AGL :

## Maven :

* Organiser son projet:
* structuration en répertoires, nommage, emplacement des fichiers,
* Compiler
* exécuter les tests
* construire les artefacts : war, jar ou ear …
* déployer
* construire des rapports qualimétriques

Maven est issu de la fondation Apache, actuellement nous avons Maven 3.

Faciliter le processus de build

Fournir un système de build uniform

* build via le **POM (Project Object Model)** + un ensemble de plugins

Fournir des informations sur le projet, par exemple :

* les dépendances utilisées par le projet
* change log (branchement avec le gestionnaire de code)
* rapports de tests unitaires (incluant la couverture de code)

Encourager de meilleures pratiques de développement

* l’exécution et le reporting de test unitaire fait partie du cycle de build standard en Maven
* organisation de la structure de répertoires du projet

Maven gère les dépendances du projet :

* les dépendances sont déclarées dans le projet
* si elles ne sont pas présentes dans le projet, Maven les télécharge (et aussi les dépendances transitives)
* **Maven a un large dépôt (repository)** auquel se référer
* Permettre la construction des différents artéfacts du projet :

1. le logiciel issu du build (jar, war, ear, etc)
2. le site du projet
3. des rapports de test (yc de couverture)

* Pouvoir facilement inclure dans les phases de construction des phases de test, d’analyse de code en utilisant les plugins dédiés
* **Aller facilement vers de l’intégration continue**

La configuration du projet est centralisée dans un fichier appelé Project Object Model, autrement le pom en XML qui contient les déclarations des plugins, des dépendances etc.

Facilitation de l’édition de pom

* effective pom,
* ajout de dépendance par clic, génération de la portion de pom correspondante
* détection des erreurs syntaxiques

**ant** (~2000, initialement projet Apache)

* flexible, n’impose aucune convention ni structure de code
* bcp de choses à faire “à la main”, pas de gestion des dépendances

**gradle** (~2007)

* construit sur les concepts de maven et ant
* moins utilisé que Maven (logique) mais de + en + (ex Spring)
* plus rapide que Maven
* plus souple que Maven
* mais apprentissage plus long

Le build Maven suit un cycle de vie pour déployer et distribuer un projet.

Il y a 3 cycles de vie prédéfinis :

**default** : cycle de vie principal = déploiement du projet

**clean** : nettoie le projet (supprimer les fichiers générés lors du précédent build)

**site** : pour créer le site contenant la documentation du projet

Chaque cycle de vie est une séquence de phases.

Une phase est une étape du cycle de vie. Les phases d’un cycle sont exécutées en séquence. Pour exécuter la phase n, il y a nécessairement au préalable exécution des n-1 phases précédentes (celles associées à au moins un goal). Le cycle de vie default contient 23 phases, dont les principales sont :

* **validate**: vérification que toutes les infos nécessaires au build sont présentes
* **compile**: compilation du code source
* **test-compile** : compilation du code source de test
* **test** : exécution des tests unitaires
* **package** : packaging du code compilé en un format distribuable (jar, war, …)
* **verify** : lancement de vérifications sur les résultats des test d’intégration pour s’assurer que les critères de qualité sont satisfaits
* **install** : installation du package dans le dépôt local (pour utilisation locale par d’autres projets)
* **deploy** : copie le package vers un dépôt distant

Chaque plugin définit des goals

Les goals peuvent être associés aux phases du cycle de vie

Exemple. Le plugin compiler a comme goals :

* compile
* test-compile
* help

notation d’un goal

* plugin:goal

une phase qui n’est associée à aucun goal n’est pas exécutée

une phase avec plusieurs goals les exécute en séquence

Associations entre phases et goals : Spécifier le packaging :

Spécifier le packaging visé (par défaut : jar). Chaque packaging vient associer certaines phases à certains goals. Par exemple le packaging jar définit les associations suivantes :

| **Phase** | **plugin:goal** |
| --- | --- |
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | jar:jar |
| install | install:install |
| deploy | deploy:deploy |

Un plugin maven fournit des goals supplémentaires.

Les **plugins sont configurables**. La **configuration se fait dans le POM.** Souvent le plugin définit une configuration par défaut. Exemples :

* configuration des versions des dépendances à utiliser
* configuration du port à utiliser pour un serveur à lancer pour les tests

Sur Maven, deux hiérarchies parallèles : **l’une pour le code du projet, l’autre pour le test.**

Le nom “fully qualified” d’un projet est : **<groupId>:<artifactId>:<version>.**

Pour déclarer une dépendance, on utilise la carte d’identité minimale de la dépendance : groupId, artifactId, version

Mais pourquoi déclarer une dépendance ?

* **Maven a un plugin dédié à la gestion des dépendances**
* Ce plugin rend vraiment service !

Lors du build (phase …), Maven va regarder si la dépendance spécifiée existe dans le projet

* si oui parfait
* sinon il la télécharge

Maven gère aussi les dépendances transitives

**mvn package** lance le build lifecycle default

**mvn clean** package commence par faire un clean avant de lancer le build

Caractéristique majeure de Maven

**Un projet déclare ses dépendances, et Maven les “gère” :**

* Téléchargement des dépendances manquantes
* Téléchargement des **dépendances transitives (les dépendances des dépendances)**
* Analyse des dépendances : lesquelles manquent lesquelles sont (peut-être) en trop etc …

Maven choisit la définition **“la moins profonde” dans l’arbre (appelée “nearest définition").** (Pour la version).

Pour chaque dépendance déclarée, on peut déclarer sa portée (son scope)

**6 scopes :**

* **compile** (par défaut). Les dépendances de cette portée sont disponibles dans tous les classpaths du projet, et sont propagées aux projets dépendants
* **provided**. Les dépendances de cette portée sont ajoutées au classpath utilisés pour la compilation et le test, mais pas l’exécution (sont supposées fournies par l’environnement d’exécution). Ne sont pas propagées.
* **runtime**. Dépendance uniquement requise à l’exécution, donc dans le classpath runtime et test mais pas compile.
* **test**. Dépendance requise uniquement pour le test (compilation et exécution du test). Exemple : mockito et junit.
* **system**. Comme provided mais le jar est fourni (et ne sera pas cherché dans le repository).
* **import**. A placer seulement dans une dépendance dans la section <dependencyManagement>, avec un type POM. Indique que cette dépendance doit être remplacée par toutes les dépendances affectives trouvées dans le POM.

Classification des dépendances en :

* **used and declared**
* **used and undeclared** : dépendances transitives. Il est mieux de les ajouter explicitement en dépendances pour mieux en contrôler les versions
* **unused and declared** : dépendances qui ne sont pas utilisées après une analyse du code compilé. Mais peuvent être utilisées quand même à l’exécution. A manier avec précaution

src/main/webapp - Web application sources

LICENSE.txt - Project’s license

NOTICE.txt - Notices and attributions required by libraries that the project depends on

**2 sortes de plugins :**

* **build** : ceux qui sont exécutés durant le build (configurés dans la section build du pom) exemple : failsafe (pour les tests d’intégration)
* **reporting** : ceux qui sont utilisés durant la génération du site (configurés dans la section reporting du pom) exemple : plugin javadoc

**Tout POM peut hériter d’un autre POM**. Par défaut tous POM hérite du super-POM.

**simplest POM** : celui qui est réellement défini dans le projet maven peut construire le pom effectif (effective pom)

* mvn help:effective-pom
* combine le pom hérité, le super-pom et le simplest pom

## Junit :

Différentes propriétés à tester

* satisfaction des fonctionnalités requises
* qualité de service (temps de réponse, utilisation mémoire, ...)
* robustesse
* sûreté de fonctionnement
* utilisabilité

Plusieurs niveaux (échelles)

* **Unitaire / Intégration / Système / Acceptation (ou recette)**

Différents niveaux d’accessibilité

* Test boîte noire (souvent fonctionnel) / Test boîte blanche (souvent structurel) / Test boîte grise ?

Plusieurs types classiques

* test fonctionnel / test de non-régression / test de montée en charge

Origine : Xtreme programming (test-first development), méthodes agiles

* framework de test écrit en Java
* open source : www.junit.org

Objectifs

* **test d’applications en Java / faciliter la création des tests / tests de non régression**

Enchaîne l’exécution des méthodes de test définies par le testeur. Facilite la définition des tests grâce à des assertions, des méthodes d’initialisation et de finalisation. Permet en un seul clic de savoir quels tests ont échoué/planté/réussi. Junit lance seulement les tests.

**Le framework définit toute l’infrastructure nécessaire pour** :

* écrire des tests
* définir leurs oracles
* lancer les tests

Utiliser Junit :

* **définir les tests**
* **s’en remettre à JUnit pour leur exécution**
* **ne pas appeler explicitement les méthodes de test**

On crée une ou plusieurs classes destinées à contenir les tests : les classes

de test. On y insère des méthodes de test.

Une méthode de test

* fait appel à une ou plusieurs méthodes du système à tester (communément appelé **SUT, System Under Test**),
* ce qui suppose d’avoir une instance d’une classe du système à tester (la

création d’une telle instance peut être placée à plusieurs endroits, voir plus loin),

* inclut des instructions permettant un verdict automatique : les assertions.

Les classes de tests : Contient les méthodes de test (sans ordre). Peut contenir des méthodes particulières pour positionner l’environnement de test (souvent stocké en attribut)

Une méthode de test s’intéresse à une seule unité de code/un seul comportement

* **doit rester courte**
* Les méthodes de test sont indépendantes les unes des autres (pas d’ordre)

Junit versions <4 : les méthodes de test commencent par le mot test

JUnit versions ≥4 : **annotées @Test**

Les méthodes de test seront appelées par Junit, **dans un ordre supposé quelconque.**

**Sont sans paramètres et sans type de retour.**

Les verdicts sont définis grâce aux assertions placées dans les cas de test.

* **Pass (vert) : pas de faute détectée**
* **Fail (rouge) : échec, violation d’assertion**
* **Error : le test n’a pas pu s’exécuter correctement (exception inattendue)**

L’exécution d’une méthode test **s’arrête à la première assertion violée**. Donc il est déconseillé d’écrire plusieurs assertions dans le même test.

JUnit 5 : Méthodes avec annotations **@BeforeEach et @AfterEach** ;

JUnit 4 : Méthodes avec annotations @Before et @After ;

JUnit 3 : Méthodes appelées setUp et tearDown

* exécutées avant/après **chaque méthode** de test (l’exécution est pilotée par le framework, et pas le testeur)
* possibilité d’annoter plusieurs méthodes (ordre d’exécution indéterminé)
* publiques et non statiques

JUnit 5 : Méthodes avec annotations **@BeforeAll et @AfterAll**

JUnit 4 : Méthodes avec annotations @BeforeClass et @AfterClass (pas en JUnit 3)

* exécutées avant (resp. après) **la première** (resp. dernière) méthode de test
* une seule méthode pour chaque annotation
* **publiques et statiques** (sauf en JUnit 5 si le cycle de vie est perClass avec : @TestInstance(LifeCycle.PER\_CLASS))

L’annotation @Test peut prendre en paramètre le type d’exception attendue @Test(expected=monexception.class) :

* Succès ssi cette exception est lancée.

public static <T extends Throwable > T assertThrows (Class <T > expectedType , Executable executable )

Décortiquons déjà le premier paramètre :

* type : Class<T>
* contraint par : T extends Throwable
* donc on attend une classe d’exception
* Introspection en Java : existence d’une classe java Class, paramétrée par un type T ; permet entre autres d’obtenir une instance de la classe (de type T donc) et aussi de manipuler des classes dans un programme Java ...

Deuxième paramètre :

* Attend une instance d'un exécutable

L’annotation **@Test** peut prendre en paramètre un timeout : @Tes**t (timeout=10)** (en ms).

* Fail si la réponse n’arrive pas avant le timeout.

Annotation **@Ignore** (paramètre optionnel : du texte) pour ignorer le test

Annotation **@Disabled** (paramètre optionnel : du texte) pour désactiver le test

**assertThat([value], [matcher statement]);**

* exemples :
* assertThat(x, is(3)) ;
* assertThat(x, is(not(4))) ;
* assertThat(responseString, either(containsString("color")).or(containsString("colour"))) ;
* assertThat(myList, hasItem("3")) ;

On choisit entre **not(s), either(s).or(ss), each(s).**

**AssumeThat(File.separatorChar, is(“/”))**

* L’assertion **suivante sera ignorée si la supposition n’est pas vérifiée**

**Les tests paramétrés :** utiliser des *méthodes de test avec des jeux de données de test*

*différents*. La classe de test est annotée avec @RunWith(Parameterized.class). La méthode qui retourne les paramètres doit être statique.

**Le principe des tests unitaires** : tester une unité logicielle en isolation. Par exemple une classe ou un groupe de classes. Exécution initiale : s’assurer de la qualité d’une unité logicielle. **Non régression : après chaque modification de l’unité logicielle, on relance**

**les tests unitaires.** **Exécution “continue” : placement des tests sur une plateforme CI.**

Après les tests unitaires viennent les tests d’intégration, de systèmes et de recettes.

## 

## Mocks :

**Pourquoi simuler ?** (Pas de blagues de cul Gatié)

* **L’environnement du SUT est complexe ou coûteux** à mettre

en place (environnement matériel, base de données, ...).

* **Mise en place de situations exceptionnelles difficiles à déclencher** (out of memory, ...).
* L’environnement du SUT **n’est pas encore disponible ou fiabilisé**.
* Le SUT **appelle du code lent.**
* Le SUT fait appel à des **méthodes non déterministes** (fonction de l’heure, de nombres générés aléatoirement, ...)
* On simule alors tout ou partie de l’environnement qui pose problème.

Nous n’écrivons pas les simulateurs à la main, nous les injectons :

Injecter les simulateurs

* Remplacer le vrai élément par le simulateur dans le code sous test
* MAIS sans modifier le code sous test
* Utiliser de **l’injection de dépendance**
* Prévoir l’injection de dépendance lors de l’écriture du code ! (testabilité)

**Une doublure de test est un objet factice se substituant lors du test aux objets réels.**

Ces objets peuvent être écrits à la main ou générés. Les objets factices doivent partager la même interface que les vrais objets.

Différents types de doublures : **Dummy / Stub / Fake / Spy / Mock**

**Dummy** : Objets vides qui n’ont pas de fonctionnalités implémentés. Sont transmis mais jamais réellement utilisés. Utilisés pour remplir des listes de paramètres.

**Fake** : Le fake implémente de manière simpliste le comportement attendu d’une classe. Le fake est générique : il n’est pas spécifique à un test. Met en place des raccourcis qui le rendent inutilisable en production.

**Spy** : Doublure capable de vérifier l’utilisation qui en est faite.

**Stubs** : Bouchon de test, classe utilisée pour en simuler une autre. **Fournit des réponses pré-définies aux appels réalisés lors du test**. Écrit grâce à la connaissance de la classe à simuler.

**Mocks** : Objet pré-programmés avec des présuppositions qui forment la spécification des appels qu’ils sont censés recevoir. Le testeur configure le mock de manière à lui donner le comportement souhaité. Permet de réaliser une vérification comportementale.

Mockito permet de générer des mocks.

Quand créer un mock :

**Quand on veut éviter (pour des raisons vues précédemment) de faire appel au vrai environnement du SUT.** Par exemple quand certains éléments de l’environnement

dépendent du temps, ou d’un aléa.

Dans le test, au lieu d’utiliser une instance d’une classe "réelle" de l’environnement (et qu’on souhaite remplacer), on demande à Mockito d’utiliser une doublure. On paramètre la doublure pour qu’elle ait un comportement permettant le test. On écrit le test, qui utilisera donc la doublure.

Création d’un mock : utilisation de la méthode statique mock ou de l’annotation **@Mock**. Nécessite la classe à mocker ou son interface

Pour utiliser les annotations mockito : ajouter **@ExtendWith(MockitoExtension.class)** sur la classe de test. *On ne peut pas utiliser l’annotation @Mock sur une variable locale* (d’une méthode de test), seulement sur un attribut (normal, c’est une annotation).

**Il faut ensuite paramétrer le mock, sinon il retournera 0 ou nul ou false etc.**

On décrit le comportement du mock avec la méthode **when**

On exprime quelque chose du genre : Mockito, quand (when !) le mock recevra tel appel, alors il faut répondre ceci

* On peut faire des vérifications comportementales avec la méthode **verify**

On vérifie quelque chose du genre : Mockito, est-ce que telle méthode a bien été appelée au moins une fois avec tel paramètre ?

* On peut spécifier des comportements à vérifier un peu complexes grâce à des **matchers**

**Mock et spy :**

On veut espionner un objet instance d’une classe réelle (pas un mock) ...

* ... et éventuellement en altérer le comportement
* **Par exemple pour juste mocker une méthode** (remplacer le comportement réel par un autre, et garder tous les autres comportements réels)
* On utilise alors un spy, dont le comportement par défaut est celui de la classe de l’objet espionné

**Limites :**

* On ne peut pas mocker les méthodes privées, ni les méthodes final
* On ne peut pas mocker les méthodes equals et hashcode
* On ne peut pas mocker les classes final

**Si un vérify échoue alors le test entier échoue.**

**Argument matchers :**

Permettent une spécification de paramètres flexible dans les when ou les verify

* Ne plus utiliser Matchers (deprecated pour cause de conflit de nommage avec hamcrest)
* limitation : si on utilise des argument matchers, tous les arguments doivent être des argument matchers.

PowerMock permet de pallier les limites de mockito. **C’est un framework de test** permettant de **contourner les limitations** classiques des frameworks de mock comme Mokito. On peut mocker les méthode privées, statiques et ou final, mais aussi des classes finales.

Injecter les mocks :

* **Injection par constructeur** : l’objet que l’on veut simuler est reçu en paramètre du constructeur SUT.
* **Injection par accesseur/mutateur** : l’objet que l’on veut simuler est injecté dans le SUT via l’appel à un accesseur en écriture.

Option possible en Mockito (dans certains cas) : l’annotation **@InjectMocks.**

Les mocks permettent de simuler certains comportements, afin d’en tester d’autres :

Situations où les mocks sont utiles :

* présence d’aléatoire ou dépendance au temps
* accès à du code lent
* accès à des bases de données
* accès à du code non encore développé
* test en isolation

Les outils comme Mockito permettent de créer facilement des simulateurs spécifiques à un test : les mocks. Ces mocks doivent être en général injectés dans le système sous test (SUT pour les zboubs). Si l’injection n’est pas possible, le système est alors difficilement testable.

## Qualimétrie :

**Les tests ne servent pas à mesurer la qualité du code**, il ne font qu’augmenter la confiance dans le logiciel. De nombreux tests ne sont pas garants de qualité.

Une mesure classique de qualité des tests est l’analyse de la

couverture du code qu’ils couvrent. Plusieurs mesures de couverture peuvent être utilisées. La plupart des mesures se basent sur une représentation sous forme de graphe de flot de contrôle ou de données des programmes testés.

C’est un graphe orienté et connexe (N, A, e,s) :

* N : ensemble de sommets

sommet = bloc d’instructions exécutées en séquence

* E : relation N × N

débranchements possibles du flot de contrôle

* e : sommet d’entrée du programme
* s : sommet de sortie du programme

Un arc entre deux noeuds n1 et n2 représente le transfert du contrôle de n1 à n2 :

* avec étiquetage par une condition booléenne c : transfert si c est vraie ;
* sans étiquetage : transfert inconditionnel.

Les noeuds représentent soit :

* un bloc d’instructions séquentielles - graphiquement un rectangle ;
* un prédicat (nœud de décision) qui permet le transfert du contrôle - graphiquement un losange.

Deux noeuds particuliers :

* un unique nœud d’entrée (sans prédécesseur)
* un unique nœud de sortie (sans successeur)

**La couverture ne suffit pas à estimer la qualité du code.**

L’analyse par mutation permet de mesurer la qualité d’un test. Un bon test détecte des erreurs si il y en a.

Principe du test par Mutation :

* On introduit volontairement des erreurs dans le SUT
* On regarde si les tests les détectent

Les mutants simulent des fautes classiques. Modifient le comportement du programme donc y insèrent des fautes. Les tests doivent trouver la faute : exécution des jeux de test sur les mutants.

But du test : Tuer les mutants

**Score = nb mutants tués / nb mutants total**

Analyse des mutants survivants.

**Exemple d’opérateur de mutation :**

* changer un < en <=
* changer un + en -
* altérer les valeurs (changer a = 1; en a = 2)
* changer un && en ||

**Tests incomplets** : il manque un test/une assertion permettant de détecter cette faute.

**Mutant équivalent** : La mutation opérée a produit un programme SUT équivalent au SUT.

* Certains mutants équivalents peuvent être détectés
* Mais dans le cas général indécidable

Mutant survivants car pas de couverture.

Il faut beaucoup de mutants, l’analyse par mutation est longue :

* temps de génération des mutants
* temps d'exécution de tous les jeux de tests sur chaque mutant
* temps de synthèse des résultats

**Des indicateurs "classiques" de complexité/qualité :**

* Nombre total de lignes de codes
* Nombre de lignes de codes par classe
* Nombre de méthodes par classe
* Nombre total de méthodes
* Ratio lignes de codes/nombre de méthodes
* Ratio lignes de codes/nombre de classes
* Ratio lignes de commentaires/lignes de codes

Indice de spécialisation : (NORM\*DIT)/NoM

* NORM = Nombre de méthodes redéfinies
* DIT = Profondeur dans l’arbre d’héritage
* NoM = Nombre de méthodes de la classe

**Exemples d’autres métriques** : couplage, instabilité d’un package, niveau d’abstraction, complexité cyclomatique.

**Code smells :** Soupçons de mauvaises pratiques de conception logicielle, **pouvant conduire à des défauts(comme des problèmes de maintenabilité etc)**.

**Exemples de Bad Smells** : **Feature envy** (jalousie de caractéristique) : quand une méthode

(ou plusieurs) d’une classe est plus intéressée par les membres (attributs et méthodes) d’une autre classe que les siens.

Quand on détecte un potentiel défaut par des indicateurs : Si on en déduit qu’il y a un défaut, on fait un **refactoring.**

Refactoring (réusinage) : opération consistant à retravailler le code source d’un programme informatique :

* pour obtenir un **code fonctionnellement équivalent**
* mais avec une **lisibilité améliorée** (ou d’autres caractéristiques extra-fonctionnelles améliorées)

Exemple d’un refactoring :

Suite à détection d’une feature envy

* Déplacer une (partie de) méthode d’une classe dans une autre si elle utilise plus les données de celle-ci que les siennes.
* Suite à une détection de Blob
* Séparation du blob en plusieurs classes, en suivant une séparation des préoccupations et hiérarchiser au besoin

Conclusion :

Indicateurs de qualité des tests

* Critères de couverture du code source par les tests
* analyse par mutation

Indicateurs de complexité/qualité

* indicateurs nombreux, éventuellement interdépendants
* se calculent par analyse statique du code
* à analyser avec précaution

## Docker :

**Le DevOps : Les différentes étapes CBTRDOM**

* Code
* Build
* Test
* Release
* Deploy
* Operate
* Monitor

L’impasse des serveurs et des OS :

* Un serveur applicatif pour chaque système d’exploitation
* Un serveur applicatif pour chaque application
* Un serveur backup en cas de problèmes

Machine virtuelle : Densité infrastructure :

* 1 seule grosse machine
* n machines virtuelles

**Virtualisation lourde :**

Good

* Multi OS
* Isolée
* Ressource réservée
* Consommation énergétique optimisée

Bad :

* Start lent
* Ressources

**Conteneur d’application :**

* Virtualisation légère (non réservée)
* Volatile, permet la scalabilité on-demand
* Permet à une équipe de s’abstraire de l’OS
* Au coeur de la révolution DevOps

Hyperviseur vs Conteneur :

* Point de vue applicatif : c’est la même architecture logicielle
* Conteneur transportable : création d’image distribuable

**Docker**

• Platform as a Service (**PaaS**) v2

* PaaS v1 (Heroku) simple espace de déploiement cloud de site web

**• Immutabilité**

* On ne modifie pas
* On redéploye une nouvelle image
* Une philosophie de déploiement

• Reproductibilité

* ça marche sur ma machine (sic)
* Ça marchera chez vous !

**StateLess/StateFull :**

* **State Full = s'allumer dans l’état précédent (data management)**
* **State Less = toujours les mêmes actions réalisées de manière indépendantes de l’historique (traitement requête http), aucun état nécessaire.**

1 Conteneur = 1 processus :

* LAMP = 3 Conteneurs (Apache, MySQL, PhP)
* docker-compose pour assembler tous les composants (stack) via un fichier manifest

**Setup :**

* Un fichier batch de configuration (DokerFile)
* Un fichier d’assemblage (docker-compose.yml)
* Un build

**Dockerfile :**

FROM source

RUN exécute

ADD ajoute

WORKDIR home

EXPOSE port

VOLUME disque

CMD run initial

**Docker-compose.yml : (Fichier d’assemblage)**

* Définir des services
* Les lier entre eux :
* Origine
* Dépendance
* Variables d’environnement
* Statut

## Gitlab CI :

Décomposition/Organisation :

* Production
* Pré-Production
* Développement
* Intégration
* Tests
* Staging

**Gitlab est plus qu’un gestionnaire de versions.**

Gestion = Automatisation :

* Long à faire à la main
* Enchaînements de différents niveaux conceptuels
* Peut avoir une grande diversité ou branchement

Découvrir les bugs tôt : feedbacks rapides, réguliers.

**Cycle de vie du bug : DACCRC**

1. Découverte du bug (par exécution d’un test)
2. Analyse (pourquoi ce bug ?)
3. Catégorisation du bug (critique, majeur ou mineur)
4. Correction du bug (pas toujours du code)
5. Retest, vérifications (retour phase correction possible)
6. Clôture du bug

**L‘intégration continue :**

* **Fréquent**
* Équipe informée
* Développeur ne peuvent rester seuls face à leur feature
* Développeurs se synchronisent souvent ce qui évite les divergences entre les branches
* **Process rodés**
* Prépare la livraison sans souci
* **Simplifie la quality analysis**
* Même si c’est une grosse feature, intégration partielle (avec indicateurs de qualité maj)

**.gitlab-ci.yml :**

Désormais, créé automatiquement lors de la génération d’un nouveau GIT, décrit un pipeline et ses étapes.

Java CI pour Maven/Java/JUnit.

**Continuous Delivery CD** :

* Réduire au maximum le temps entre la création de la ligne de code et l’utilisation réelle par les utilisateurs finaux

Conclusion : On-Going Process

## DEVOP en entreprise :

Les outils dans un projet :

1. **Mise en place des environnements : Docker**

Flexibilité et rapidité, installation des outils nécessaires sur les VM. Le logiciel « Docker » est une technologie de conteneurisation qui permet la création et **l'utilisation de conteneurs Linux**. Avec la technologie Docker, vous pouvez traiter les conteneurs comme des machines virtuelles très légères et modulaires. En outre, ces conteneurs vous offrent une grande flexibilité : **vous pouvez les créer, déployer, copier et déplacer d'un environnement à un autre**, ce qui vous permet d'optimiser vos applications pour le cloud.

1. **Mise à disposition des sources : Git**

Mise à disposition des sources, permet de gérer les versions des applications. Outil de collaboration qui permet le développement parallèle.

1. **L’industrialisation : CI/CD**

L’industrialisation, automatise les tâches, permet le build et le déploiement, permet l’analyse du code. Permet le test automatique. Permet de chaîner le build et le déploiement des sources. SonarQube. CheckMarx. Dependency Check.

La qualimétrie du code : dépend de sa facilité d’utilisation, du budget et de l’équipe.

1. **Les tests**

Les bouchons : une implémentation/service/fichier qui permet de tester le bon fonctionnement, à ne pas couter cher et sert aussi à ne pas faire de bétises.

Les **TUA (tests unitaires automatisés)** dépendent du budget, de la facilité d’implémentation et de la complexité du projet. Ils permettent d’assurer la non-régression ( ont pour but de s’assurer que les modifications effectuées n’ont pas entraîné d’effet de bord, en altérant les parties du code non modifiées.), facilitent le refactoring, et permettent le TDD.

Les **TDD (tests driven development)** écrire les tests avant de coder. Facilité grandement le refactoring.

Mais il faut aussi : un bon IDE, une gestion des tâches, un partage des documents et une sauvegarde de mot de passe.