration-test`, sinon la sous-phase `post-integration-test` ne s'exécutera pas ⚠

Exécuter Les Tests d'Intégration: Le Plugin `failsafe` (1/3)

- Pour exécuter les tests d'intégration nous allons introduire un nouveau plugin: `failsafe`
- Ce plugin exécute les tests ayant le suffixe `IT.java` (par exemple: `MaClasseIT.java`)
- Ce plugin s'exécute lors de la phase `integration-test`

Exécuter Les Tests d'Intégration: Le Plugin `failsafe` (2/3)

- Configurez le plugin Maven Spring Boot pour les tests d'intégration (section `<build>` `<plugins>`):

Copy

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <configuration>
    <skipTests>${skipIntegrationTests}</skipTests>
  </configuration>
</plugin>
```

Exécuter Les Tests d'Intégration: Le Plugin `failsafe` (2/3)

Cela crée les commandes suivantes:

- `mvn test`: lance les tests unitaires
- `mvn verify`: lance les tests unitaires et d'intégration
- `mvn verify -DskipUnitTests=true`: lance uniquement les tests d'intégration

Tests d'Intégrations: Et concrètement avec le menu-server?

- Dans les faits… nous n'allons pas utiliser les phases `pre-integration-test` et `post-integration-test`
  - → Nous n'avons pas de serveur de base de données a démarrer.
  - → SpringBoot intègre le démarrage et l'arrêt du serveur web dans l'exécution des tests via l'annotation `@SprintBootTest`.
- C'est un projet pédagogique!
  - Dans un "vrai" projet, on voudrait peut-être démarrer / éteindre un serveur de base de données dans ces étapes.

Nous allons écrire un test d'intégration pour l'appel `GET /menus`

🎓 Exercice : Ecrire un test d'integration (1/4)

Mise en place d'un test vide

Copy

```
// src/test/java/com/cicdlectures/menuserver/controller/MenuControllerIT.java
// Lance l'application sur un port aléatoire.
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
// Indique de relancer l'application à chaque test.
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
public class MenuControllerIT {

  @LocalServerPort
  private int port;

  private URL getMenusURL() throws Exception {
    return new URL("http://localhost:" + port + "/menus");
  }

  @Test
  @DisplayName("lists all known menus")
  public void listsAllMenus() throws Exception {
  }
}
```

🎓 Exercice : Ecrire un test d'integration (2/4)

Maintenant, on appelle le serveur et on verifie que l'appelle qu'il nous reponds une 200

Copy

```java
// src/test/java/com/cicdlectures/menuserver/controller/MenuControllerIT.java
// Lance l'application sur un port aléatoire.
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
// Indique de relancer l'application à chaque test.
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
public class MenuControllerIT {
  // ...

  @Test
  @DisplayName("lists all known menus")
  public void listsAllMenus() throws Exception {
    ResponseEntity<MenuDto[]> response = this.template.getForEntity(getMenusURL().toString(), MenuDto[].class);

    assertEquals(HttpStatus.OK, response.getStatusCode());
  }
}
```

Bon, c’est bien sympa mais notre test n’est pas satisfaisant en l’etat. Il faut maintenat valider notre comportement pricipal: lister tous les menus connus

🎓 Exercice : Ecrire un test d'integration (3/4)

D'abord il faut provisionner des données en base.

Copy

```
public class MenuControllerIT {
  // ...
  // Injecte automatiquement l'instance du menu repository
  @Autowired
  private MenuRepository menuRepository;

  private final List<Menu> existingMenus = Arrays.asList(
      new Menu(null, "Christmas menu", new HashSet<>(Arrays.asList(new Dish(null, "Turkey", null), new Dish(null, "Pecan
      new Menu(null, "New year's eve menu", new HashSet<>(Arrays.asList(new Dish(null, "Potatos", null), new Dish(null, "


  @BeforeEach
  public void initDataset() {
    for (Menu menu : existingMenus) {
      menuRepository.save(menu);
    }
  }

  // ...
}
```

13.43

Il ne nous reste qu'a changer le corps du test pour verifier que le contenu de la reponse est celui auquel on s'attends.

Copy

```java
public class MenuControllerIT {
  // ...

  @Test
  @DisplayName("lists all known menus")
  public void listsAllMenus() throws Exception {
    // On declare la valeur attendue.
    MenuDto[] wantMenus = {
        new MenuDto(Long.valueOf(1), "Christmas menu",
            new HashSet<DishDto>(
                Arrays.asList(new DishDto(Long.valueOf(1), "Turkey"), new DishDto(Long.valueOf(2), "Pecan Pie")))),
        new MenuDto(Long.valueOf(2), "New year's eve menu", new HashSet<DishDto>(
            Arrays.asList(new DishDto(Long.valueOf(3), "Potatos"), new DishDto(Long.valueOf(4), "Tiramisu")))) };

    // On fait la requête et on recupere la reponse.
    ResponseEntity<MenuDto[]> response = this.template.getForEntity(getMenusURL().toString(), MenuDto[].class);

    // On verifie le status de reponse.
    assertEquals(HttpStatus.OK, response.getStatusCode());

    // On list le corps de la reponse.
    MenuDto[] gotMenus = response.getBody();
```

🎓 Exercice: Activez les tests dans votre CI

Changez le workflow de ci de votre binôme (ou le votre) pour qu'à chaque build:

- Les tests unitaires soient lancés
- Les tests d'integrations soient lancés

💡 Pensez à bien regarder le cycle de vie des phases Maven

Checkpoint 🎯

On a vu :

- ❌ Les limites des tests unitaires
- 🏭 Comment faire des tests d'intégration
- 🤔 Tester n'est pas facile mais très utile

Docker

Remise à niveau / Rappels

🤔 Quel est le problème ?

	Development VM	QA Server	Single Prod Server	Production Cluster	Public cloud	Developer's Laptop	Customer Servers
Static Website	?	?	?	?	?	?	?
Web Frontend	?	?	?	?	?	?	?
Background Workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Source: https://blog.docker.com/2013/08/paas-present-and-future/

Déjà vu ?

L’IT n’est pas la seule industrie à résoudre des problèmes…

✅ Solution: Le conteneur intermodal

"Separation of Concerns"

Comment ça marche ?

"Virtualisation **Légère**"

Conteneur != VM

"Separation of concerns": 1 "tâche" par conteneur

VMs && Conteneurs

Non exclusifs mutuellement

🎓 Exercice : où est mon conteneur ?

- Retournez dans Gitpod
- Dans un terminal, exécutez les commandes suivantes :

```
# Affichez la liste de tous les conteneurs en fonctionnement (aucun)
docker container ls

# Exécutez un conteneur
docker container run hello-world # Equivalent de l'ancienne commande 'docker run'


docker container ls
docker container ls --all
# Quelles différences ?
```

Copy

🩻 Anatomie

- Un service "Docker Engine" tourne en tâche de fond et publie une API REST
- La commande `docker run` ... a envoyé une requête `POST` au service
- Le service a télécharge une **Image** Docker depuis le registre **DockerHub**,
- Puis a exécuté un **conteneur** basé sur cette image

✅ Solution : Où est mon conteneur ?

Le conteneur est toujours présent dans le "Docker Engine" même en étant arrêté

```
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS     NAMES
109a9cdd3ec8   hello-world   "/hello"   33 seconds ago   Exited (0) 17 seconds ago             festive_faraday
```

Copy

- Un conteneur == une commande "conteneurisée"
  - cf. colonne "**COMMAND**"
- Quand la commande s’arrête : le conteneur s’arrête
  - cf. code de sortie dans la colonne "**STATUS**"

🏋 Exercice : Cycle de vie d'un conteneur en tâche de fond

- Lancez un nouveau conteneur en tâche de fond, nommé `webserver-1` et basé sur l'image `nginx`
  - 💡 `docker container run --help` ou Documentation en ligne
- Affichez les "logs" du conteneur (==traces d'exécution écrites sur le stdout + stderr de la commande conteneurisée)
  - 💡 `docker container logs --help` ou Documentation en ligne
- Comparez les versions de Linux de Gitpod et du conteneur

? Regardez le contenu du fichier `/etc/os-release`

✅ Solution : Cycle de vie d'un conteneur en tâche de fond

```
docker container run --detach --name=webserver-1 nginx
# <ID du conteneur>

docker container ls

docker container logs webserver-1

cat /etc/os-release
# ... Ubuntu ...
docker container exec webserver-1 cat /etc/os-release
# ... Debian ...
```

Copy

🤔 Comment accéder au serveur web en tâche de fond ?

```
$ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS     NAMES
ee5b70fa72c3   nginx     "/docker-entrypoint.…"   3 seconds ago   Up 2 seconds   80/tcp    webserver-1
```

- ✅ Super, le port 80 (TCP) est annoncé (on parle d'"exposé")…
- ❌ … mais c'est sur une adresse IP privée

```
docker container inspect \
  --format='{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
  webserver-1
```

🎓 Exercice : Accéder au serveur web via un port publié

- **But :** Créez un nouveau conteneur `webserver-public` accessible publiquement
- Utilisez le port `8080` publique
- 💡 Flag `--publish` pour `docker container run`
- 💡 GitPod va vous proposer un popup : choisissez "Open Browser"

✅ Solution : Accéder au serveur web via un port publié

```
docker container run --detach --name=webserver-public --publish 8080:80 nginx
# ... container ID ...

docker container ls
# Le port 8080 de 0.0.0.0 est mappé sur le 80 du conteneur

curl http://localhost:8080
# ...
```

Copy

🤔 D’où vient "hello-world" ?

- Docker Hub (https://hub.docker.com) : C’est le registre d’images "par défaut"
  - Exemple : Image officielle de "nginx"
- 🎓 Cherchez l’image `hello-world` pour en voir la page de documentation
  - 💡 pas besoin de créer de compte pour ça
- Il existe d’autre "registres" en fonction des besoins (GitHub GHCR, Google GCR, etc.)

🤔 Que contient "hello-world" ?

- C’est une "image" de conteneur, c’est à dire un modèle (template) représentant une application auto-suffisante.
  - On peut voir ça comme un "paquetage" autonome
- C’est un système de fichier complet:
  - Il y a au moins une racine `/`
  - Ne contient que ce qui est censé être nécessaire (dépendances, librairies, binaires, etc.)

🤔 Pourquoi des images ?

- Un **conteneur** est toujours exécuté depuis une **image**.
- Une **image de conteneur** (ou "Image Docker") est un modèle ("template") d'application auto-suffisant.

⇒ Permet de fournir un livrable portable (ou presque).

🤔 Application Auto-Suffisante ?

C’est quoi le principe ?

🤔 Pourquoi fabriquer sa propre image ?

Essayez ces commandes dans Gitpod :

```
cat /etc/os-release
# ...
git --version
# ...

# Même version de Linux que dans GitPod
docker container run --rm ubuntu:20.04 git --version
# docker: Error response from daemon: failed to create shim task: OCI runtime create failed: runc create failed: unable t

# En interactif ?
docker container run --rm --tty --interactive ubuntu:20.04 git --version
```

Copy

⇒ Problème : git n'est même pas présent !

🎓 Fabriquer sa première image

- **But :** fabriquer une image Docker qui contient `git`
- Dans votre workspace Gitpod, créez un nouveau dossier `/workspace/docker-git/`
- Dans ce dossier, créer un fichier `Dockerfile` avec le contenu ci-dessous :

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
```

Copy

- Fabriquez votre image avec la commande `docker image build --tag=docker-git <chemin/vers/docker-git/`
- Testez l'image fraîchement fabriquée

? 💡 `docker image ls`

✅ Fabriquer sa première image

```
mkdir -p /workspace/docker-git/ && cd /workspace/docker-git/

cat <<EOF >Dockerfile
FROM ubuntu:20.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
EOF

docker image build --tag=docker-git ./

docker image ls | grep docker-git

# Doit fonctionner
docker container run --rm docker-git:latest git --version
```

Copy

Conventions de nommage des images

```
[REGISTRY/][NAMESPACE/]NAME[:TAG|@DIGEST]
```

Copy

- Pas de Registre ? Défaut: `registry.docker.com`
- Pas de Namespace ? Défaut: `library`
- Pas de tag ? Valeur par défaut: `latest`
  - ⚠ Friends don't let friends use `latest`
- Digest: signature unique basée sur le contenu

Conventions de nommage : Exemples

- `ubuntu:20.04` ⇒ `registry.docker.com/library/ubuntu:20.04`
- `dduportal/docker-asciidoctor` ⇒ `registry.docker.com/dduportal/docker-asciidoctor:latest`
- `ghcr.io/dduportal/docker-asciidoctor:1.3.2@sha256:xxxx`

🎓 Utilisons les tags

- Il est temps de "taguer" votre première image !

```
docker image tag docker-git:latest docker-git:1.0.0
```

Copy

- Testez le fonctionnement avec le nouveau tag
- Comparez les 2 images dans la sortie de `docker image ls`

✅ Utilisons les tags

```
docker image tag docker-git:latest docker-git:1.0.0

# 2 lignes
docker image ls | grep docker-git
# 1 ligne
docker image ls | grep docker-git | grep latest
# 1 ligne
docker image ls | grep docker-git | grep '1.0.0'

# Doit fonctionner
docker container run --rm docker-git:1.0.0 git --version
```

Copy

🎓 Mettre à jour votre image (1.1.0)

- Mettez à jour votre image en version `1.1.0` avec les changements suivants :
  - Ajoutez un `LABEL` dont la clef est `description` (et la valeur de votre choix)
  - Configurez `git` pour utiliser une branche `main` par défaut au lieu de `master` (commande `git config --global init.defaultBranch main`)
- Indices :
  - 💡 Commande `docker image inspect <image name>`
  - 💡 Commande `git config --get init.defaultBranch` (dans le conteneur)
  - 💡 Ajoutez des lignes **à la fin** du `Dockerfile`
  - 💡 Documentation de référence des `Dockerfile`

Mettre à jour votre image (1.1.0)

```
cat ./Dockerfile
FROM ubuntu:20.04
RUN apt-get update && apt-get install --yes --no-install-recommends git
LABEL description="Une image contenant git préconfiguré"
RUN git config --global init.defaultBranch main

docker image build -t docker-git:1.1.0 ./docker-git/
# Sending build context to Docker daemon  2.048kB
# Step 1/4 : FROM ubuntu:20.04
#  ---> e40cf56b4be3
# Step 2/4 : RUN apt-get update && apt-get install --yes --no-install-recommends git
#  ---> Using cache
#  ---> 926b8d87f128
# Step 3/4 : LABEL description="Une image contenant git préconfiguré"
#  ---> Running in 0695fc62ecc8
# Removing intermediate container 0695fc62ecc8
#  ---> 68c7d4fb8c88
# Step 4/4 : RUN git config --global init.defaultBranch main
#  ---> Running in 7fb54ecf4070
# Removing intermediate container 7fb54ecf4070
#  ---> 2858ff394edb
Successfully built 2858ff394edb
Successfully tagged docker-git:1.1.0

docker container run --rm docker-git:1.0.0 git config --get init.defaultBranch
docker container run --rm docker-git:1.1.0 git config --get init.defaultBranch
```

Copy

Checkpoint 🎯

- Une image Docker fournit un environnement de système de fichier auto-suffisant (application, dépendances, binaries, etc.) comme modèle de base d'un conteneur
- On peut spécifier une recette de fabrication d'image à l'aide d'un `Dockerfile` et de la commande `docker image build`
- Les images Docker ont une convention de nommage permettant d'identifier les images très précisément

⚠ Friends don't let friends use `latest` ⚠

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,
    // ...
  }
]
```

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;
  // ...
}
```

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

Qu’est 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 manoeuvre 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.

🎓 Exercice : Créez et "taggez" la version v0.0.1 de votre menu-server

```
# Créer un tag.
git tag -a v0.0.1 -m "Release version v0.0.1"


# Publier un tag sur le remote origin.
git push origin v0.0.1
```

Copy

"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

Hello Github Releases!

Une release GiHub est associée à un tag git et porte :

- Un titre
- Un descriptif des changements
- Une collection de d'assets dont:
  - Des tarballs du code source a cette version (automatique)
  - Et éventuellement des fichiers de votre choix (des binaires compilés par exemple)

Prérequis: 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`])

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

Copy

🎓 Exercice: Créer une Release depuis le CI

Changez votre workflow de CI de façon à ce que sur un push de tag, le CI effectue les tâches suivantes dans un nouveau job:

- Une fois que l'étape `build` est terminée
- Télécharge et décompresse l'artefact généré par le job `build`
- Créer une nouvelle release dans votre dépôt ayant pour titre le nom du tag
- Upload `jar` de l'application dans cette release nouvellement créée

On vous suggère d'utiliser la CLI `gh` fournie par GitHub:

- Documentation de `gh release create`

16.9

Déploiement Continu

Continuous Deployment

🤔 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 triviale 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

Introduction à Google Cloud Run

- Dans le cadre de ce cours nous allons utiliser Google Cloud Run
  - Un Produit de Google Cloud Platform (GCP)
- Cloud Run Permets d'exécuter des images de containers (🐳 wink wink) et de les exposer sur internet sans avoir a se soucier de l'infrastructure en dessous.
- Ideal dans le cadre de notre projet!
- Environment sponsorise par Voi pour le cours.

Construire une Image de Container pour Cloud Run

- A la racine de votre dépôt menu-server créez un fichier `Dockerfile` avec le contenu suivant :

Copy

```
# Depuis l'image de base azul/zulu-openjdk:11 (qui embarque un JRE dans la version 11)
FROM azul/zulu-openjdk:17

# Copier l'archive JAR depuis l'hôte dans le fichier /opt/app/menu-server.jar de l'image
COPY target/menu-server.jar /opt/app/menu-server.jar

# Définis la commande par défaut du container à java -jar /opt/app/menu-server.jar  --server.port=${PORT}
# La variable d'environnement PORT est définie par Google Cloud Run à la création du container.
CMD ["java","-jar","/opt/app/menu-server.jar", "--server.port=${PORT}"]
```

🎓 Exercice: créez et exécutez l'image dans votre Workspace

- Dans un terminal, lancez les commandes suivantes:

Copy

```
# On repackage l'app
mvn package

IMAGE_NAME="cicdlectures/menu-server:test"

# Construit une image docker portant le tag `cicdlectures/menu-server:test`
docker image build --tag="${IMAGE_NAME}" ./

# Lance un container basé sur l'image `cicdlectures/menu-server:test` sans specifier la variable d'environnement PORT
docker container run --tty --interactive --rm --publish 8080:9090 "${IMAGE_NAME}"
# Badaboum attendu!

# Lance un container basé sur l'image `cicdlectures/menu-server:test`
docker container run --tty --interactive --rm --env PORT=9090 --publish 8080:9090 "${IMAGE_NAME}"

# Vérifiez que vous pouvez faire des requêtes au menu-server....
# Et Ctrl+C pour terminer l'exécution du container
```

Déployer dans Cloud Run

Les grandes étapes d'un déploiement dans Cloud Run

1. On construit une image de container de l'application et la publie dans une registry d'images Google Cloud.
2. On demande ensuite a Cloud Run d'instancier un nouveau container utilisant la nouvelle image publiée.

🎓 Exercice: Connectez vous a Google Cloud depuis votre Workspace

Cela nécessite un compte Google, si vous n'en avez pas vous pouvez en créer un ici.

Copy

```
# Authentifie votre instance gitpod auprès de google cloud
gcloud auth login

# Paramétrise le projet
GCP_PROJECT_NAME=voi-sdbx-ensg-2023
GCP_REGISTRY=europe-west9-docker.pkg.dev

# Indique a la CLI google cloud d'utiliser le projet partage pour le cours.
gcloud config set project "${GCP_PROJECT_NAME}"

# Authentifie le démon docker de votre instance auprès de la registry Google Cloud.
gcloud auth configure-docker "${GCP_REGISTRY}"
```

🎓 Exercice: Déployez votre Menu Server dans Cloud Run

Copy

```
GCP_IMAGE_NAME="${GCP_REGISTRY}/${GCP_PROJECT_NAME}/cicd-registry/<votre-binôme>/menu-server:test-cloudrun"

# On renomme l'image avec le nom de la registry
docker image tag "${IMAGE_NAME}" "${GCP_IMAGE_NAME}"

# On publie l'image dans la registry google cloud
docker image push "${GCP_IMAGE_NAME}"

# On déploie l'image publiée dans cloud run
gcloud run deploy <votre-dépôt> --image="${GCP_IMAGE_NAME}" --region=europe-west9
```

Mais le faire manuellement c’est pas du déploiement continu! Il faut le faire depuis GitHub action!

🎓 Exercice: Mise en Place du Déploiement Continu dans Cloud Run depuis votre Workflow

Changez votre workflow de CI pour que, sur un évènement de push de tag de version:

- Une fois le build terminé un nouveau job `release-cloud-run` soit lancé
- Ce job effectue dans l'ordre:
  - Télécharge l'artefact de l'étape `build`
  - Checkout le depot (pour rapatrier le Dockerfile)
  - Appelle l'action Cloud Run Release pour deployer automatiquement une nouvelle version de votre application.

Bibliographie

Ligne de commande

- https://blog.balthazar-rouberol.com/category/essential-tools-and-practices-for-the-aspiring-software-developer
- 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

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

Tests

- (FR) http://douche.name/blog/nomenclature-des-tests-logiciels/
- http://martinfowler.com/bliki/UnitTest.html
- https://en.wikipedia.org/wiki/Software_testing
- http://martinfowler.com/tags/testing.html
- http://martinfowler.com/bliki/TestCoverage.html
- http://martinfowler.com/bliki/TestDrivenDevelopment.html

Autre

- https://dduportal.github.io/cours/
- https://www.jenkins.io/node/

Merci !

- damien.duportal <chez> gmail.com
- @DamienDuportal
- jlevesy <chez> gmail.com
- @jlevesy

Slides: https://cicd-lectures.github.io/slides/2023