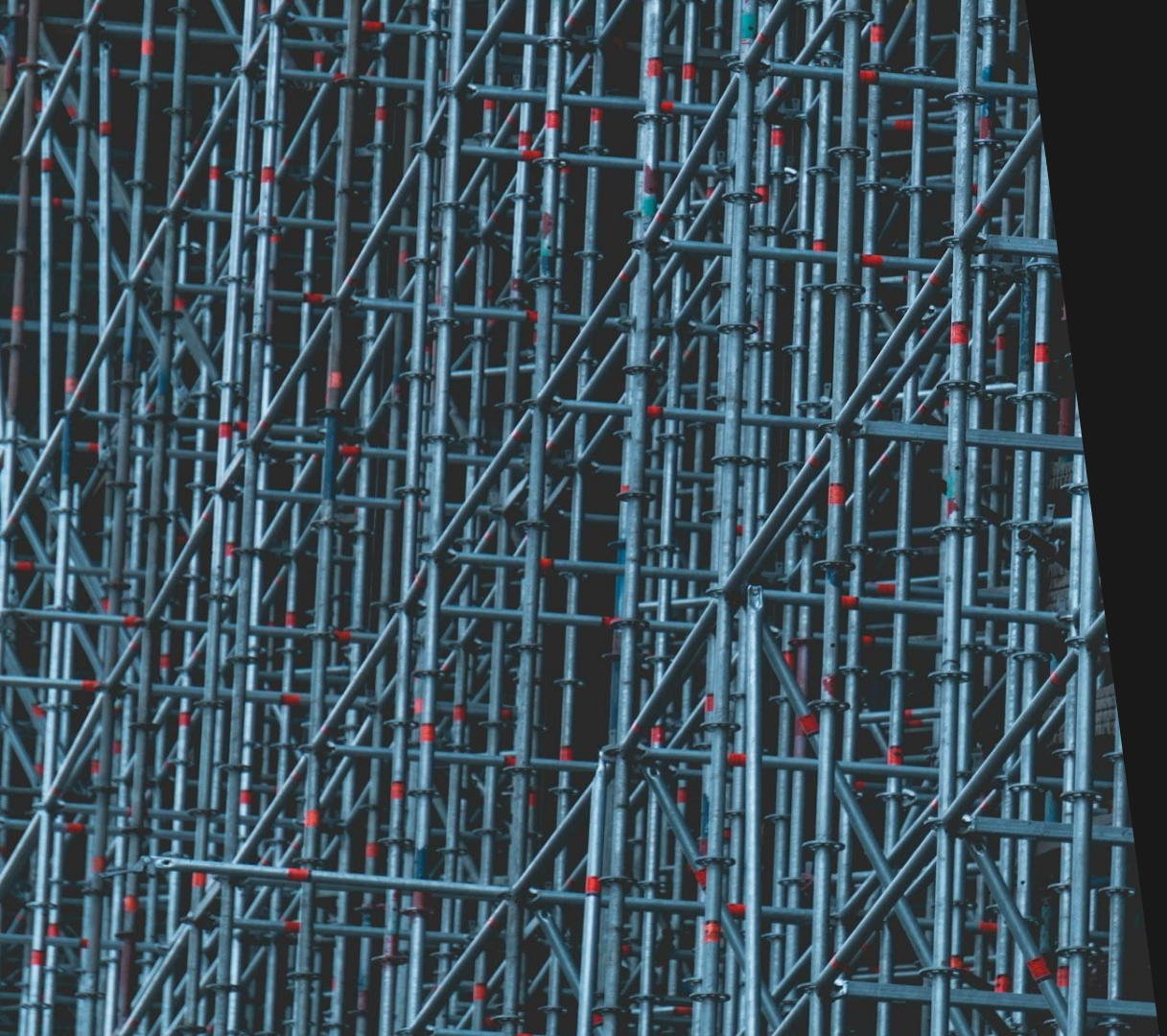
Maîtrise des Grands Projets Informatiques

Plan du cours

- Introduction aux Concepts de Base de l'Organisation des Entreprises
- Cycle de vie et cycle de développement logiciel
- Activités de développement logiciel
- Avant-projet
- Collecte des besoins : cas d'utilisation, user's story
- Analyse Conception
- **Codage**
- Intégration
- Déploiement et production : plateforme de déploiement continu, surveillance des applications, conteneurisation en production (docker)
- Activités de gestion de projets



Scaffolding

Qu’est-ce que le scaffolding

Le scaffolding désigne le processus de génération automatique d’une structure de base pour un projet logiciel.

Ils servent à gérer les dépendances, configurer l'environnement, compiler et packager l’application.

Pourquoi les utiliser

- Gain de temps: Évite de recréer manuellement la structure de base à chaque nouveau projet.
- Cohérence: Assure une structure de projet standardisée et cohérente au sein d'une équipe ou d'une organisation.
- Meilleures pratiques: Intègre souvent des bonnes pratiques de développement et des conventions de codage.
- Productivité: Permet aux développeurs de se concentrer sur la logique métier plutôt que sur les aspects techniques de configuration.

Maven

Maven est un outil de gestion de projet et d'automatisation de build pour les projets Java.

Il repose sur le concept de Project Object Model (POM) et permet de gérer les dépendances, de compiler le code source, de packager les artefacts (fichiers JAR, WAR), et d'exécuter des tests.

Structure d’un projet Maven

Un projet Maven est structuré autour du fichier pom.xml, qui définit :

- Les dépendances à inclure dans le projet (ex : des bibliothèques tierces).
- Les plugins nécessaires pour différentes tâches (compilation, test, etc.).
- Les phases du cycle de vie de construction (compile, test, package, install, deploy).

Maven: Caractéristiques

- Gestion des dépendances: Il télécharge automatiquement les bibliothèques nécessaires depuis des référentiels centraux ou privés.
- Référentiels: Maven utilise des référentiels pour stocker les artefacts (bibliothèques, plugins, etc.).
- Cycle de vie du projet: Un cycle de vie standard pour les projets est défini, qui inclut des phases telles que la compilation, le test, l'emballage et le déploiement.
- Plugins: Ils permettent d’ajouter des fonctionnalités supplémentaires, comme la génération de rapports, l'exécution de tests, etc.

Maven: Exemple d'un fichier pom.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.example </groupId>
   <artifactId> demo-project </artifactId>
   <version> 1.0-SNAPSHOT </version>

   <dependencies>
       <dependency>
           <groupId> org.springframework </groupId>
           <artifactId> spring-core </artifactId>
           <version> 5.3.0</version>
       </dependency>
   </dependencies>
</project>
```

Maven: Gestion des dépendances

- groupId : Identifie le groupe de la dépendance.
- artifactId : Identifie l'artefact spécifique.
- version : Spécifie la version de l'artefact.
- scope (optionnel) : Définit le contexte dans lequel la dépendance est utilisée (par exemple, compile, test, runtime, etc.).

Maven: Exemple de gestion des dépendances

```
<dependencies>
    <dependency>
        <groupId> junit</groupId>
        <artifactId> junit</artifactId>
        <version> 4.12</version>
        <scope> test</scope>
    </dependency>
    <dependency>
       <groupId> org.springframework </groupId>
        <artifactId> spring-core </artifactId>
        <version> [5.3.9,5.4) </version>
    </dependency>
</dependencies>
```

Maven: Référentiel

Les référentiels (repositories) sont des emplacements où les artefacts (bibliothèques, plugins, etc.) sont stockés et récupérés.

Il existe deux types de référentiels : les référentiels centraux (publics) et les référentiels privés (locaux ou distants).

Par défaut Maven utilise le référentiel central de Maven.

Maven: Référentiels déclaration

Il est possible de déclarer les référentiels dans le fichier pom.xml du projet.

Il est également possible d'ajouter des repository dans le fichier **settings.xml** du répertoire **~/.m2** local.

Maven: Exemple de déclaration de référentiels pom.xml

```
<repositories>
   <repository>
       <id>central</id>
       <url>https://repo.maven.apache.org/maven2 </url>
   </repository>

   <repository>
       <id>my-private-repo</id>
       <url>https://my-private-repo.com/maven2 </url>
   </repository>
</repositories>
```

Exemple de déclaration de référentiels settings.xml

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
          http://maven.apache.org/xsd/settings-1.0.0.xsd" >

   <localRepository> /chemin/vers/mon/local-repo </localRepository>

</settings>
```

Maven: Gestion des référentiels

Il est possible d’ajouter plusieurs options aux référentiels:

- Authentification
- Profils
- Proxies

Cycle de vie et phases de Maven

Le lifecycle (cycle de vie) de Maven est un ensemble de phases séquentielles qui permettent de construire, tester et déployer un projet.

Maven offre trois cycles de vie principaux : default, clean, et site.

Maven cycle de vie: Default

Les phases les plus importantes du cycle de vie "default" sont :

1. validate
2. compile
3. test
4. package
5. verify
6. install
7. deploy

Maven cycle de vie: Default Phases

1. validate : vérifie que le projet est correct et que toutes les informations nécessaires sont disponibles.
2. compile : compile le code source du projet.
3. test : exécute les tests unitaires en utilisant un framework de test comme JUnit. Le code est compilé, mais pas encore packagé.
4. package : empaquette le code compilé dans un format distribué (fichier JAR, WAR, etc.).

Maven cycle de vie: Default Phases

1. verify : exécute des vérifications pour s'assurer que le package est valide et qu'il respecte les critères de qualité.
2. install : installe le package dans le dépôt local (typiquement, votre machine) pour qu'il puisse être utilisé comme dépendance dans d'autres projets locaux.
3. deploy : copie le package final dans un dépôt distant où d'autres développeurs ou projets peuvent y accéder.

Maven cycle de vie: Default Phases

Les phases s’exécutent dans cet ordre.

Si on fait appel à la phase de test, les phases précédentes (validate, compile) seront également appelées.

Maven cycle de vie: clean

Le cycle de vie clean est utilisé pour nettoyer le répertoire de travail du projet (souvent le répertoire target), afin de s'assurer que le build démarre avec un environnement propre.

Les phases les plus importantes du cycle de vie "clean" sont :

1. pre-clean : exécute des actions avant le nettoyage.
2. clean : supprime les fichiers générés par les builds précédents (généralement, supprime le dossier target).
3. post-clean : exécute des actions après le nettoyage.

Maven cycle de vie: site

Le cycle de vie site gère la génération de la documentation et des rapports pour le projet. Il se compose de ces phases :

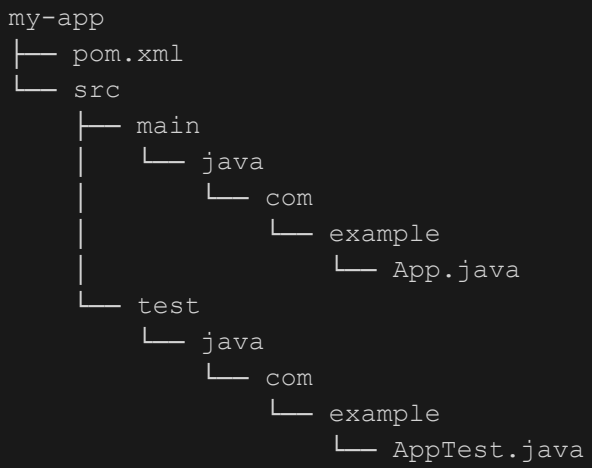
1. pre-site : exécute des actions avant la génération du site.
2. site : génère la documentation du projet.
3. post-site : exécute des actions après la génération du site.
4. site-deploy : déploie le site généré dans un serveur distant.

Maven Archetype

Un archetype Maven c'est un template de projet préconfiguré qui vous fournit une structure de base, des fichiers de configuration et des dépendances initiales.

```
mvn archetype:generate
```

Maven structure de projet quickstart



Maven:
Exemple de commandes

- mvn compile : compile le code source.
- mvn test : compile le code source, puis exécute les tests.
- mvn package : compile le code, exécute les tests, puis empaquette le projet (génère un JAR ou WAR).
- mvn install : exécute toutes les phases jusqu'à l'installation dans le dépôt local.
- mvn clean : nettoie les fichiers générés par un build précédent.

Maven: Plugins

Les plugins Maven sont des extensions qui ajoutent des fonctionnalités spécifiques au cycle de vie de build de Maven.

Ils permettent d'automatiser diverses tâches de développement, telles que la compilation, le test, l'emballage, le déploiement, la génération de documentation.

Exemple d’utilisation de plugins dans le pom.xml

```
<build>
   <plugins>
       <plugin>
           <groupId>org.apache.maven.plugins </groupId>
           <artifactId> maven-compiler-plugin </artifactId>
           <version> 3.8.1</version>
           <configuration>
               <source>${maven.compiler.source} </source>
               <target> ${maven.compiler.target} </target>
           </configuration>
       </plugin>
   </plugins>
</build>
```

Exemple d’utilisation de plugins: commande Maven

```
mvn clean package -Dmaven.compiler.source= 17
-Dmaven.compiler.target= 17
```

Maven: Avantages et inconvénients

- Grand écosystème de plugins
- Convention over configuration: Impose une structure standard de projet.
- Moins flexible que d’autres outils
- Le XML…

Gradle

Gradle est un outil de build moderne qui combine la flexibilité de scripting d'Ant et la gestion des dépendances de Maven.

Gradle supporte plusieurs langages de programmation comme Java, Groovy, Kotlin, et Scala.

Il est particulièrement apprécié pour les projets Android et son approche déclarative à travers un fichier de configuration de type DSL.

Structure d’un projet Gradle

Un projet Gradle est structuré autour du fichier build.gradle, qui définit :

- Les dépendances à inclure dans le projet (ex : des bibliothèques tierces).
- Les plugins nécessaires pour différentes tâches (compilation, test, empaquetage, etc.).
- Les tâches de build que vous pouvez personnaliser et exécuter (compilation, test, package, etc.).

Gradle: Exemple d'un fichier build.gradle

```
plugins {
   id 'java'
}

group 'com.example'
version '1.0-SNAPSHOT'

repositories {
   mavenCentral()
}

dependencies {
   implementation 'org.springframework:spring-core:5.3.20'
   testImplementation group: 'junit', name:'junit', version: '4.13+'
}

tasks.named('test', Test) {
   useJUnitPlatform()
}
```

Gradle: Gestion des dépendances

- group : Identifie le groupe de la dépendance.
- name : Identifie l'artefact spécifique.
- version : Spécifie la version de l'artefact.

Le contexte de la dépendance est fait à l’aide des mots clés: implementation, testImplementation, runtimeOnly, api

Gradle: Exemple de gestion des dépendances

```
dependencies {
    implementation 'org.springframework:spring-core:5.3.9'
    testImplementation 'junit:junit:4.12'
    runtimeOnly 'org.apache.commons:commons-lang3:3.11'
}
```

Gradle: Exemple de référentiel

```
repositories {
   mavenCentral()
   jcenter()
   maven {
       url "https://my-private-repo.com/maven2"
       credentials {
           username = "myUsername"
           password = "myPassword"
       }
   }
}
```

Gradle: Tasks

Les tâches Gradle sont des unités de travail définies dans un script de build Gradle, qui peuvent être exécutées pour accomplir des actions spécifiques, telles que la compilation, le test, l'emballage et le déploiement du code.

Gradle: Exemple de Task

```
task hello {
    doLast {
        println 'Hello, Gradle World!'
    }
}

task greet {
    dependsOn hello
    doLast {
        println 'This is a custom greeting task!'
    }
}
```

Gradle: Blocs de tâche

- doFirst : Spécifie des actions à exécuter avant que la tâche ne commence.
- doLast : Indique des actions à exécuter après que la tâche ait terminé.
- dependsOn : Définit les dépendances d'une tâche, garantissant qu'une ou plusieurs tâches soient exécutées avant celle-ci.
- finalizedBy : Spécifie une ou plusieurs tâches qui doivent être exécutées après la tâche courante, qu'elle réussisse ou échoue.
- onlyIf { condition } : Exécute une tâche seulement si la condition spécifiée est vraie.
- mustRunAfter : Indique que la tâche ne doit pas être exécutée avant les tâches spécifiées, contrôlant l'ordre d'exécution.

Gradle: Java Tasks

- compileJava : Compile les fichiers source Java.
- processResources : Copie les ressources dans le répertoire de build.
- test : Exécute les tests unitaires.
- jar : Crée un fichier JAR du projet compilé.
- build : Une tâche principale qui exécute plusieurs tâches, comme la compilation, les tests, et la génération d'artefacts.

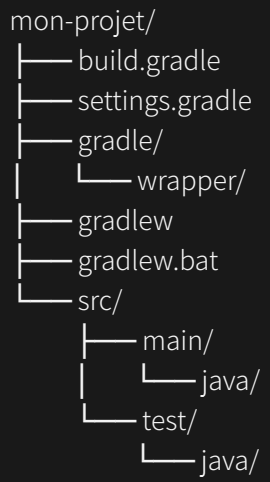
Gradle: Build types

Les types de build Gradle sont des configurations spécifiques qui définissent les tâches et les dépendances nécessaires pour construire différents types de projets.

Gradle: Exemple de build type

```
gradle init --type
 java-application
```

Gradle: Structure de projet java



Gradle: Wrapper

Le Gradle Wrapper est un script qui permet d'exécuter Gradle sans avoir à l'installer manuellement sur chaque machine, en garantissant que tous les membres de l'équipe utilisent la même version de Gradle pour le projet.

Gradle: settings.gradle

Le fichier settings.gradle dans Gradle est utilisé pour configurer les paramètres du projet, notamment pour définir les sous-projets dans un projet multi-module et pour spécifier des options de configuration globales.

Gradle: Commandes communes

- gradle build: Compile le code source, exécute les tests et génère les artefacts .
- gradle clean: Supprime le répertoire build/, nettoyant ainsi tous les fichiers générés par les précédents builds.
- gradle test: Exécute les tests unitaires définis dans le projet.
- gradle run: Exécute l'application Java (si le plugin application est utilisé et qu'une classe principale est spécifiée).
- gradle assemble: Génère les artefacts (comme des fichiers JAR) sans exécuter les tests.

Gradle: Plugins

Les plugins dans Gradle peuvent être utilisés pour ajouter des tâches, ou d’autres fonctionnalités à des tâches existantes.

Gradle: Exemple d’utilisation de plugins

```
plugins {
   id 'application'
}

application {
   mainClassName = 'com.example.App'
}
```

Gradle: Exemple de plugin

La commande gradle installDist permettra de créer une distribution pour votre application.

Celle-ci pourra être installée et exécutée sur n'importe quelle machine ayant une JVM à une version supportant votre application.

Gradle: Avantages et inconvénients

- Plus rapide
- Configuration flexible
- Compatible avec Maven
- Peut devenir complexe
- Moins structurant pour les débutants

Poetry

Poetry est un outil de gestion de dépendances et de packaging pour les projets Python.

Il vise à simplifier la gestion des bibliothèques Python, remplaçant des outils plus anciens comme pip et setuptools.

Poetry se distingue par son approche tout-en-un, gérant à la fois les dépendances, la publication de packages, et l'isolation des environnements.

Structure d’un projet Poetry

Un projet Poetry est structuré autour du fichier pyproject.toml, qui définit :

- Les dépendances à inclure dans le projet
- Les scripts ou commandes personnalisées pour différentes tâches
- Les informations sur le projet
- Les outils de gestion et de configuration, comme pytest pour les tests.

Poetry: Exemple d'un fichier pyproject.toml

```
[tool.poetry]
name = "mon-projet"
version = "0.1.0"
description = "Un projet Python utilisant
Poetry"
authors = ["Auteur <auteur@example.com>"]
license = "MIT"

[tool.poetry.dependencies]
python = "^3.8"
requests = "2.26.0"
flask = "^2.0.1"

[tool.poetry.dev-dependencies]
pytest = "^6.2.5"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Poetry: Gestion des dépendances

Les dépendances sont regroupées en 2 catégories distinctes:

- tool.poetry.dependencies:
  Définit les dépendances du projet comme la version de python ou les bibliothèques nécessaires.
- tool.poetry.dev-dependency:
  Définit les dépendances utilisées lors du développement.

Poetry: Gestion du référentiel

Par défaut, Poetry utilise PyPI (Python Package Index) comme dépôt.

Il est cependant possible de définir d'autres repository dans le fichier pyproject.toml.

Poetry: Exemple de gestion de référentiel

```
[[tool.poetry.source]]
name = "mon-depot-prive"
url =
"https://mon-depot-prive.com/simple"
default = true
```

Poetry: Phases de Gestion

Poetry ne suit pas un cycle de vie strict comme Maven. Plusieurs commandes existent cependant pour gérer les différentes étapes de développement:

- Installation des dépendances
- Exécution des scripts de tests
- Création d’un package
- Publier sur PyPI

Poetry: Installation des dépendances

La gestion des dépendances dans poetry peut passer par le fichier pyproject.toml et les commandes suivantes permettent de les gérer:

- poetry install: permet d'installer les dépendances
- poetry add numpy: permet d'ajouter une dépendance dans le projet.

Poetry: fichier poetry.lock

Lorsque l'on installera des dépendances, un fichier poetry.lock sera généré.

Dans ce fichier sera contenu des versions cohérentes de dépendances dans le projet.

Il sert également à éviter la résolution de la totalité des dépendances à chaque exécution de poetry install

Après l'ajout de nouvelles dépendances, il faudra utiliser la commande poetry update pour le mettre à jour.

Poetry: Exécution des scripts

L’exécution des tests ou des scripts doivent être appelés explicitement via les commandes suivantes:

- poetry run python my-app/main.py
- poetry run pytest

Poetry: Création d’un package.

La commande poetry build génèrera un package python.

Par défaut cette commande génère 2 fichiers:

- my-project-0.1.0.tar.gz: Le fichier de distribution source.
- my-project-0.1.0-py3-none-any.whl: Le fichier de distribution wheel.

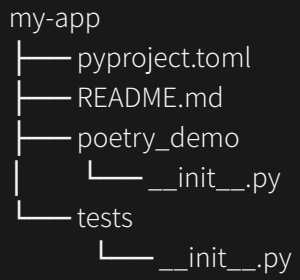
Poetry: Setup

Poetry ne propose pas d'archetype, cependant, il peut initialiser la structure de base d'un projet python.

Cela peut s'effectuer à l'aide de la commande suivante:

```
poetry new my-app
```

Poetry: Structure d’un projet



Poetry: Avantages et inconvénients

- Spécifiquement conçu et intégré avec Python
- Ecosystème plus jeune que Gradle
- Moins de fonctionnalités avancées

Containerisation

Containerisation: Pourquoi ?

La conteneurisation est une méthode de virtualisation légère qui permet de créer des environnements isolés pour exécuter des applications.

Elle permet de regrouper une application et ses dépendances dans un conteneur autonome, qui peut être exécuté de manière cohérente sur n'importe quel environnement.

Contrairement aux machines virtuelles, qui virtualisent le matériel, les conteneurs virtualisent le système d'exploitation, permettant ainsi une meilleure efficacité et une plus grande portabilité.

Containerisation: Avantages

- Portabilité: Un container comprend tout ce dont l'application a besoin pour fonctionner ce qui permet de déployer la même application partout.
- Isolation: Chaque conteneur est isolé des autres ainsi que de l'hôte.
- Légèreté: Contrairement aux machines virtuelles, les conteneurs partagent le noyau de l'hôte réduisant leur utilisations de ressources et leur temps de démarrage..

Docker

Docker est une plateforme open-source qui permet de créer, déployer et exécuter des applications dans des conteneurs.

Il est basé sur des technologies de conteneurisation de Linux (LXC), mais ajoute des fonctionnalités pratiques et une couche de gestion facilitée.

Docker: Composants

- Daemon Docker
- CLI Docker
- Dockerfile
- Image Docker
- Conteneur Docker
- Volume Docker
- Registry

Docker: Daemon

C'est le processus principal qui s'exécute en arrière-plan sur l'hôte, responsable de la gestion des conteneurs, des images, des réseaux, et des volumes.

Il interagit directement avec le système d'exploitation pour gérer les conteneurs.

Docker: CLI

L'interface en ligne de commande Docker permet aux utilisateurs d'interagir avec le Docker Daemon.

Elle envoie des commandes au démon via l'API Docker. Cette API est en REST.

Docker: Dockerfile

Un Dockerfile est un fichier texte contenant une série d'instructions permettant de construire automatiquement une image Docker, incluant toutes les dépendances, configurations et étapes nécessaires pour exécuter une application dans un conteneur.

Docker: Exemple de Dockerfile

```
FROM node:14

WORKDIR /app

COPY . ./

RUN npm install

EXPOSE 3000

CMD ["npm", "start"]
```

Docker: Image

Ce sont des instantanés en lecture seule contenant tout ce qu'il faut pour exécuter une application (code, bibliothèques, dépendances).

Les images sont téléchargées ou construites localement sur l'hôte et stockées dans un registre.

Docker: Créer une image

A partir d'un Dockerfile il sera nécessaire de construire une image pour pouvoir créer un conteneur.

```
docker build -t nom_de_l_image .
```

Cette image peut être ensuite partagée via diverses méthodes.

Docker: Conteneur

Un conteneur est une instance d'une image.

Lorsque vous exécutez une image, vous obtenez un conteneur, qui est comme un processus isolé dans son propre environnement.

Ils sont isolés les uns des autres et partagent le noyau de l'hôte, ce qui les rend légers et efficaces.

Chaque conteneur a son propre espace de noms pour les processus, les réseaux et les systèmes de fichiers.

Docker: Démarrer un container

Pour démarrer un container docker, il faut utiliser la commande suivante à l'aide du CLI:

```
docker run -d -p 8080:80 nom_de_l_image
```

Docker: Volume

Ce sont des mécanismes de stockage permettant aux conteneurs de persister des données indépendamment de leur cycle de vie, en stockant les données sur un système de fichier..

Docker: Exemple de volume

Pour indiquer un volume ou un réseau dans Docker, il y a deux méthodes principales : via la ligne de commande (CLI) ou via un Dockerfile.

Il faut noter que lorsque cela est indiqué dans l'image ce n'est qu'à titre informatif, c'est le container qui indiquera où se trouvera le volume.

Docker: Exemple de volume

En ligne de commande:

```
docker run -d -v /opt/my-app:/application nom_de_l_image
```

```
docker volume create mon_volume
```

```
docker run -d -v mon_volume:/application nom_de_l_image
```

Dans un Dockerfile:

```
VOLUME /application
```

Docker: Réseau

Les réseaux permettent de gérer la communication entre les conteneurs et d’isoler des groupes de conteneurs au sein d'un même réseau.

Lorsque l’on indique le réseau, il faut obligatoirement qu’il existe au préalable.

Docker: Créer un réseau

```
docker network create mon_reseau

docker run -d --network mon_reseau nom_de_l_image
```

RUTH

Naomi Loses Her Husband and Sons

1 In the days when the judges
ruled,[a2] there was a famine in the
land.[b] So a man from Bethlehem in
Judah, together with his wife and two
sons, went to live for a while in the
country of Moab.[c] 2The man's name
was Elimelek, his wife's name was Na-
omi, and the names of his two sons
were Mahlon and Kilion. They were
Ephrathites from Bethlehem,[d] Ju-
dah. And they went to Moab and lived
there.

3Now Elimelek, Naomi's husband,
died, and she was left with her two
sons. 4They married Moabite wom-
en, one named Orpah and the oth-
er Ruth.[e] After they had lived there
about ten years, 5both Mahlon and
Kilion also died, and Naomi was left
without her two sons and her hus-
band.

Naomi and Ruth Return to Bethlehem

6When Naomi heard in Moab that
the LORD had come to the aid of his
people[f] by providing food[g] for them,
she and her daughters-in-law pre-
pared to return home from there.
7With her two daughters-in-law she
left the place where she had been liv-
ing and set out on the road that would
take them back to the land of Judah.

8Then Naomi said to her two
daughters-in-law, "Go back, each of
you, to your mother's home. May
the LORD show you kindness,[h] as you
have shown kindness to your dead
husbands[i] and to me. 9May the LORD
grant that each of you will find rest[j] in
the home of another husband."

Then she kissed them goodbye and
they wept aloud 10and said to her, "We
will go back with you to your people."

11But Naomi said, "Return home,
my daughters. Why would you come
with me? Am I going to have any
more sons, who could become your
husbands?[k] 12Return home, my
daughters; I am too old to have an-
other husband. Even if I thought
there was still hope for me — even if I
had a husband tonight and then gave
birth to sons — 13would you wait until
they grew up? Would you remain un-
married for them? No, my daughters.
It is more bitter for me than for you,
because the LORD's hand has turned
against me!"

14At this they wept aloud again.
Then Orpah kissed her mother-in-
law[m] goodbye, but Ruth clung to her.[l]

15"Look," said Naomi, "your sister-
in-law is going back to her people and
her gods.[n] Go back with her."

16But Ruth replied, "Don't urge me
to leave you[o] or to turn back from you.
Where you go I will go, and where you
stay I will stay. Your people will be
my people and your God my God.[p]
17Where you die I will die, and there
I will be buried. May the LORD deal
with me, be it ever so severely,[q] if even
death separates you and me." 18When
Naomi realized that Ruth was deter-
mined to go with her, she stopped
urging her.[r]

19So the two women went on until
they came to Bethlehem. When they
arrived in Bethlehem, the whole town
was stirred[s] because of them, and the
women exclaimed, "Can this be Nao-
mi?"

20"Don't call me Naomi,[t]" she told
them. "Call me Mara,[u] because the Al-
mighty[v][w] has made my life very bit-
ter.[v] 21I went away full, but the LORD
has brought me back empty.[w] Why call
me Naomi? The LORD has afflicted[x]
me; the Almighty has brought mis-
fortune upon me."

22So Naomi returned from Moab
accompanied by Ruth the Moabite,
her daughter-in-law, arriving in Beth-
lehem as the barley harvest[y] was be-
ginning.[z]

[a] 1 Traditionally *judged* [b] *20 Naomi* means *pleasant.* [c] *20 Mara* means *bitter.*
[d] *20 Hebrew Shaddai*; also in verse 21
[e] *21 Or has testified against*

Ruth Meets Boaz in the Grain Field

2 Now Naomi had a relative[z] on
husband's side, a man of sta
ing from the clan of Elimelek,[a] wh
name was Boaz.[b]

2And Ruth the Moabite said to
omi, "Let me go to the fields and pi
up the leftover grain[c] behind anyo
in whose eyes I find favor."

Naomi said to her, "Go ahead, m
daughter." 3So she went out, entere
a field and began to glean behind th
harvesters. As it turned out, she wa
working in a field belonging to Boaz,
who was from the clan of Elimelek.

4Just then Boaz arrived from Beth-
lehem and greeted the harvesters,
"The LORD be with you!"[d]

"The LORD bless you!"[e]" they an-
swered.

5Boaz asked the overseer of his har-
vesters, "Who does that young woman
belong to?"

6The overseer replied, "She is the
Moabite[f] who came back from Moab
with Naomi. 7She said, 'Please let me
glean and gather among the sheaves
behind the harvesters.' She came into
the field and has remained here from
morning till now, except for a short
rest in the shelter."

8So Boaz said to Ruth, "My daugh-
ter, listen to me. Don't go and glean in
another field and don't go away from
here. Stay here with the women who
work for me. 9Watch the field where
the men are harvesting, and follow
along after the women. I have told
the men not to lay a hand on you. And
whenever you are thirsty, go and get
a drink from the water jars the men
have filled."

10At this, she bowed down with
her face to the ground.[g] She asked
him, "Why have I found such favor
in your eyes that you notice me[h] — a
foreigner?"[i]

11Boaz replied, "I've been told all
about what you have done for your
mother-in-law[j] since the death of
your husband — how you left your fa-
ther and mother and your homeland
and came to live with a people you did
not know before.[k] 12May the LORD re-
pay you for what you have done. May
you be richly rewarded by the LORD,[l]
the God of Israel, under whose wings[m]
you have come to take refuge."

[z] 1 Hebrew *Shaddai*; also in verse 21
[a] *1 Or has testified against*
[b] *1 Traditionally* [c] *2 Hebrew* [d] *4 Or* [e] *4 Or* [f] *6 Or* [g] *10 Or* [h] *10 Or* [i] *10 Or* [j] *11 Or* [k] *11 Or* [l] *12 Or*

Semantic Versioning

Semantic Versioning

Le Semantic Versioning (SemVer) est une **convention** de **gestion** de **version** de logiciels qui permet de **communiquer** clairement les **changements** apportés à une application ou une bibliothèque.

Il est particulièrement utile pour les développeurs et les utilisateurs de comprendre la **compatibilité** entre différentes **versions** d'un logiciel.

Semver fonctionnement

Le SemVer utilise un format de version composé de trois segments principaux : MAJOR.MINOR.PATCH, par exemple 1.0.0.

- **MAJOR** (Majeure) : Ce segment est **incrémenté** lorsque des modifications **incompatibles** avec les versions précédentes sont introduites.
- **MINOR** (Mineure) : Ce segment est **incrémenté** lorsque de **nouvelles** fonctionnalités sont ajoutées de manière **rétrocompatible**.
- **PATCH** (Correctif) : Ce segment est incrémenté lorsque des **corrections** de **bugs** rétrocompatibles sont apportées.

Semver: Fonctionnalités supplémentaires

- **Pré-versions** : Utilisées pour les **versions** de **développement**, les bêtas, les alphas, etc.
  Par exemple, 1.0.0-alpha, 1.0.0-beta.1.
- **Métadonnées** : Informations **supplémentaires** qui ne **modifient pas** la version **principale**.
  Par exemple, 1.0.0+build20231010.

Semantic Versioning: Règles additionnelles

- **Zéro initial** : La version **0.y.z** est **utilisée** pour le développement **initial**. Tout peut **changer**. La version **publique** commence à **1.0.0**.
- **Pré-versions** : Les pré-versions ont une **priorité inférieure** à la **version** associée. Par exemple, 1.0.0-alpha < 1.0.0.

Gestion des sources Git et Git Flow

Rappels Git

La gestion des sources est une pratique essentielle dans le développement logiciel pour suivre les modifications apportées au code, collaborer efficacement avec d'autres développeurs, et assurer la qualité et la stabilité du produit final. Git est l'un des systèmes de gestion de versions les plus populaires, et Git Flow est une méthodologie de travail basée sur Git qui facilite la gestion des branches et des versions.

Git composants

1. Dépôt (Repository)

2. Commit

3. Branche (Branch)

4. Fusion (Merge)

5. Conflit (Conflict)

Git: Repository

Un dépôt est une base de données qui contient toutes les versions d'un projet. Il peut être local (sur votre machine) ou distant (sur un serveur).

Git: Commit

Un commit est une sauvegarde d'un ensemble de modifications apportées aux fichiers. Chaque commit a un identifiant unique et un message décrivant les modifications.

Git: Branche

Une branche est une ligne de développement indépendante. Les branches permettent de travailler sur différentes fonctionnalités ou corrections de bugs en parallèle sans interférer les uns avec les autres.

Git: Fusion

La fusion est le processus de combiner les modifications de différentes branches. Cela permet d'intégrer les nouvelles fonctionnalités ou corrections dans la branche principale.

Git: Conflit

Un conflit survient lorsque deux modifications incompatibles sont apportées au même fichier. Git ne peut pas fusionner automatiquement ces modifications, et une intervention manuelle est nécessaire pour résoudre le conflit.

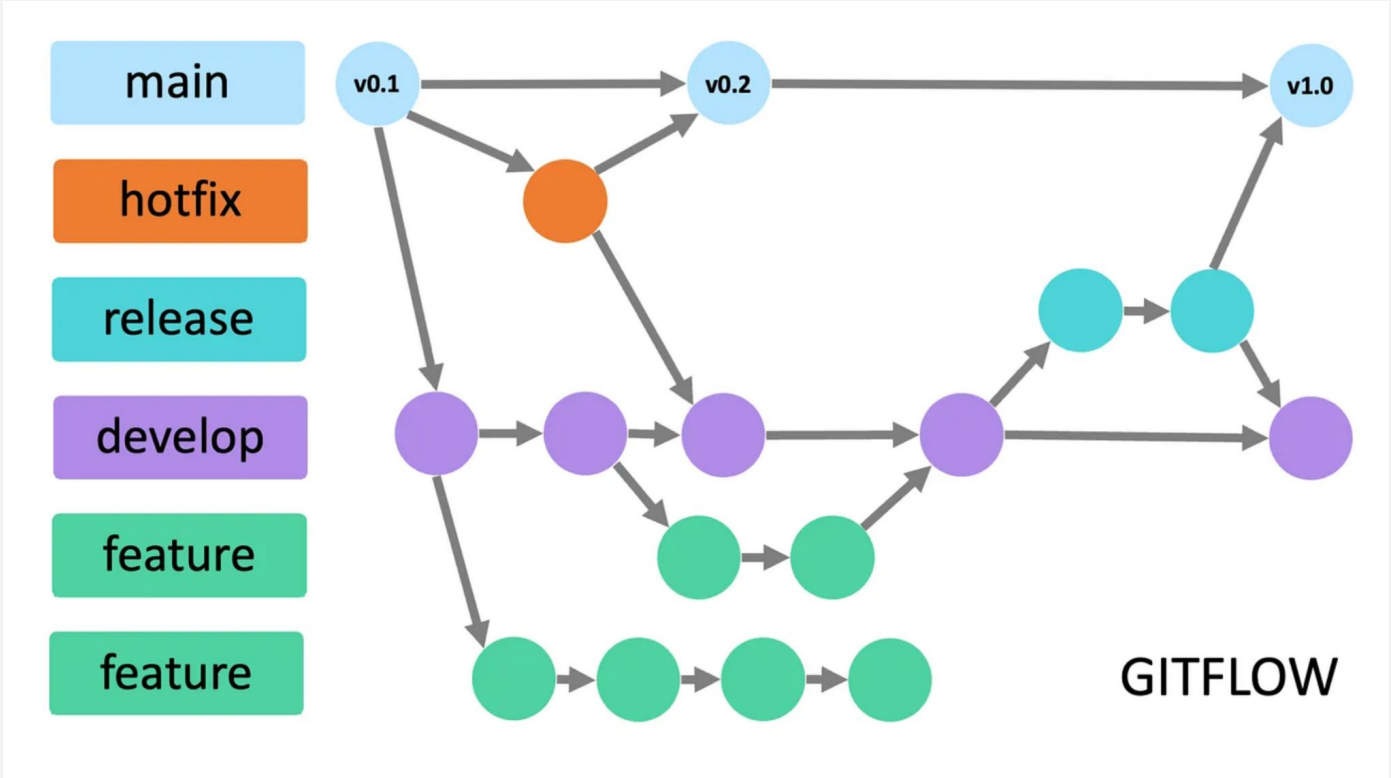
Git: Commandes

```
git init
git clone <URL>
git add <fichier>
git commit -m "message"
git push
git pull
git branch
git checkout <branche>
git merge <branche>
```

Git Flow

Git Flow est une méthodologie de travail basée sur Git qui définit un ensemble de règles et de conventions pour gérer les branches et les versions d'un projet.

Git Flow


main
hotfix
release
develop
feature
feature
v0.1
v0.2
v1.0
GITFLOW

Git Flow: Branches Principales

main (ou master) : La branche principale qui contient le code de production.

develop : La branche de développement qui contient le code de la prochaine version.

Git Flow: Branches de Support

feature : Branches utilisées pour développer de nouvelles fonctionnalités. Elles sont créées à partir de develop et fusionnées dans develop une fois la fonctionnalité terminée.

release : Branches utilisées pour préparer une nouvelle version. Elles sont créées à partir de develop et fusionnées dans main et develop une fois la version prête.

hotfix : Branches utilisées pour corriger des bugs critiques en production. Elles sont créées à partir de main et fusionnées dans main et develop une fois le bug corrigé.

Documentation

Documentation

1. Compréhension et Maintenance

La documentation aide les développeurs à comprendre rapidement le code, même s'ils ne l'ont pas écrit eux-mêmes. Cela facilite la maintenance et l'ajout de nouvelles fonctionnalités.

2. Collaboration

Dans les projets collaboratifs, la documentation permet aux membres de l'équipe de travailler ensemble de manière plus efficace. Elle réduit les malentendus et les erreurs de communication.

3. Réutilisabilité

Un code bien documenté est plus facile à réutiliser. Les développeurs peuvent comprendre rapidement comment utiliser une fonction ou une classe, ce qui accélère le développement.

Documentation: Docstring

```
def add(a, b):
    """
    Add two numbers and return the result.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The sum of a and b.
    """
    return a + b
```

Documentation: Commentaire

```
# Calculate the factorial of a number
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Documentation: pydoc

pydoc est un outil intégré à Python qui permet de générer et d'afficher la documentation à partir des docstrings présentes dans le code source.

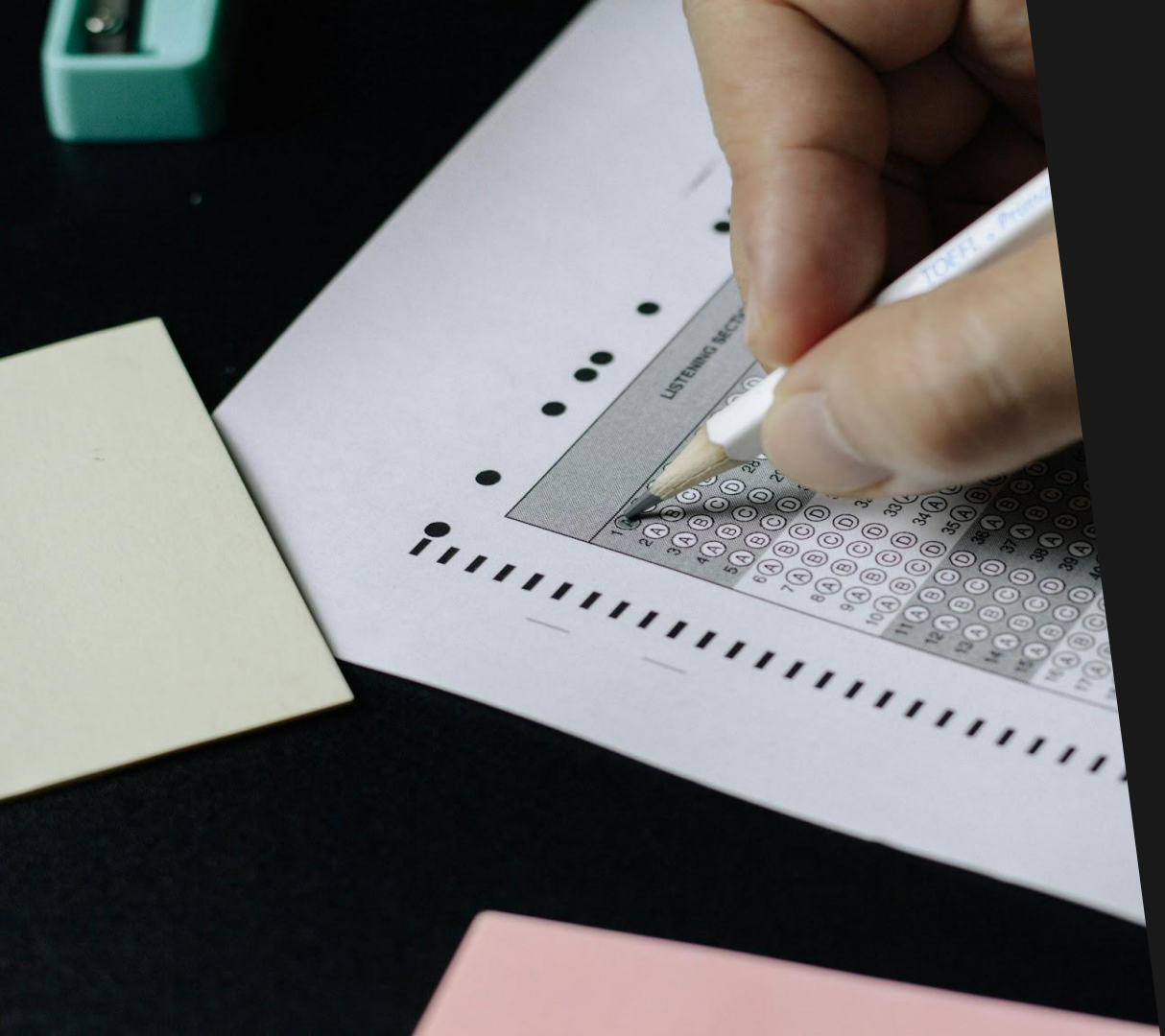
Il est particulièrement utile pour les développeurs qui souhaitent explorer rapidement la documentation des modules, classes, fonctions et méthodes sans avoir à quitter l'environnement de développement.

Documentation: Commandes pydoc

pydoc math

pydoc -p 8080

pydoc -w math



Test unitaires

Avantages des tests unitaires

1. Détection Précoce des Bugs
2. Facilitation des Refactorisations
3. Documentation Vivante
4. Amélioration de la Conception
5. Réduction des Risques

Tests unitaires: pytest

```
import pytest
from calculator import add, subtract

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(-1, -1) == -2

def test_subtract():
    assert subtract(5, 3) == 2
    assert subtract(5, 5) == 0
    assert subtract(-1, -1) == 0
```

Pytest: fixtures

```
import pytest
from calculator import add, subtract

@pytest.fixture
def sample_data():
    return {
        'add': [(2, 3, 5), (-1, 1, 0), (-1, -1, -2)],
        'subtract': [(5, 3, 2), (5, 5, 0), (-1, -1, 0)]
    }

def test_add(sample_data):
    for a, b, expected in sample_data['add']:
        assert add(a, b) == expected

def test_subtract(sample_data):
    for a, b, expected in sample_data['subtract']:
        assert subtract(a, b) == expected
```

Plan du cours

- Introduction aux Concepts de Base de l'Organisation des Entreprises
- Cycle de vie et cycle de développement logiciel
- Activités de développement logiciel
- Avant-projet
- Collecte des besoins : cas d'utilisation, user's story
- Analyse Conception
- Codage
- **Intégration**
- Déploiement et production : plateforme de déploiement continu, surveillance des applications, conteneurisation en production (docker)
- Activités de gestion de projets

Intégration continue et Tests d'Intégration

Définition

L'intégration continue est une pratique de développement logiciel où les développeurs intègrent fréquemment leur code dans un dépôt partagé.

Chaque intégration est vérifiée par une série de tests automatisés pour détecter les erreurs rapidement.

Intégration continue: Avantages

Détection Précoce des Bugs : Les erreurs sont détectées rapidement, ce qui réduit les coûts de correction.

Amélioration de la Qualité : Les tests automatisés garantissent que le code fonctionne correctement.

Réduction des Risques : Les modifications sont testées immédiatement, ce qui réduit les risques de régressions.

Automatisation : Les tests et les déploiements peuvent être automatisés, ce qui accélère le cycle de développement.

Collaboration : Les développeurs peuvent travailler ensemble de manière plus efficace, car les conflits de code sont détectés rapidement.

Tests d'Intégration

Les tests d'intégration vérifient que les différentes parties d'un système fonctionnent correctement ensemble.

Contrairement aux tests unitaires, qui se concentrent sur des unités de code isolées, les tests d'intégration vérifient les interactions entre les composants.

Test d’intégration: Avantages

Détection des Problèmes d'Intégration : Les tests d'intégration permettent de détecter les problèmes liés à l'interaction entre les composants.

Validation des Interfaces : Ils vérifient que les interfaces entre les composants sont correctement implémentées.

Amélioration de la Qualité : Les tests d'intégration garantissent que le système fonctionne correctement dans son ensemble.

Réduction des Risques : Les modifications apportées à un composant sont testées pour s'assurer qu'elles n'affectent pas les autres composants.

Intégration continue: Gitlab CI

GitLab CI/CD est une solution d'intégration continue et de déploiement continu intégrée à GitLab.

Elle permet d'automatiser les processus de développement, de test et de déploiement, ce qui améliore la qualité et la rapidité du développement logiciel

Gitlab CI: Composants

- Pipelines
- Jobs
- Stages
- Runners
- Artifacts
- Caching
- Environnements
- Schedules

Gitlab CI: Pipelines

Un pipeline est une série d'étapes (ou jobs) qui s'exécutent dans un ordre défini.

Les pipelines peuvent inclure des étapes de construction, de test, de déploiement, etc.

Gitlab CI: Jobs

Un job est une unité de travail dans un pipeline.

Chaque job peut être configuré pour exécuter des scripts, utiliser des environnements spécifiques, et dépendre d'autres jobs.

Gitlab CI: Stages

Les stages sont des groupes de jobs qui s'exécutent séquentiellement.

Par exemple, vous pouvez avoir des stages pour la construction, les tests, et le déploiement.

Gitlab CI: Runners

Les runners sont des agents qui exécutent les jobs définis dans les pipelines.

GitLab propose des runners partagés, mais vous pouvez également configurer vos propres runners pour exécuter les jobs sur vos propres machines.

Gitlab CI: Artifacts

Les artifacts sont des fichiers générés par les jobs et stockés par GitLab.

Ils peuvent être utilisés par des jobs ultérieurs dans le pipeline.

Gitlab CI: Caching

Le caching permet de stocker des fichiers entre les exécutions de jobs pour améliorer les performances.

Par exemple, vous pouvez mettre en cache les dépendances pour éviter de les télécharger à chaque exécution.

Gitlab CI: Environnements

Les environnements permettent de définir des environnements de déploiement (comme staging, production, etc.) et de suivre les déploiements dans ces environnements.

Gitlab CI: Schedules

Les schedules permettent de planifier l'exécution des pipelines à des moments spécifiques.

.gitlab-ci.yml

```
stages:
 - test
 - build
 - deploy

variables:
 PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
 paths:
  - .cache/pip

before_script:
 - python -m venv venv
 - source venv/bin/activate
 - pip install -r requirements.txt

unit_tests:
 stage: test
 script:
  - pytest tests/test_calculator.py
```

```
integration_tests:
 stage: test
 script:
  - python app.py &
  - sleep 5  # Attendre que l'application démarre
  - pytest tests/test_app.py

build_docs:
 stage: build
 script:
  - cd docs
  - make html
 artifacts:
  paths:
   - docs/build/html/
  expire_in: 1 week

deploy:
 stage: deploy
 script:
  - echo "Deploying to production..."
 only:
  - main
```

Tests d'Intégration

Les tests d'intégration vérifient que les différentes parties d'un système fonctionnent correctement ensemble.

Contrairement aux tests unitaires, qui se concentrent sur des unités de code isolées, les tests d'intégration vérifient les interactions entre les composants.

Test d’intégration: Avantages

Détection des Problèmes d'Intégration : Les tests d'intégration permettent de détecter les problèmes liés à l'interaction entre les composants.

Validation des Interfaces : Ils vérifient que les interfaces entre les composants sont correctement implémentées.

Amélioration de la Qualité : Les tests d'intégration garantissent que le système fonctionne correctement dans son ensemble.

Réduction des Risques : Les modifications apportées à un composant sont testées pour s'assurer qu'elles n'affectent pas les autres composants.

Plan du cours

- Introduction aux Concepts de Base de l'Organisation des Entreprises
- Cycle de vie et cycle de développement logiciel
- Activités de développement logiciel
- Avant-projet
- Collecte des besoins : cas d'utilisation, user's story
- Analyse Conception
- Codage
- Intégration
- **Déploiement et production : plateforme de déploiement continu, surveillance des applications, conteneurisation en production (docker)**
- Activités de gestion de projets

Déploiement Continu

Déploiement continu

Le déploiement est le processus de mise en production d'une application ou d'un service.

Le déploiement continu (CD) est une pratique de développement logiciel où les modifications de code sont automatiquement déployées en production après avoir passé les tests automatisés.

Cela permet de livrer des fonctionnalités plus rapidement et de manière plus fiable.

Surveillance des applications

La surveillance des applications est le processus de suivre et d'analyser les performances, la disponibilité, et les erreurs d'une application en production.

La surveillance permet de détecter les problèmes rapidement et de prendre des mesures correctives pour garantir la stabilité et la performance de l'application.

Outils utilisés dans le déploiement et la surveillance

Les outils utilisés dans le déploiement et la surveillance incluent Git pour la gestion de version, Docker pour la conteneurisation, GitLab CI pour l'automatisation des pipelines de déploiement continu, et Prometheus pour la surveillance des métriques et des alertes en temps réel.

Ces outils permettent de garantir la portabilité, la reproductibilité, et la stabilité des applications en production.

Plan du cours

- Introduction aux Concepts de Base de l'Organisation des Entreprises
- Cycle de vie et cycle de développement logiciel
- Activités de développement logiciel
- Avant-projet
- Collecte des besoins : cas d'utilisation, user's story
- Analyse Conception
- Codage
- Intégration
- Déploiement et production : plateforme de déploiement continu, surveillance des applications, conteneurisation en production (docker)
- **Activités de gestion de projets**

Gestion de projet

Outils de gestion de projet

Les outils de gestion de projet sont des logiciels et des applications qui aident à planifier, organiser, et suivre les activités de gestion de projet.

Ils facilitent la collaboration, la communication, et la gestion des ressources.

Indicateurs de performance

Les indicateurs de performance (KPI) sont des mesures quantifiables qui permettent d'évaluer l'efficacité et l'efficience des activités de gestion de projet.

Ils aident à suivre les progrès, à identifier les écarts, et à prendre des décisions informées.

Exemples de KPI

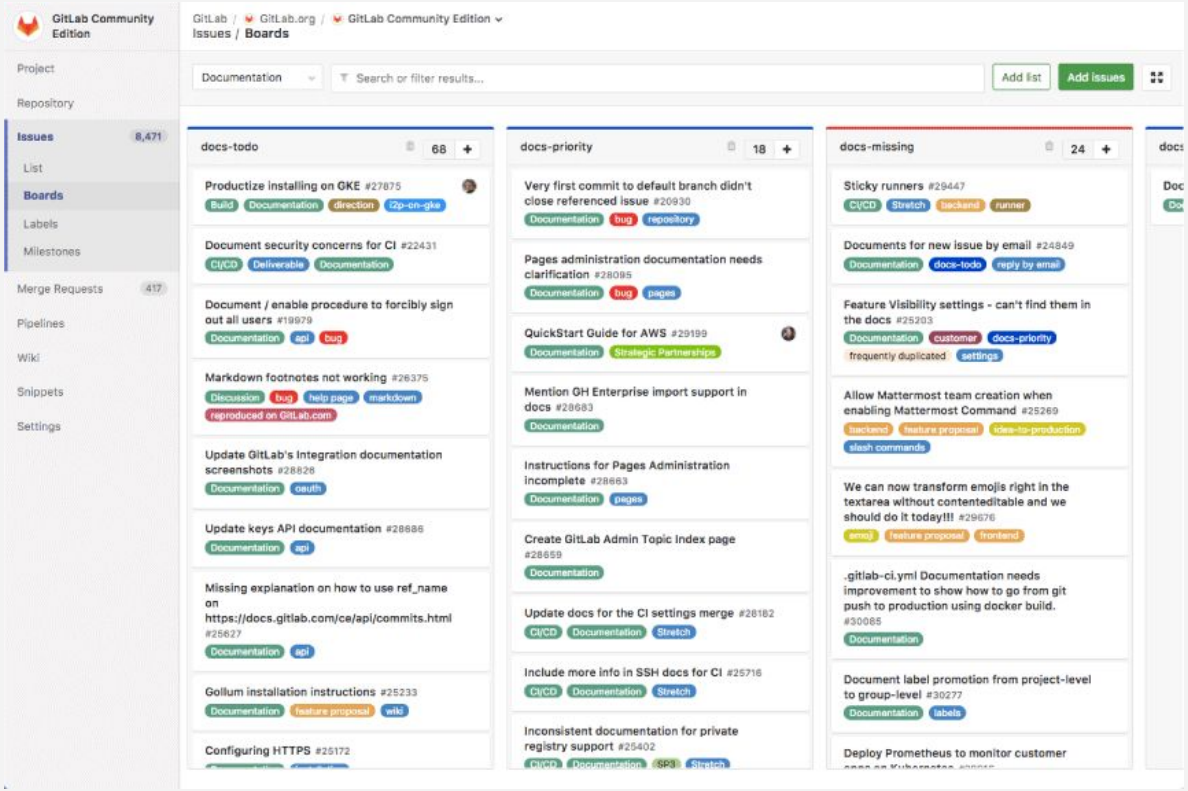
- Délais : Mesurer si le projet respecte les échéances prévues.
- Coûts : Suivre les dépenses par rapport au budget alloué.
- Qualité : Évaluer la conformité des livrables aux spécifications et aux attentes.
- Ressources : Surveiller l'utilisation des ressources humaines et matérielles.
- Satisfaction des Parties Prenantes : Mesurer le niveau de satisfaction des parties prenantes par rapport aux attentes et aux objectifs du projet.

Exemple d’outil de gestion de projet: Gitlab CI

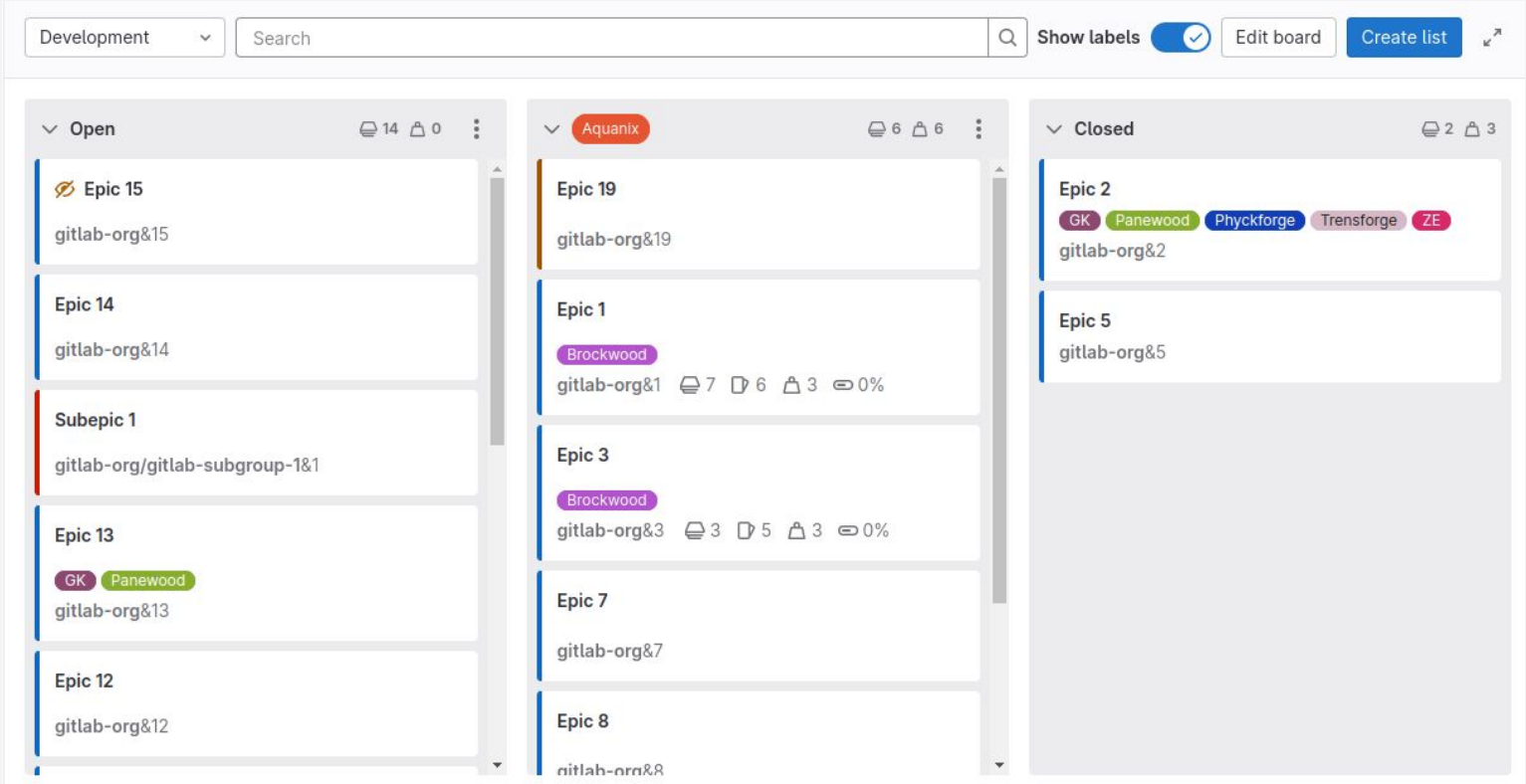
- Issues
- Epics
- Roadmap (Feuille de route)
- Milestones (Jalons)
- Time Tracking

https://about.gitlab.com/features/

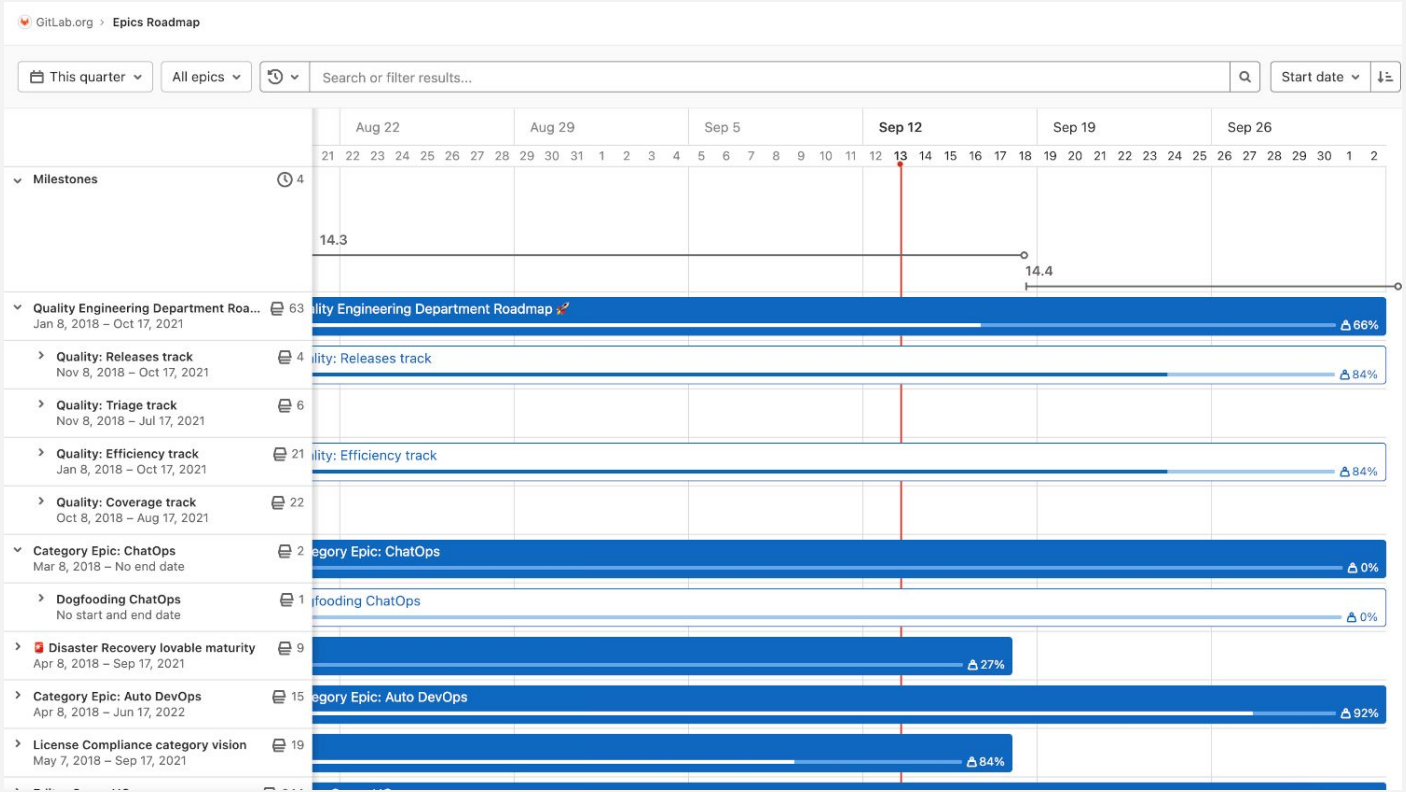
Issues


GitLab Community Edition
GitLab / GitLab.org / GitLab Community Edition
Issues / Boards
Project
Repository
Issues 8,471
List
Boards
Labels
Milestones
Merge Requests 417
Pipelines
Wiki
Snippets
Settings
Documentation
Search or filter results...
Add list
Add issues
docs-todo 68
Productize installing on GKE #27875
Build Documentation direction i2p-on-gke
Document security concerns for CI #22431
CI/CD Deliverable Documentation
Document / enable procedure to forcibly sign out all users #19979
Documentation api bug
Markdown footnotes not working #26375
Discussion bug help page markdown reproduced on GitLab.com
Update GitLab's Integration documentation screenshots #28828
Documentation oauth
Update keys API documentation #28686
Documentation api
Missing explanation on how to use ref_name on https://docs.gitlab.com/ce/api/commits.html #25627
Documentation api
Gollum installation instructions #25233
Documentation feature proposal wiki
Configuring HTTPS #25172
docs-priority 18
Very first commit to default branch didn't close referenced issue #20930
Documentation bug repository
Pages administration documentation needs clarification #28085
Documentation bug pages
QuickStart Guide for AWS #29199
Documentation Strategic Partnerships
Mention GH Enterprise import support in docs #28683
Documentation
Instructions for Pages Administration incomplete #28663
Documentation pages
Create GitLab Admin Topic Index page #28659
Documentation
Update docs for the CI settings merge #28182
CI/CD Documentation Stretch
Include more info in SSH docs for CI #25716
CI/CD Documentation Stretch
Inconsistent documentation for private registry support #25402
CI/CD Documentation SP3 Stretch
docs-missing 24
Sticky runners #29447
CI/CD Stretch backend runner
Documents for new issue by email #24849
Documentation docs-todo reply by email
Feature Visibility settings - can't find them in the docs #25203
Documentation customer docs-priority frequently duplicated settings
Allow Mattermost team creation when enabling Mattermost Command #25269
backend feature proposal idea-to-production slash commands
We can now transform emojis right in the textarea without contenteditable and we should do it today!!! #29676
emoji feature proposal frontend
.gitlab-ci.yml Documentation needs improvement to show how to go from git push to production using docker build. #30085
Documentation
Document label promotion from project-level to group-level #30277
Documentation labels
Deploy Prometheus to monitor customer

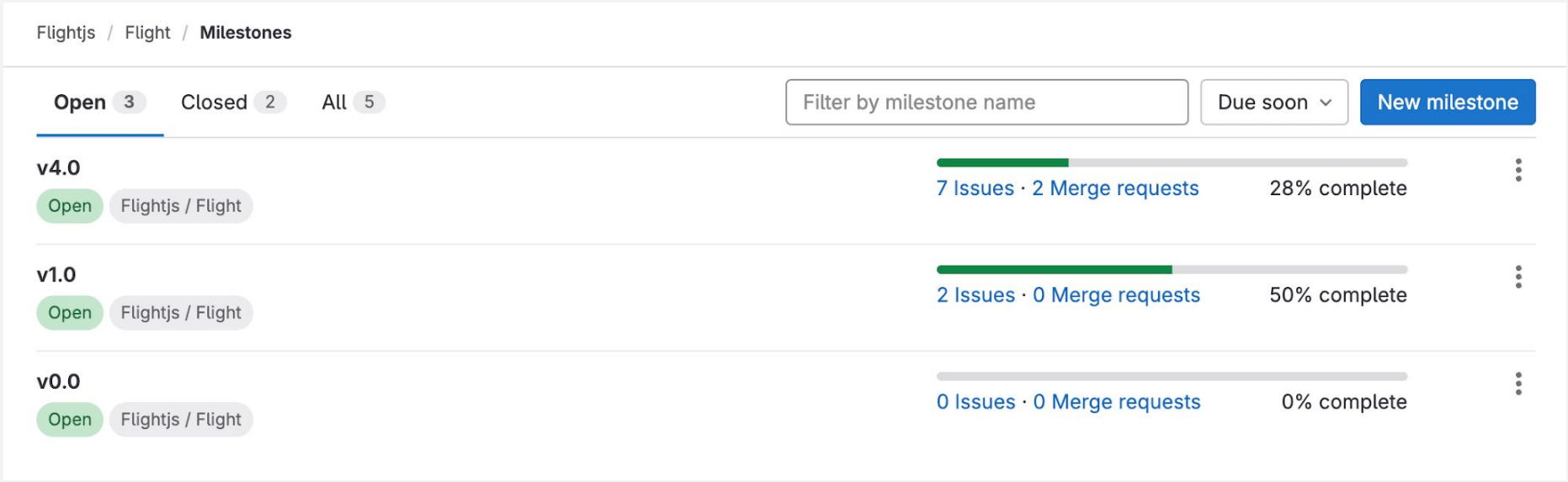
Epics


Development
Search
Show labels
Edit board
Create list
Open
14 0
Epic 15
gitlab-org&15
Epic 14
gitlab-org&14
Subepic 1
gitlab-org/gitlab-subgroup-1&1
Epic 13
GK
Panewood
gitlab-org&13
Epic 12
gitlab-org&12
Aquanix
6 6
Epic 19
gitlab-org&19
Epic 1
Brockwood
gitlab-org&1 7 6 3 0%
Epic 3
Brockwood
gitlab-org&3 3 5 3 0%
Epic 7
gitlab-org&7
Epic 8
Closed
2 3
Epic 2
GK
Panewood
Phyckforge
Trensforge
ZE
gitlab-org&2
Epic 5
gitlab-org&5

Roadmap


GitLab.org › Epics Roadmap
This quarter
All epics
Search or filter results...
Start date
Aug 22
Aug 29
Sep 5
Sep 12
Sep 19
Sep 26
Milestones 4
14.3
14.4
Quality Engineering Department Roa... 63
Jan 8, 2018 – Oct 17, 2021
lity Engineering Department Roadmap
66%
Quality: Releases track 4
Nov 8, 2018 – Oct 17, 2021
lity: Releases track
84%
Quality: Triage track 6
Nov 8, 2018 – Jul 17, 2021
Quality: Efficiency track 21
Jan 8, 2018 – Oct 17, 2021
lity: Efficiency track
84%
Quality: Coverage track 22
Oct 8, 2018 – Aug 17, 2021
Category Epic: ChatOps 2
Mar 8, 2018 – No end date
egory Epic: ChatOps
0%
Dogfooding ChatOps 1
No start and end date
fooding ChatOps
0%
Disaster Recovery lovable maturity 9
Apr 8, 2018 – Sep 17, 2021
27%
Category Epic: Auto DevOps 15
Apr 8, 2018 – Jun 17, 2022
egory Epic: Auto DevOps
92%
License Compliance category vision 19
May 7, 2018 – Sep 17, 2021
84%

Milestones


Flightjs / Flight / Milestones
Open 3
Closed 2
All 5
Filter by milestone name
Due soon
New milestone
v4.0
Open
Flightjs / Flight
7 Issues · 2 Merge requests
28% complete
v1.0
Open
Flightjs / Flight
2 Issues · 0 Merge requests
50% complete
v0.0
Open
Flightjs / Flight
0 Issues · 0 Merge requests
0% complete

Time Tracking