

Bloc de construction du Pipeline CI/CD

Développement d'une Preuve de Concept

MedHead+

Auteur(s) et contributeur(s)

| Nom & Coordonnées | Qualité & Rôle | Société |
|-------------------|---------------------------------------|------------|
| Gérald ATTARD | Consultant en Architecture logicielle | XXXXXXXXXX |

Historique des modifications et des révisions

| N° version | Date | Description et circonstance de la modification | Auteur |
|------------|------------|--|---------------|
| 1.0 | 18/01/2023 | Création du document | Gérald ATTARD |

Validation

| N° version | Nom & Qualité | Date & Signature | Commentaires & Réserves |
|------------|--|------------------|-------------------------|
| 1.0 | Kara Trace CIO, Ursa Major Health | | |
| | Anika Hansen, PDG, Jupiter Scheduling Inc. | | |
| | Chris Pike Architecte d'entreprise principal, Schedule Shed | | |

Tableau des abréviations

| Abr. | Sémantique |
|------|--|
| CLUF | Contrant de Licence Utilisateur Final |
| MTBF | Mean-Time-Between-Failure (temps moyen qui sépare deux erreurs en production) |
| MTTR | Mean-Time-To-Recover (temps moyen de correction entre deux erreurs de production) |
| MVP | Minimum Viable Product (trad. <i>produit minimal viable</i>) |
| Repo | Repository (trad. <i>dépôt de fichiers</i>) |
| SAW | Statement for Architecture Work (trad. <i>déclaration pour l'architecture</i>) |
| SBB | Solution Building Block (trad. <i>bloc de construction de la solution</i>) |

Table des matières

| | |
|--|----|
| I. Introduction..... | 5 |
| II. Fonctionnalités spécifiques et attributs..... | 5 |
| II.A. Intégration continue..... | 5 |
| II.A.1. Plannification du développement des fonctionnalités..... | 6 |
| II.A.2. Compiltation et intégration du code..... | 6 |
| II.A.2.a. contrôleur de code source..... | 6 |
| II.A.2.b. orchestrateur..... | 7 |
| II.A.3. Tests (unitaires)..... | 8 |
| II.A.4. Mesure de la Qualité du code..... | 9 |
| II.A.5. Stockage du code..... | 10 |
| II.B. Déploiement continu..... | 10 |
| II.B.1. Codification de l'infrastructure et déploiement..... | 10 |
| II.B.2. Test (de l'application)..... | 11 |
| II.B.3. Supervision et alerte..... | 11 |
| II.C. Outils..... | 12 |
| II.C.1. Contexte d'installation..... | 13 |
| II.C.2. Java..... | 14 |
| II.C.2.a. Installation..... | 14 |
| II.C.2.b. Configuration..... | 14 |
| II.C.3. Maven..... | 15 |
| II.C.3.a. Installation..... | 15 |
| II.C.3.b. Configuration..... | 15 |
| II.C.4. Jenkins..... | 16 |
| II.C.4.a. Installation..... | 16 |
| II.C.4.b. Configuration..... | 17 |
| II.C.5. Docker..... | 18 |
| II.C.5.a. Installation..... | 18 |
| II.C.5.a.i. Préambule : configuration du dépôt..... | 18 |
| II.C.5.a.ii. installation de l'applicatif..... | 18 |
| II.C.5.b. Configuration..... | 19 |
| II.C.6. Configuration de Jenkins pour intégrer Docker..... | 20 |
| III. Etape de réalisation du pipeline..... | 21 |
| IV. Finalisation du pipeline..... | 22 |
| IV.A. Technique..... | 23 |

I. Introduction

Pour introduire cette SBB, il sera nécessaire d'être conscient que le processus de réalisation de la Poc se compose de trois domaines consécutifs :

1. l'intégration continue ;
2. la livraison continue ;
3. le déploiement continu.

Ainsi, bien que ces trois domaines soient différents, les deux premiers points seront intrinséquement liés et automatisés par le pipeline CI/CD lui-même et ces deux domaines seront présentés en un seul point dans ce document.

II. Fonctionnalités spécifiques et attributs

Ce paragraphe a pour objectif de présenter les processus d'intégration continue et de livraison continue permettant de :

- accélérer le Time-To-Market ;
- réduire les erreurs lors des livraisons ;
- assurer une continuité de service des applications.

II.A. Intégration continue

L'intégration continue est un ensemble de pratiques utilisées en génie logiciel, consistant à vérifier, à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée, afin d'assurer que :

- les intégrations de code soit le plus rapide possible ;
- il n'y ait aucun bug introduit par le nouveau code.

Le principe de l'intégration continue est donc de détecter les problèmes d'intégration au plus tôt dans le cycle de développement, en suivant cinq étapes successives :

- la planification du développement ;
- la compilation du nouveau code ;
- le test du nouveau code ;
- la mesure de la qualité du nouveau code ;
- la gestion et le stockage des livrables de l'application.

II.A.1. Plannification du développement des fonctionnalités

En méthode AGILE, cette étape est réalisée à partir des backlogs afin de savoir quoi développer. Il est alors nécessaire de mettre à disposition des développeurs un outil permettant notamment de :

- gérer les différentes versions de l'application,
- assurer l'intégration des fonctionnalités développées,
- garantir la priorité/priorisation des backlogs.

Ainsi, intervenant tout au long du projet, la collaboration de toute l'équipe projet est nécessaire pour assurer la planification de celui-ci. Cette planification est étroitement liée à des méthodes de gestion de projet, dont l'Agile et la méthodologie Scrum. Cette dernière a pour but de découper le projet en petites tâches à réaliser par toute l'équipe.

II.A.2. Compilation et intégration du code

Dans cette étape, il sera nécessaire de disposer de deux éléments :

- un contrôleur de code source,
- un orchestrateur.

II.A.2.a. contrôleur de code source

Le code source produit devra être disponible à chaque instant sur un dépôt central. Chaque développement fera ainsi l'objet d'un suivi de révision.

Le code devra être compilable à partir d'une récupération fraîche, et ne faire l'objet d'aucune dépendance externe.

Même s'il existe des notions de branche, la création d'une branche doit être évitée le plus possible ; excepté en local pour les propres besoins du développeur.

Ainsi, cette pratique privilégiera le développement sur la branche principale ; cela évite de maintenir plusieurs versions en parallèle. Ce genre de pratique est appelé *trunk-based development*.

II.A.2.b. orchestrateur

Toutes les étapes du processus d'intégration seront automatisées par un *orchestrateur* sachant reproduire ces étapes et gérer les dépendances entre elles.

De plus, l'utilisation d'un orchestrateur permet de donner accès, à tout moment et à tous les membres de l'équipe projet, à un tableau de bord présentant l'état de santé des étapes d'intégration continue. Ainsi, les développeurs ont, au plus tôt, la boucle de feedback nécessaire afin de garantir que l'application soit prête, à tout moment.

En outre, la première étape est de compiler le code de manière continue. En effet, sans cette étape, le code est compilé manuellement sur le poste de chaque développeur, afin que ces derniers s'assurent que leur code compile.

Malheureusement, comme énoncé précédemment, chaque développeur ne peut pas s'assurer que son code permet de bien compiler avec tous les autres développements faits par l'équipe ? Cet état de fait implique que pour chaque livraison, un développeur intègre manuellement toutes les modifications ; opération chronophage générant peine et souffrance...

La mise en place d'une première étape de compilation dans un processus d'intégration continue permet justement de ne plus se soucier si des modifications de code cassent la compilation.

Le développeur doit simplement s'assurer de bien envoyer son code source sur le dépôt central. En réalisant cela, il déclenche une première étape de compilation, avec toutes les modifications des autres développeurs.

Si la compilation ne se fait pas, le code est alors **rejeté**, et le développeur doit **corriger** ses erreurs.

Après cette première étape, le code demeure sûr, et le dépôt de code source garantit qu'à chaque instant, un développeur récupère un code qui compile.

II.A.3. Tests (unitaires)

Dans cette étape, l'orchestrateur va se charger de lancer les tests unitaires tout de suite après la compilation. Ces tests unitaires, généralement avec un framework associé, garantiront que le code respecte bien un certain niveau de qualité.

En effet, les tests unitaires permettent de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme. Plus il y a de tests unitaires, plus le code est garanti sûr. Évidemment, l'orchestrateur ne peut lancer que les tests qui ont été codés par les développeurs ; il ne peut, en aucun cas, générer de nouveaux cas de tests.

Ces tests devront s'exécuter le plus rapidement possible afin d'avoir un feedback immédiat ou presque. Pour arriver à ce niveau, il est nécessaire que les tests unitaires n'aient aucune dépendance vis-à-vis de systèmes externes, comme par exemple une base de données, ou même un système de fichiers quelconque.

En outre, les tests unitaires apportent 3 atouts à la production :

- **trouver les erreurs plus facilement** : les tests sont exécutés durant tout le développement, permettant de visualiser si le code fraîchement écrit correspond au besoin.
- **Sécuriser la maintenance** : lors d'une modification d'un programme, les tests unitaires signalent les éventuelles régressions. En effet, certains tests peuvent échouer à la suite d'une modification, il faut donc soit réécrire le test pour le faire correspondre aux nouvelles attentes, soit corriger l'erreur se situant dans le code.
- **Documenter le code** : les tests unitaires peuvent servir de complément à la documentation; il est utile de lire les tests pour comprendre comment s'utilise une méthode. De plus, il est également possible que la documentation ne soit plus à jour, mais les tests, eux, devront TOUJOURS correspondre à la réalité de l'application. Pour les puristes, l'ajout d'éléments de documentation au sein du code est prohibé car le code doit être suffisamment claire et explicite pour qu'il n'y ait pas besoin de documentation supplémentaire pour en expliquer le fonctionnement.

L'ensemble des tests unitaires sera relancé après chaque modification du code, afin de vérifier qu'il n'y ait pas de régression ou d'apparition de dysfonctionnements. Ainsi, la multiplicité des tests unitaires oblige à les maintenir dans le temps, au fur et à mesure que le développement avance.

II.A.4. Mesure de la Qualité du code

Maintenant que les tests unitaires sont écrits et exécutés, il est nécessaire d'avoir une meilleure qualité de code, et d'assurer la fiabilité et la robustesse de l'application elle-même.

Grâce à la compilation et aux tests unitaires, il est alors possible de mesurer la qualité du code ; ceci permet alors aux développeurs de maintenir dans le temps un code de qualité et d'alerter l'équipe en cas de dérive des bonnes pratiques de tests.

De plus, l'étape de qualité de code est différente de l'étape de test. Cette étape de qualité s'assure que le code demeure maintenable et évolutif au fur et à mesure de son cycle de vie, alors que les tests unitaires servent à garantir que le code implémente bien les fonctionnalités demandées, et ne contient pas (~~ou peu~~) de bugs.

Lors de l'étape de qualité de code, les développeurs vont s'efforcer de générer le moins de dette technique possible au sein de l'application. Cette dette technique est le temps nécessaire à la correction de bugs ou à l'ajout de nouvelles fonctionnalités, lorsque, par exemple, les règles de codage ne sont pas appliquées. Cette dette s'exprimera alors en heures de correction ; plus cette dette est élevée et plus le code sera difficile à maintenir et à faire évoluer.

En outre, l'étape de qualité de code fournit aussi d'autres métriques relatifs à :

- le nombre de vulnérabilités au sein du code ;
- la couverture de test ;
- les *code smells* (utilisation de mauvaises pratiques) ;
- la complexité cyclomatique (complexité du code applicatif) ;
- la duplication de code.

Dans cette étape, chaque développeur s'imposera de :

- respecter les normes et les règles définies ;
- corriger au fur et à mesure son code.

Enfin, pour renforcer la qualité du code et ne pas autoriser le déploiement d'un code de mauvaise qualité, l'équipe pourra forcer un arrêt complet du pipeline d'intégration continue, si le code n'atteint pas la qualité requise ; cette opération de dernier recours devra être perçue, par les parties prenantes, comme une preuve de l'engagement Qualité de la part de l'équipe de développeur.

II.A.5. Stockage du code

Gestion des livrables dans un entrepôt/dépôt

Une fois le code compilé, celui-ci doit être déployé dans un dépôt de livrables et versionné.

Les *binaires* produits sont communément appelés *artefacts*. Ceux-ci devront être accessibles à toutes les parties prenantes de l'application, afin de pouvoir les déployer et lancer les tests autres qu'unitaires (test de performance, test de bout en bout, etc.).

Ces artefacts seront disponibles dans un stockage, centralisé et organisé, de données. Cet emplacement pourra être une ou plusieurs bases de données où les artefacts sont localisés en vue de leur distribution sur le réseau, ou bien un endroit directement accessible aux utilisateurs.

II.B. Déploiement continu

II.B.1. Codification de l'infrastructure et déploiement

Comme son nom l'indique, cette pratique consiste à décrire une infrastructure avec du code qui est stocké avec le code de l'application et fait partie intégrante de cette dernière.

L'utilisation d'une telle codification présentera certains avantages, tels que :

- la possibilité de créer des environnements à la demande ;
- la création rapide d'environnement en quelques minutes, contre plusieurs semaines dans une entreprise classique ;
- le pilotage de l'infrastructure grâce au pipeline de livraison continue ;
- la connaissance des logiciels installés sur la plateforme, grâce à l'outillage ;
- la montée de version des environnements automatisés.

Une fois l'infrastructure codifiée, il sera alors temps de piloter les packages créés lors des étapes précédentes d'intégration continue. L'utilisation de l'infrastructure permettra alors d'automatiser ce déploiement pour permettre :

- à l'équipe, de se concentrer sur le développement de sa valeur à ajouter ;
- à n'importe quel membre de l'équipe, de déployer des logiciels.

Ainsi, l'utilisation du déploiement continu permettra alors de :

- minimiser les risques d'erreur ;
- reproduire aisément certains contextes environnementaux d'exécution ;
- déployer facilement sur un nouvel environnement ;
- réaliser fréquemment des déploiements.

II.B.2. Test (de l'application)

Bien que cette phase soit primordiale, celle-ci a été traitée au sein du document de plan de tests. Le lecteur est donc invité à se référer à ce document.

II.B.3. Supervision et alerte

Le monitoring, ou supervision, interviendra une fois l'application déployée sur un environnement, que ce soit un environnement de staging, de test, de démonstration ou de production.

Le principe est de récupérer certaines métriques qui ont du sens pour ceux qui interviennent sur l'application. Cela peut être par exemple le nombre de connexions HTTP, le nombre de requêtes à la base de données, le temps de réponse de certaines pages ou d'autres métriques plus orientées métier.

Les métriques peuvent être aussi sur la partie livraison en elle-même, ou sur le processus de développement. Par exemple, l'équipe peut mesurer le nombre de déploiements qu'elle effectue par jour, ou encore deux autres indicateurs qui sont importants afin de voir la performance de l'équipe sur la correction d'erreurs qui peuvent survenir en production, tels que :

- le **MTBF** (*Mean-Time-Between-Failure*) est le temps moyen qui sépare deux erreurs en production. Plus ce temps est élevé, plus le système est stable et fiable, notamment du fait de la qualité des tests qui sont joués lors de la livraison continue ;
- le **MTTR** (*Mean-Time-To-Recover*) est le temps moyen de correction entre deux erreurs de production. Plus ce temps est faible, plus l'équipe est apte à détecter des erreurs et à les corriger rapidement.

En ce qui concerne plus spécifiquement les alertes, celles-ci interviendront surtout au début du cycle de production, notamment pour l'utilisation métier de l'application. Elles se concrétiseront alors lors de la réalisation du **MVP** (*Minimum Viable Product*) et seront généralement d'ordre informatif.

II.C.Outils


Afin de réaliser les différentes étapes citées précédemment un ensemble d'outils logiciels est préconisé, à savoir, pour :

- l'intégration continue :
 - étape Planification : **Scrum/GitLab**.
 - étape Compilation et Intégration :
 - contrôle de source : **GitHub** ;
 - orchestrateur : **Jenkins** ;
 - compilation : **Maven**.
 - étape de Test : **Junit**.
 - étape Mesure de Qualité : *TBD*.
 - étape Gestion des livrables : **GitHub**.
- le déploiement continu : *Spinnaker, XLDpeloy, UrbanCode* ;
- les tests d'acceptance : *Confluence, FitNesse, Ranorex* ;
- les tests de performance : *Jmeter, Apache Bench, Gatling* ;
- les smoke tests : *Selenium, SoapUI, Cypress* ;
- la supervision : *Elastic, Prometheus, Graylog* ;
- MTBF / MTTR : *Dynatrace, Sysdig, New Relic* ;
- l'envoi de requêtes HTTP : **framework SpringBoot/Spring Web starter**;
- le serveur d'application/conteneurisation : **Docker** ;
- le retour d'expérience/ communication de l'équipe de développement et autres alertes : *Slack, Trello, Twitter, Teams*.

Nota : les outils mentionnés en **gras** dans la précédente liste seront ceux utilisés pour développer la PoC, les outils mentionnés en *italique* représentent des options pouvant être choisies pour le reste du développement du projet.

II.C.1. Contexte d'installation

En outre, l'environnement d'installation du pipeline CI/CD est indiqué ci-dessous :



Ubuntu

| | |
|--------------------------------|---|
| Nom de l'appareil | i8700K > |
| Modèle du matériel | Micro-Star International Co., Ltd. MS-7B45 |
| Mémoire | 32,0 Gio |
| Processeur | Intel® Core™ i7-8700K CPU @ 3.70GHz × 12 |
| Carte graphique | NVIDIA Corporation TU104 [GeForce RTX 2080 SUPER] |
| Capacité du disque | 3,6 To |
| Nom du système d'exploitation | Ubuntu 22.04.1 LTS |
| Type de système d'exploitation | 64 bits |
| Version de GNOME | 42.5 |
| Système de fenêtrage | X11 |
| Mises à jour logicielles | > |

II.C.2. Java

II.C.2.a. Installation

L'installation de ce framework a simplement nécessité la décompression d'une image téléchargée sur le site d'Oracle dans un répertoire local */opt*.

```
$ cd /opt
$ sudo wget https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.tar.gz
$ tar xvf jdk-17_linux-x64_bin.tar.gz
$ ln -s ./jdk-17 jdk
```

- Vérification de l'installation

```
$ java -version
openjdk 17 2021-09-14
OpenJDK Runtime Environment (build 17+35-2724)
OpenJDK 64-Bit Server VM (build 17+35-2724, mixed mode, sharing)
```

II.C.2.b. Configuration

La configuration consiste ici simplement à déclarer la variable d'environnement `$JAVA_HOME` en éditant le fichier `~/.bashrc` :

```
$ echo 'PIPELINE=/opt' >> " ~/.bashrc
$ echo 'export JAVA_HOME=$PIPELINE/java/jdk' >> " ~/.bashrc
$ echo 'export PATH=$JAVA_HOME/bin:${PATH}' >> " ~/.bashrc
$ source ~/.bashrc
```

II.C.3. Maven

II.C.3.a. Installation

L'installation de ce framework a simplement nécessité la décompression d'une image téléchargée sur le site d'Oracle dans un répertoire local `/opt`.

```
$ cd /opt
$ sudo wget
https://dlcdn.apache.org/maven/maven-3/3.9.0/binaries/apache-
maven-3.9.0-bin.tar.gz
$ tar xvf apache-maven-3.9.0-bin.tar.gz
$ ln -s ./apache-maven3.9.0 maven
```

- Vérification de l'installation

```
$ mvn -v
Apache Maven 3.9.0 (9b58d2bad23a66be161c4664ef21ce219c2c8584)
Maven home: /opt/maven
Java version: 17, vendor: Oracle Corporation, runtime:
/opt/java/jdk-17
Default locale: en_GB, platform encoding: UTF-8
OS name: "linux", version: "5.19.0-35-generic", arch: "amd64", \
family: "unix"
```

II.C.3.b. Configuration

La configuration consiste ici simplement à déclarer la variable d'environnement `$MAVEN_HOME` en éditant le fichier `~/.bashrc` :

```
$ echo 'export MAVEN_HOME=$PIPELINE/java/jdk' >> " ~/.bashrc
$ echo 'export PATH=$MAVEN_HOME/bin:${PATH}' >> " ~/.bashrc
$ source ~/.bashrc
```

II.C.4. Jenkins

II.C.4.a. Installation

- Ajout de la clé GPG officielle de Jenkins :

```
$ curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io.key |  
sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null
```

- Mise à jour du dépôt:

```
$ echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]  
https://pkg.jenkins.io/debian-stable binary/ | sudo tee  
/etc/apt/sources.list.d/jenkins.list > /dev/null
```

```
$ sudo apt-get update
```

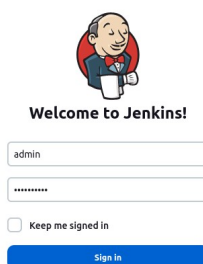
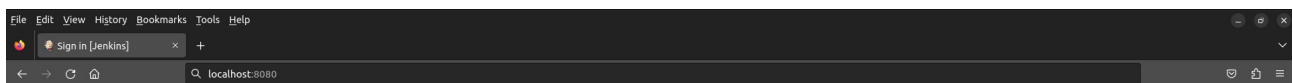
- Installation de la dernière version

```
$ sudo apt-get install jenkins
```

- Vérification de l'installation

Dans la barre d'adresse d'un navigateur Internet, saisir :

<http://localhost:8080>



...si l'installation a fonctionné nominalement, la page d'authentification de Jenkins s'affiche.

II.C.4.b. Configuration

Pour la configuration globale de Jenkins, il va falloir renseigner à Jenkins le JDK et le Maven à utiliser.

- A partir de la page d'accueil de Jenkins, aller dans *Configuration de Jenkins* puis dans *Configuration globale des outils*.
- Dans la section JDK, saisir un nom du JDK (ici *JDK*) puis le chemin ABSOLU du répertoire d'installation de celui-ci (ici *opt/java/jdk*).

JDK

JDK installations
List of JDK installations on this system

JDK Name

JAVA_HOME

☐ Install automatically ?

- Descendre dans la section Maven, saisir un nom (ici *maven*) puis le chemin ABSOLU du répertoire d'installation de Maven (ici */opt/maven*) :

Maven

Maven installations
List of Maven installations on this system

Maven Name

MAVEN_HOME

☐ Install automatically ?

...puis cliquer, sur **Appliquer** et **Sauver**.



II.C.5. Docker

II.C.5.a.Installation

II.C.5.a.i.Préambule : configuration du dépôt

- Désinstallation d'éventuelles anciennes versions :

```
$ sudo apt-get remove docker docker-engine docker.io containerd  
runc
```

- Mise à jour des index de paquets pour permettre à l'applcatif apt d'utiliser le dépôt via HTTPS :

```
$ sudo apt update
```

```
$ sudo apt-get install ca-certificates curl gnupg lsb-release
```

- Ajout de la clé GPG officielle de Docker :

```
$ sudo mkdir -m 0755 -p /etc/apt/keyrings
```

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg \  
| sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

- Mise à jour du dépôt:

```
$ echo "deb [arch=$(dpkg --print-architecture) \  
signed-by=/etc/apt/keyrings/docker.gpg] \  
https://download.docker.com/linux/ubuntu $(lsb_release -cs) \  
stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

II.C.5.a.ii. installation de l'applcatif

- Mise à jour du dépôt:

```
$ sudo apt-get update
```

- Installation de la dernière version

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io  
docker-buildx-plugin docker-compose-plugin
```

- Vérification de l'installation

```
$ sudo docker run hello-world
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

II.C.5.b. Configuration

- Ajout des groupes d'utilisateur

```
$ sudo groupadd docker
```

```
$ sudo usermod -aG docker $USER
```

```
$ newgrp docker
```

- Vérification qu'il est possible de lancer Docker sans commande *sudo* :

```
$ docker run hello-world
```

- Autorisation de Docker pour appeler Jenkins

```
$ sudo usermod -aG docker jenkins
```

```
$ sudo usermod -aG root jenkins
```

```
$ sudo chmod 666 /var/run/docker.sock
```

```
$ newgrp docker
```

- Configuration pour lancer Docker au démarrage

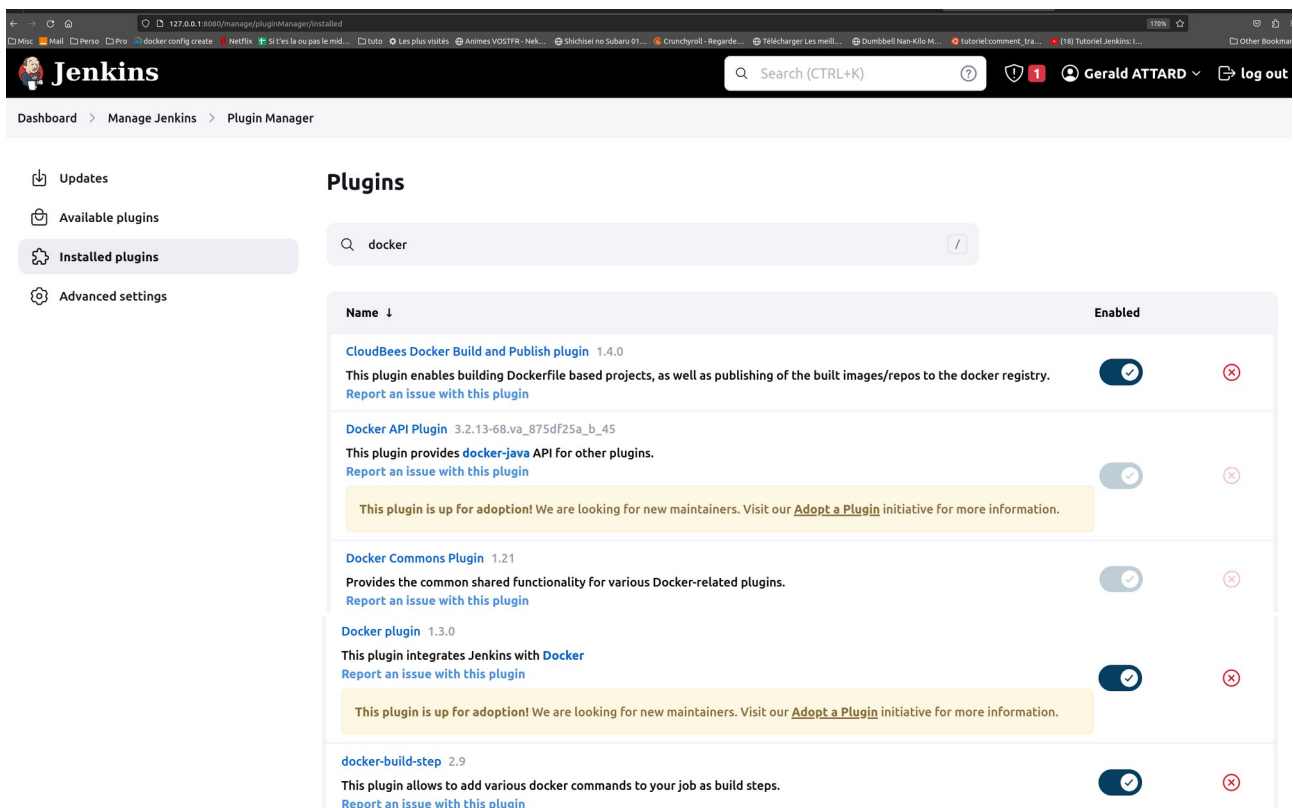
```
$ sudo systemctl enable docker.service
```

```
$ sudo systemctl enable containerd.service
```

II.C.6. Configuration de Jenkins pour intégrer Docker

La configuration nécessaire de Jenkins pour intégrer Docker se réalise en ajoutant les plugins suivants dans le menu de configuration (Manage Plugins) de Jenkins :

- *CloudBees Docker Build Publish*
- *Docker API plugin*
- *Docker commons plugin*
- *Docker plugin*
- *Docker Build Step plugin*



The screenshot shows the Jenkins web interface at the 'Manage Jenkins' > 'Plugin Manager' page. The left sidebar contains navigation links: Updates, Available plugins, Installed plugins (selected), and Advanced settings. The main area is titled 'Plugins' and features a search bar with 'docker' entered. Below the search bar, a table lists installed plugins with columns for Name and Enabled status.

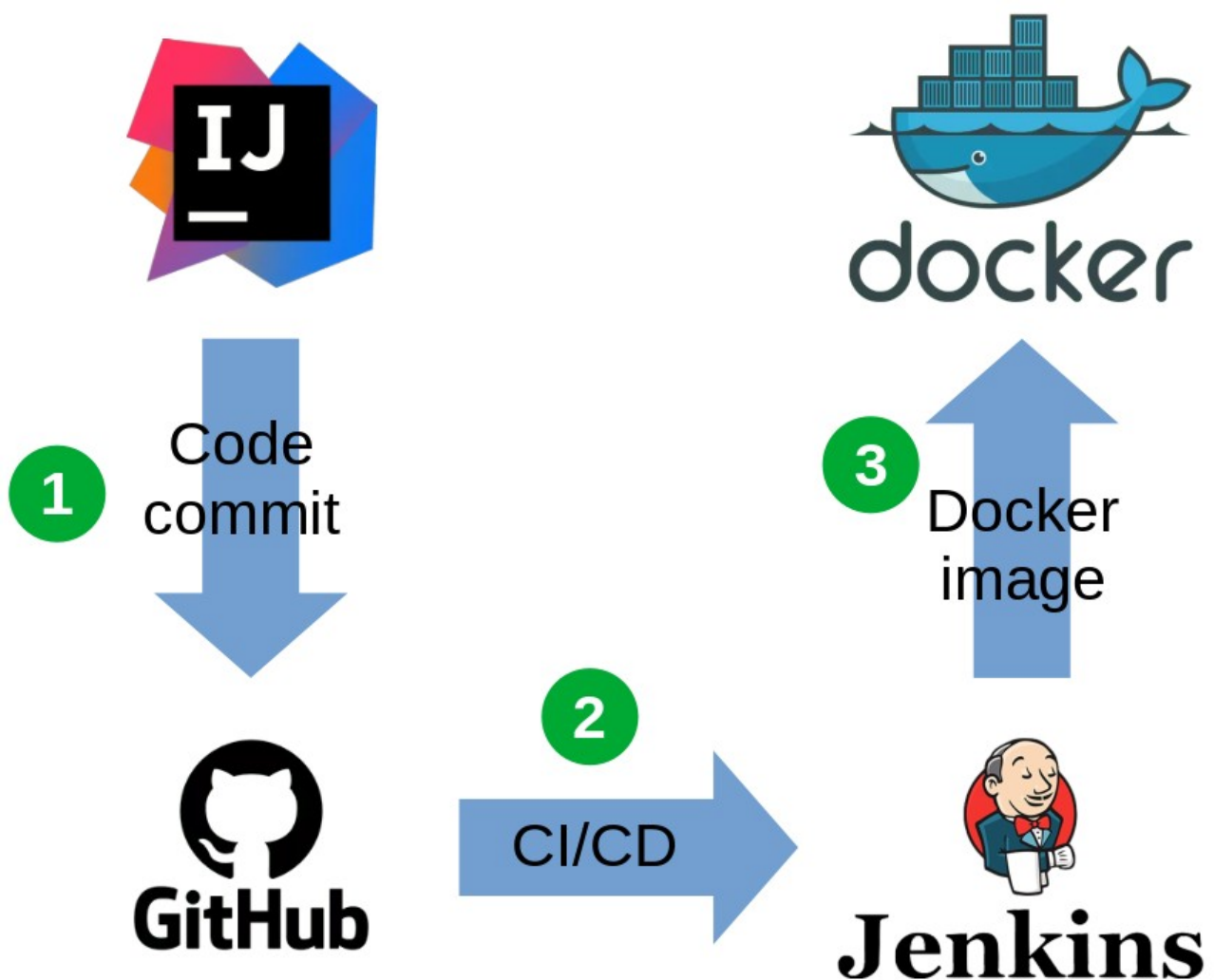
| Name ↓ | Enabled |
|---|-------------------------------------|
| CloudBees Docker Build and Publish plugin 1.4.0 This plugin enables building Dockerfile based projects, as well as publishing of the built images/repos to the docker registry. Report an issue with this plugin | <input checked="" type="checkbox"/> |
| Docker API Plugin 3.2.13-68.va_875df25a_b_45 This plugin provides docker-java API for other plugins. Report an issue with this plugin <div>This plugin is up for adoption! We are looking for new maintainers. Visit our Adopt a Plugin initiative for more information.</div> | <input checked="" type="checkbox"/> |
| Docker Commons Plugin 1.2.1 Provides the common shared functionality for various Docker-related plugins. Report an issue with this plugin | <input checked="" type="checkbox"/> |
| Docker plugin 1.3.0 This plugin integrates Jenkins with Docker Report an issue with this plugin <div>This plugin is up for adoption! We are looking for new maintainers. Visit our Adopt a Plugin initiative for more information.</div> | <input checked="" type="checkbox"/> |
| docker-build-step 2.9 This plugin allows to add various docker commands to your job as build steps. Report an issue with this plugin | <input checked="" type="checkbox"/> |

III. Etape de réalisation du pipeline

La réalisation du pipeline CI/CD se base sur quatre outils principaux :

- **IntelliJ IDEA** : l'IDE d'écriture du code ;
- **GitHub** : l'élément chargé du versionning et du stockage du code ;
- **Jenkins** : l'orchestrateur du pipeline CI/CD ;
- **Docker** : le conteneur d'image pour les isoler lors de leur exécution.

Le diagramme ci-dessous présente la séquence d'intervention de ces outils :

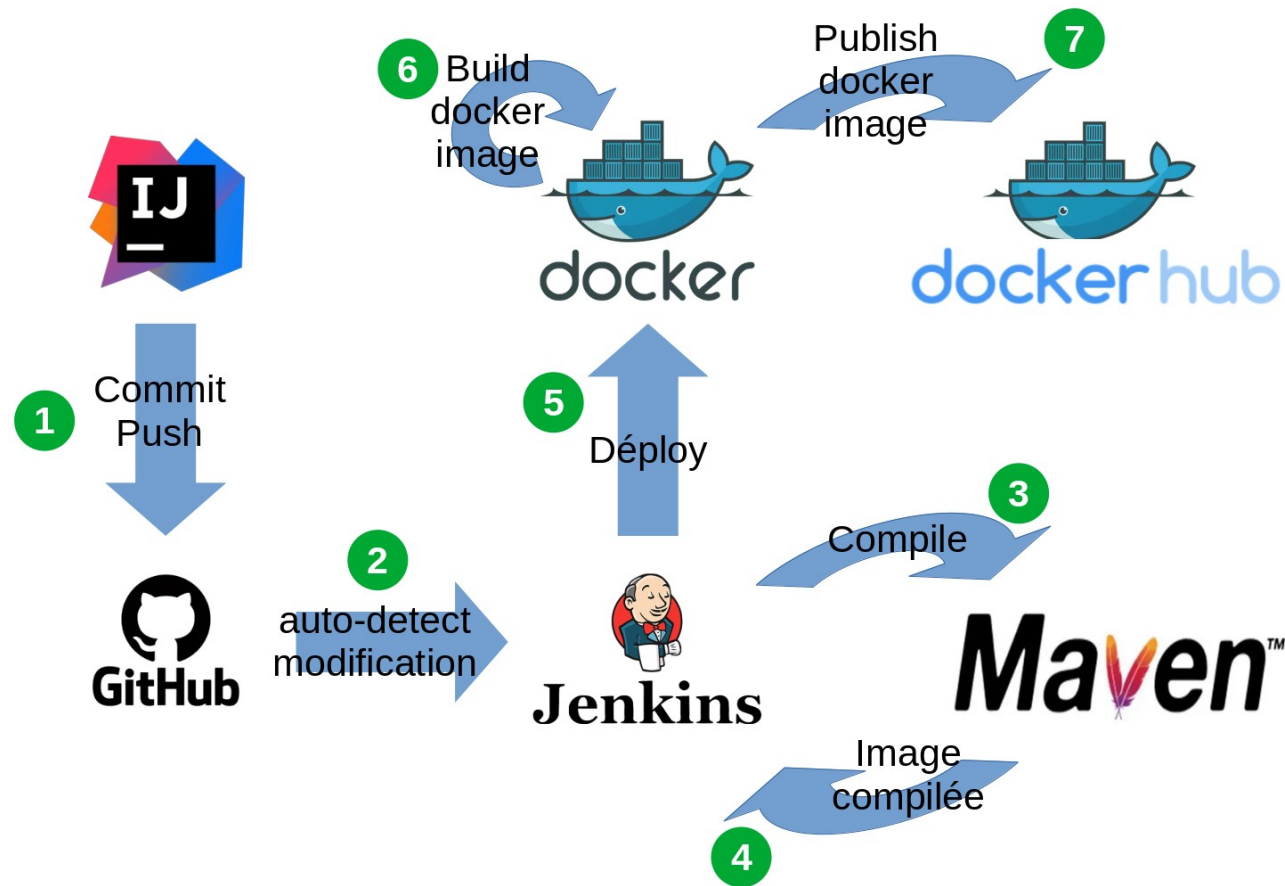


IV. Finalisation du pipeline

Une fois la structure de base du pipeline CI/CD créée, il faudra la compléter à l'aide de deux autres outils :

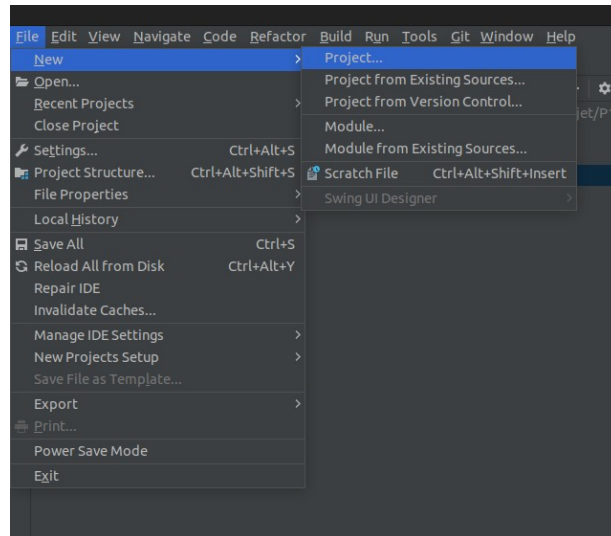
- **Maven** : le moteur d'automatisation de compilation/build d'application ;
- **DockerHub** : l'outil de stockage et de publication des images Docker créées.

Finalement, les différentes phases du pipeline se présenteront comme suit :

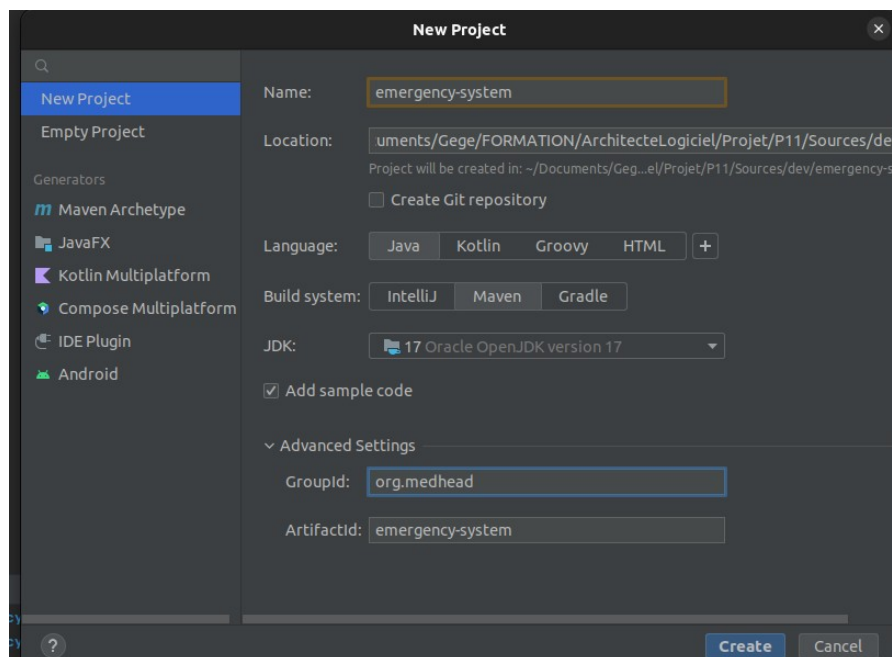


IV.A. Technique

- A partir d'INTELLIJ IDEA, créer un nouveau projet :

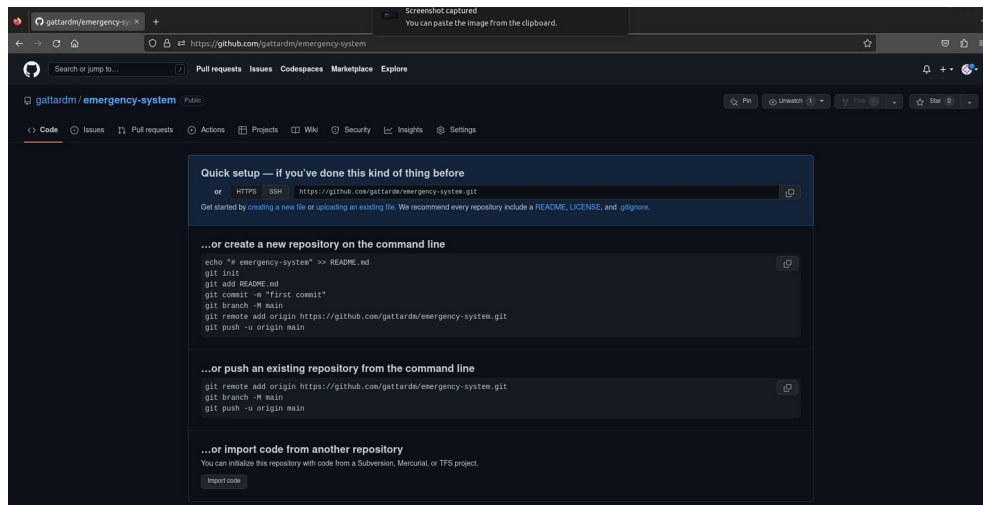


- Renseigner le nom du projet, sa localisation, le langage Java, le système de construction Maven, le JDK à utiliser (ici OpenJDK 17), le groupId (ici org.medhead), tels que montré ci-dessous :



...puis cliquer sur **Créer**.

- Lorsque le projet est créé, se rendre sur son compte GitHub et créer un dépôt avec le même nom que le projet créé ci-dessus (emergency-system) :



De retour, sur IntelliJ IDEA, ouvrir un terminal, et il va falloir initialiser le dépôt GitHub au même chemin de création que le projet créé précédemment. Pour cela, entrer les lignes suivantes dans le terminal d'IntelliJ IDEA :

```
$ cd <chemin_du_projet>
$ git init
$ git add .
$ git commit -m first "commit"
$ git remote add origin \ https://github.com/<username>/<repository>.git
$ git push -u origin master
```

```
Terminal: Local + v
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialised empty Git repository in /home/solarhis/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system/.git/
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  .idea/
  Dockerfile
  pom.xml
  src/

nothing added to commit but untracked files present (use "git add" to track)
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git add .
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git commit -m "first commit"
[master (root-commit) 6a0c0d3] first commit
 9 files changed, 175 insertions(+)
 create mode 100644 .idea/compiler.xml
 create mode 100644 .idea/encodings.xml
 create mode 100644 .idea/jarRepositories.xml
 create mode 100644 .idea/misc.xml
 create mode 100644 .idea/vcs.xml
 create mode 100644 .idea/workspace.xml
 create mode 100644 Dockerfile
 create mode 100644 pom.xml
 create mode 100644 src/main/java/org/eehead/Main.java
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git status
On branch master

nothing to commit, working tree clean
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git remote add origin https://github.com/gattardm/emergency-system.git
solarhis@solarhisg0d-HS-7845:~/Documents/Gege/FORMATION/ArchitecteLogiciel/Projet/P11/Sources/dev/emergency-system$ git push -u origin master
```


- Maintenant se rendre sur l'outil jenkins à partir d'un navigateur (<http://localhost:8080>), créer une nouvelle tâche appelée **emergency-system** en *Projet style libre*, puis faire **OK** :

Enter an item name

emergency-system

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Maven project
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

OK

- Dans la fenêtre Configuration qui s'affiche ensuite, il va falloir renseigner le projet GitHub précédemment créé(ici <https://github.com/gattardm/emergency-system>), la source de gestion du code (ici <https://github.com/gattardm/emergency-system.git>), l'ordonnanceur de gestion de version qui va aller chercher le code périodiquement (ici * * * * *) et ajouter les cibles de haut-niveau Maven (ici *install*)

☒ GitHub project

Project url ?

https://github.com/gattardm/emergency-system

Advanced...

Source Code Management

☐ None

☒ Git ?

Repositories ?

Repository URL ?

https://github.com/gattardm/emergency-system.git

☒ Poll SCM ?

Schedule ?

⚠ Do you really mean "every minute" when you say "*****"? Perhaps you meant "H*****" to poll once per hour

Would last have run at Thursday, March 9, 2023 at 10:47:03 AM Central European Standard Time; would next run at Thursday, March 9, 2023 at 10:47:03 AM Central European Standard Time.

Invoke top-level Maven targets ?

Maven Version

(Default)

Goals

install

Advanced...

- Pour vérifier que la tâche à bien été créée, il suffit de lancer un build et de regarder les logs de la console Jenkins :

Build Now

#1

9 Mar 2023, 10:50

Atom feed for all

Atom feed for failures

Jenkins

Dashboard > emergency-system > #1

Status

Changes

Console Output

View as plain text

Edit Build Information

Delete build '#1'

Git Build Data

Console Output

Started by user Gerald ATTARD

Running as SYSTEM

Building in workspace /var/lib/jenkins/workspace/emergency-system

The recommended git tool is: NONE

No credentials specified

> git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/emergency-system/.git # timeout=10

Fetching changes from the remote Git repository

> git config remote.origin.url https://github.com/gattard/emergency-system.git # timeout=10

Fetching upstream changes from https://github.com/gattard/emergency-system.git

> git --version # timeout=10

> git --version # 'git version 2.34.1'

> git fetch --tags --force --progress -- https://github.com/gattard/emergency-system.git +refs/heads/*:refs/remotes/origin/* # timeout=10

> git rev-parse refs/remotes/origin/master^{commit} # timeout=10

Checking out Revision 6a0c0d30b612cfeef961471a92d9dcadc3deb7e3 (refs/remotes/origin/master)

> git config core.sparsecheckout # timeout=10

> git checkout -f 6a0c0d30b612cfeef961471a92d9dcadc3deb7e3 # timeout=10

Commit message: "first commit"

First time build. Skipping changelog.

[emergency-system] \$ /opt/maven/bin/mvn install

[INFO] Scanning for projects...

[INFO] ----- org.medhead:emergency-system > -----

[INFO] Building emergency-system 1.0-SNAPSHOT

[INFO] from pom.xml

[INFO] ----- [jar] -----

[INFO] --- resources:3.3.0:resources (default-resources) @ emergency-system ---

[INFO] skip non existing resourceDirectory /var/lib/jenkins/workspace/emergency-system/src/main/resources

[INFO] --- compiler:3.10.1:compile (default-compile) @ emergency-system ---

[INFO] Changes detected - recompiling the module!

[INFO] Compiling 1 source file to /var/lib/jenkins/workspace/emergency-system/target/classes

[INFO] --- resources:3.3.0:testResources (default-testResources) @ emergency-system ---

[INFO] skip non existing resourceDirectory /var/lib/jenkins/workspace/emergency-system/src/test/resources

[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ emergency-system ---

[INFO] No sources to compile

[INFO] --- surefire:3.0.0-M8:test (default-test) @ emergency-system ---

[INFO] No tests to run.

[INFO] --- jar:3.3.0:jar (default-jar) @ emergency-system ---

[INFO] Building jar: /var/lib/jenkins/workspace/emergency-system/target/emergency-system.jar

[INFO] --- install:3.1.0:install (default-install) @ emergency-system ---

[INFO] Installing /var/lib/jenkins/workspace/emergency-system/pom.xml to /var/lib/jenkins/.m2/repository/org/medhead/emergency-system/1.0-SNAPSHOT/emergency-system-1.0-SNAPSHOT.pom

[INFO] Installing /var/lib/jenkins/workspace/emergency-system/target/emergency-system.jar to /var/lib/jenkins/.m2/repository/org/medhead/emergency-system/1.0-SNAPSHOT/emergency-system-1.0-SNAPSHOT.jar

[INFO] BUILD SUCCESS

[INFO] -----

[INFO] Total time: 0.970 s

[INFO] Finished at: 2023-03-09T10:50:44+01:00

[INFO] -----

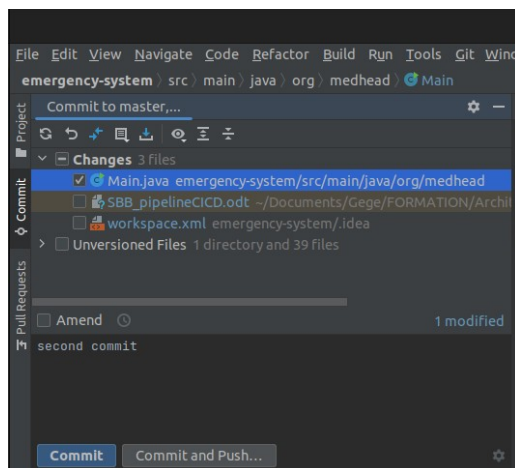
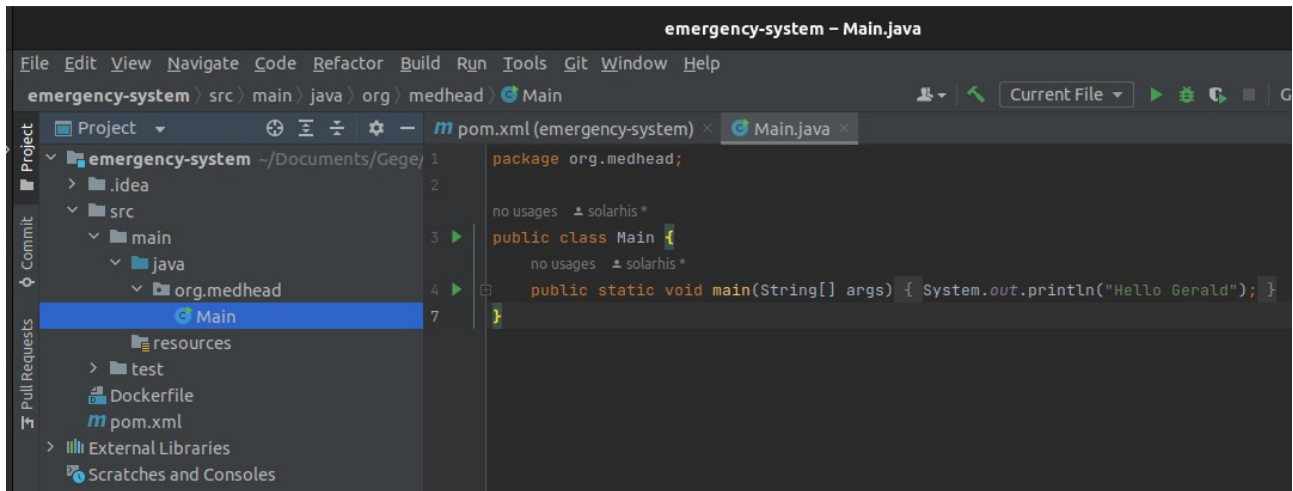
Finished: SUCCESS

Nota : il est possible de voir dans le log une ligne relative à précédemment commit effectué plutôt, ceci montre bien que le Jenkins s'est synchronisé avec GitHub

```
> git checkout -f 6a0c0d30b612cfeef961471a92d9dcadc3deb7e3 # timeout=10
Commit message: "first commit"
First time build. Skipping changelog.
```

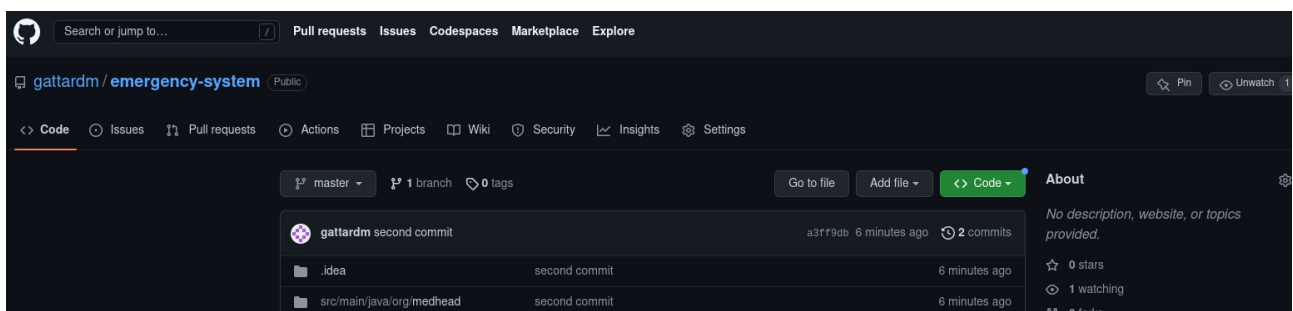
26 / 30

- Maintenant, il va falloir vérifier que Jenkins lance bien un build automatiquement, à chaque fois que le code est modifié sur le GitHub. Pour cela, il suffit de retourner sur IntelliJ IDEA, de modifier une classe et de la commit/push.



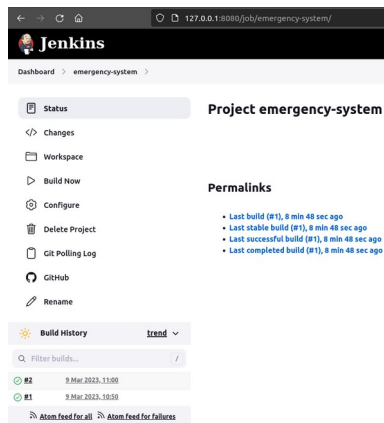
(ici le message commit est « *second commit* »)

- Une fois le commit/push réalisé, il est possible de se rendre sur le GitHub pour vérifier la mise à jour :

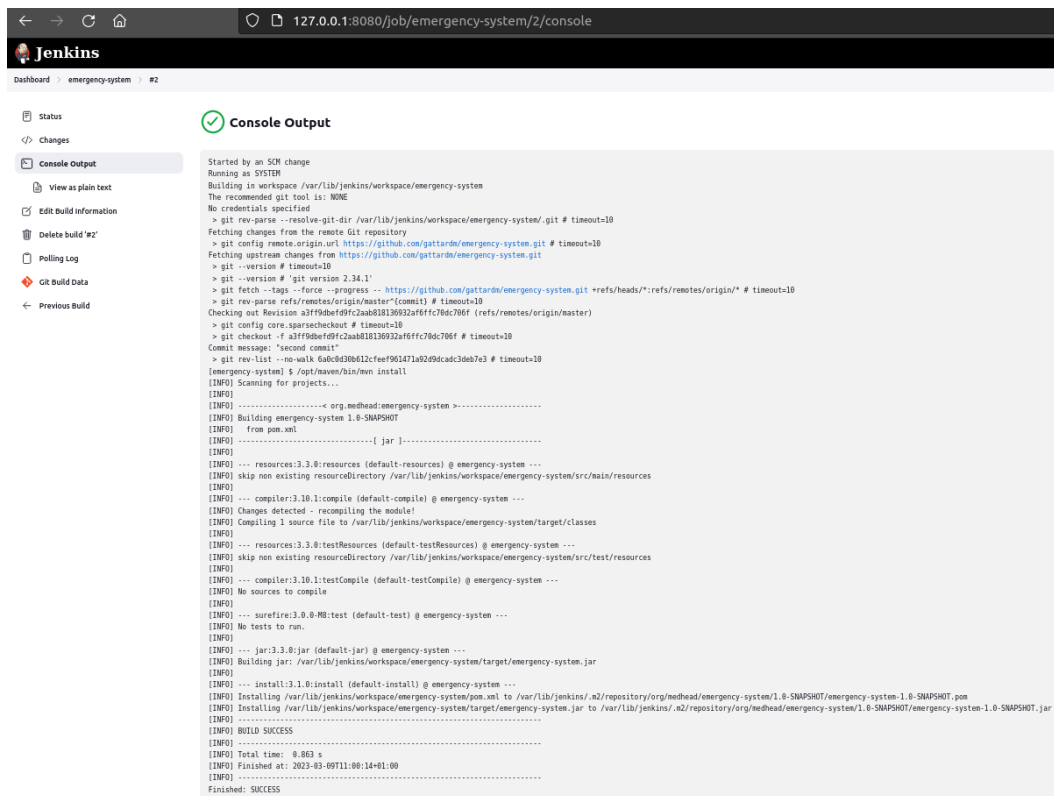


...on voit bien le message « *second commit* » affiché, c'est donc que le commit/push est fonctionnel.

- Maintenant, il faut retourner sur le Jenkins pour vérifier qu'il ait bien détecté la modification du code :



Effectivement, un second build (#2) a été exécuté. Ouvrir une console Jenkins pour voir ses logs :

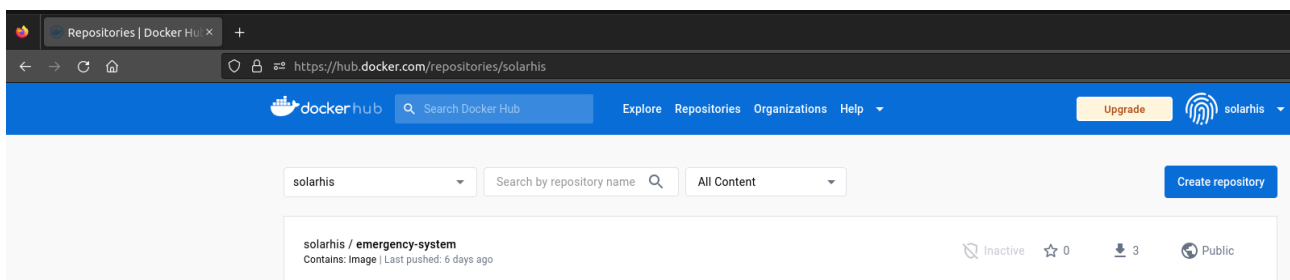


..il est possible de voir dans les logs du #2 build a donc été exécuté AUTOMATIQUEMENT, dès que Jenkins a détecté la précédente modification de code. Pour s'en assurer il est possible de lire, dans le log précédent, le message du second commit :

```
> git checkout -f a3ff9dbefd9fc2aab818136932af6ffc70dc706f # timeout=10
Commit message: "second commit"
```

- A partir d'ici, il va falloir brancher Docker à Jenkins. Pour cela, assurez-vous d'avoir bien exécuter les tâches relatives au paragraphe §II.C.6. *Configuration de Jenkins pour intégrer Docker* avant de continuer.
- Ouvrir la page Configuration de la tâche et y ajouter une nouvelle cible de haut-niveau, nommée *Docker Build and Publish*, puis renseigner le Repository Name correspondant au conteneur de votre **DockerHub**(ici *solarhis/emergency-system*) et les credentials pour y accéder :

- Puis relancer un build du job et aller vérifier sur le DockerHub que l'image Docker a bien été publiée :



Le pipeline est en place et prêt pour accueillir le code de l'application.

MedHead+