



# Formation Docker

Ali Koudri

[ali.koudri@gmail.com](mailto:ali.koudri@gmail.com)



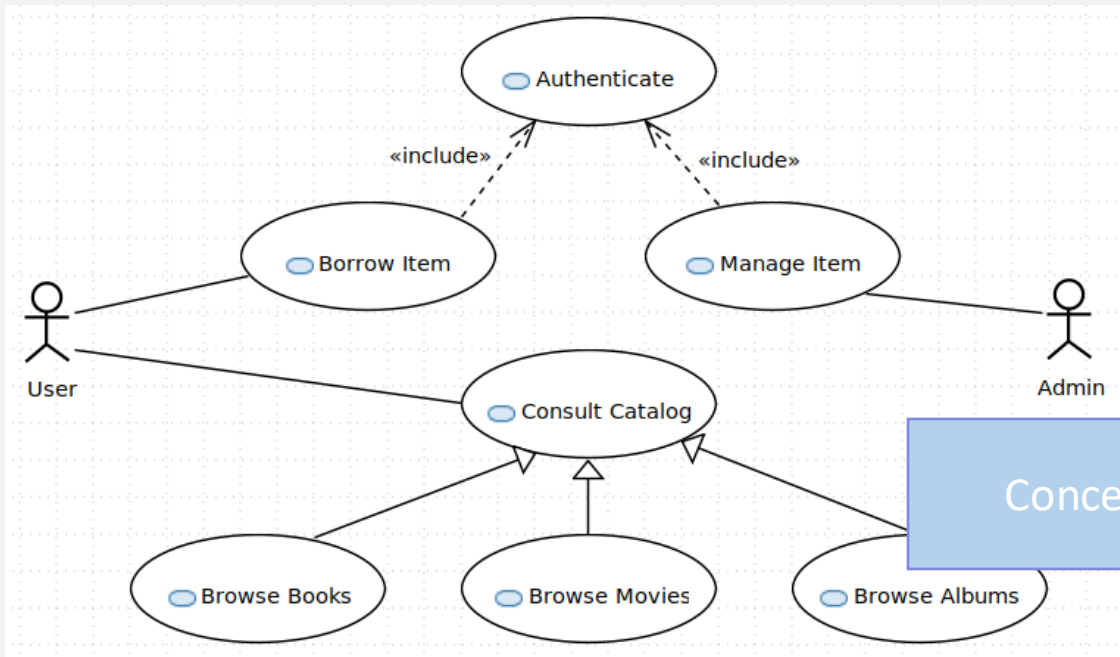
# Plan de la formation

- Introduction
- Présentation Générale
- Gestion des images (part 1)
- Gestion des conteneurs
- Gestion des images (part 2)
- Gestion des réseaux
- Gestion des volumes
- Composition de services
- Gestion de la sécurité
- Déploiement sur AWS
- Considérations transverses
- Conclusion

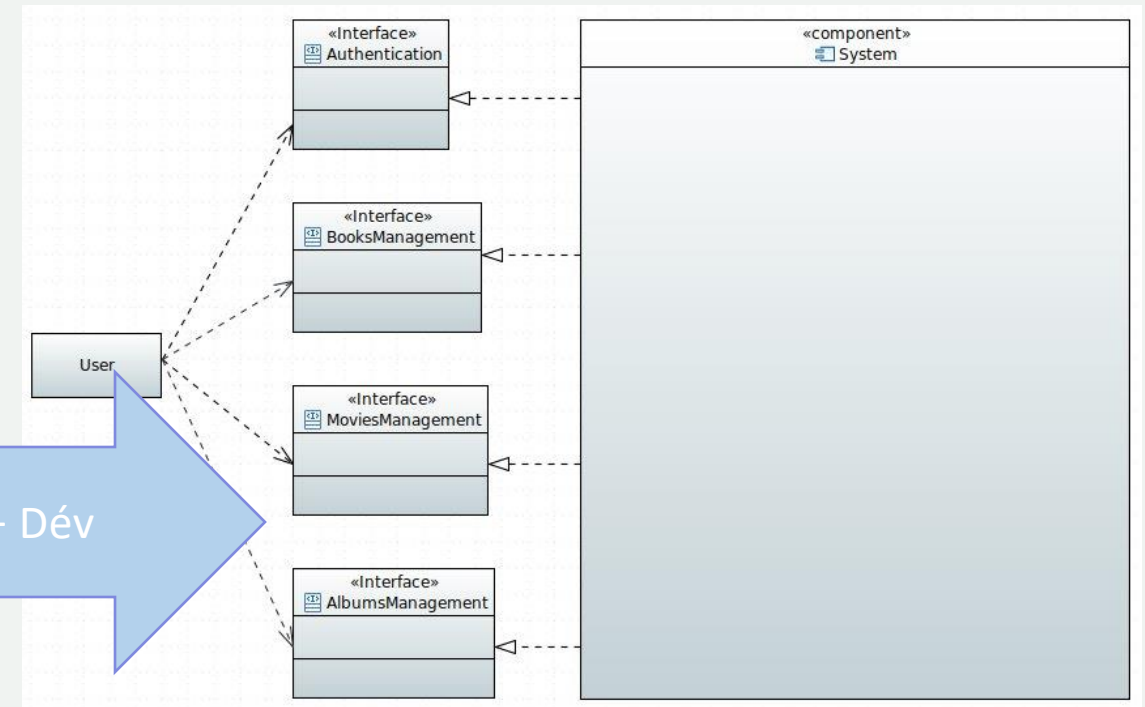
# Introduction

# Exemple d'application

- Application de gestion de contenus multimédia



Conception + Dév

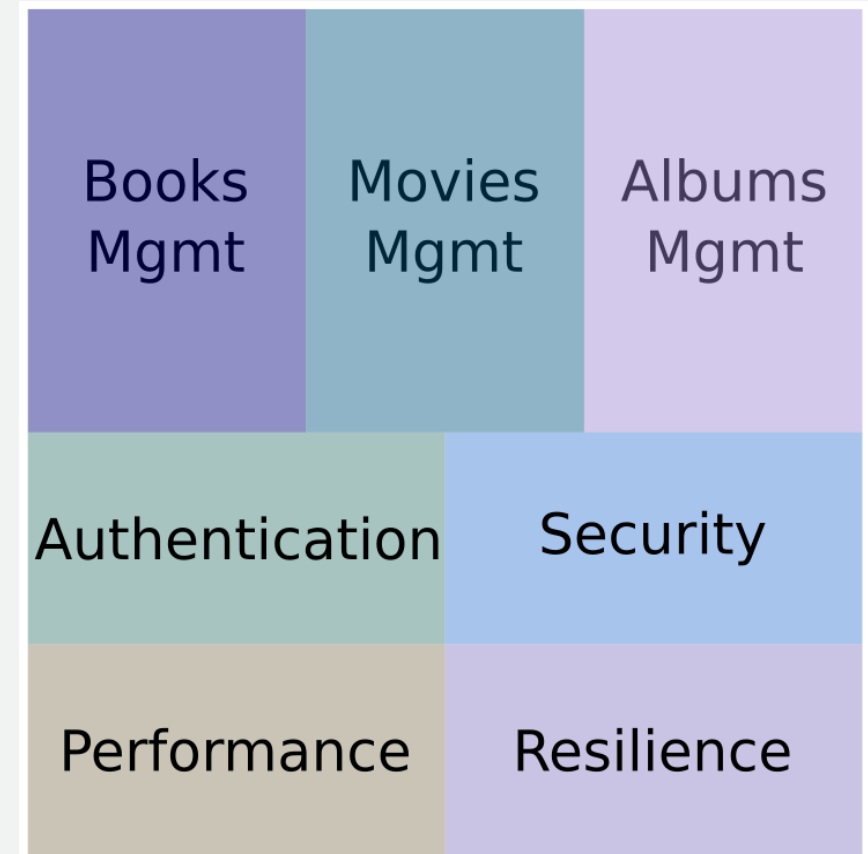


Depuis la formalisation du besoin, on aboutit après un travail d'analyse et de conception au développement d'un système qui implante un certain nombre d'interfaces

**Quels sont les problèmes posés par cette solution ?**

# Application monolithique

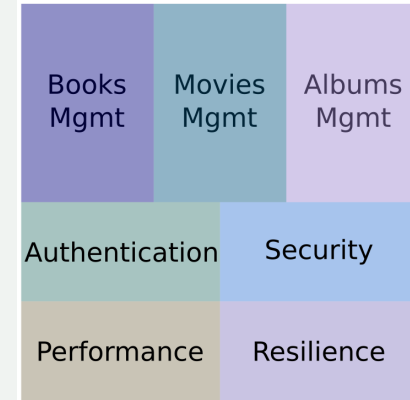
- Applications dotées de nombreuses fonctionnalités, toutes regroupées en un seul exécutable logique
- Avec l'évolution des besoins et l'élargissement du périmètre fonctionnel, une application peut grossir et se complexifier au point où plus grand monde n'arrive à comprendre son architecture
- De fait, les capacités d'évolution d'une application monolithiques s'amenuisent avec temps
  - L'organisation, les usages et les compétences changent, les standards et les technologies évoluent
  - Le passage à l'échelle s'avère être de plus en plus difficile et sa **réactivité** se dégrade





# Architectures monolithiques

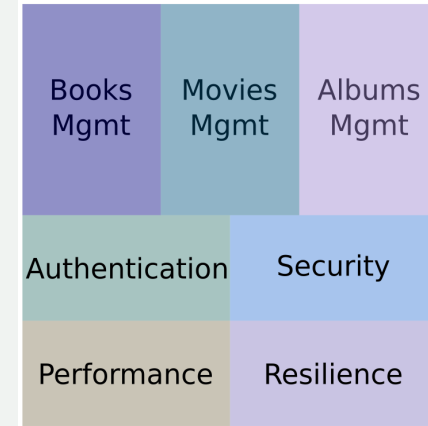
## Problèmes



- Les avancées technologiques ces dernières années et les demandes croissantes en fonctionnalités posent de grands défis dans la conception, le développement et l'opérationnalisation de systèmes informatiques.
  - Les architectures logicielles "classiques", monolithiques, ne répondent pas à ces défis et de nouvelles architectures plus flexibles, plus résilientes et plus faciles à maintenir sont aujourd'hui nécessaires.
- **Évolutivité:** Il peut être coûteux de faire évoluer l'ensemble de l'application alors que bien souvent une seule partie fait l'objet d'une forte demande.
  - L'idéal serait de pouvoir ne faire évoluer que les composants nécessaires, ce qui se traduirait par une utilisation plus efficace des ressources.
- **Goulets d'étranglement:** Toute modification, aussi minime soit-elle, nécessite le redéploiement de l'ensemble de l'application. Cela peut ralentir les processus de développement et de déploiement.
  - L'idéal serait de permettre un déploiement indépendant des différents services qui constituent l'application, ce qui faciliterait des mises à jour plus rapides et plus fréquentes.

# Architectures monolithiques

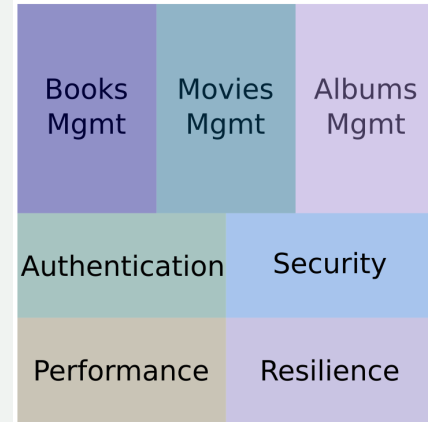
## Problèmes



- **Gestion de la pile technologique:** Une application monolithique est généralement limitée à une seule pile technologique.
  - Il serait plus pratique de pouvoir développer et combiner des services sur la base de technologies, de cadres et de langages différents qui conviennent le mieux à leurs exigences spécifiques.
- **Gestion de la complexité:** Au fur et à mesure qu'une application monolithique se développe, sa base de code peut devenir lourde et complexe, ce qui la rend difficile à comprendre et à maintenir.
  - Le fait de décomposer l'application en éléments composables plus petits rendrait l'application plus facile à développer et à maintenir.
- **Fiabilité:** Dans une architecture monolithique, un bogue dans n'importe quelle partie de l'application peut potentiellement entraîner l'effondrement de l'ensemble du système.
  - Une meilleure isolation des défaillances n'entraînerait pas nécessairement l'arrêt de l'ensemble de l'application.

# Architectures monolithiques

## Problèmes



- **Manque de flexibilité:** Les fournisseurs de solutions ont plus que jamais besoin de se reposer sur des pratiques **plus agiles**.
  - Différentes équipes pourraient travailler simultanément sur différents services, ce qui accélérerait le développement et l'innovation.
- **Dépendances aux fournisseurs:** Les architectures monolithiques peuvent conduire à un verrouillage du fournisseur ou de la plateforme.
  - Il faudrait pouvoir utiliser des services provenant de différents fournisseurs ou plateformes, ce qui réduirait le risque de verrouillage.
- **Gestion du travail:** La gestion d'une grande équipe sur une base de code monolithique unique peut s'avérer difficile.
  - L'idéal serait de se reposer sur une approche qui permette à des équipes plus petites et plus ciblées de s'approprier des services spécifiques, ce qui améliorerait la coordination et l'efficacité.



# Illustration des problèmes

- En 2013, le site web amazon.com a connu une panne importante qui a entraîné une perte d'environ 66 240 dollars par minute, soit un total de près de 2 millions de dollars sur la base de son chiffre d'affaires net de 2012. Cet incident met en évidence l'impact financier que les pannes de système peuvent avoir sur une entreprise, en particulier pour les sociétés opérant à l'échelle d'Amazon.
  - <https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/?sh=7b9f109c495c>
- En 2011, le site web de Walmart a dû faire face à la forte charge de trafic du Black Friday, ce qui a entraîné des ruptures de pages de paiement, des paniers d'achat vides et des erreurs de connexion. Cette situation a non seulement frustré les clients, mais a également entraîné des pertes de ventes et nui à la réputation de l'entreprise. En outre, les magasins physiques ont été le théâtre d'actes de violence, ce qui a encore terni l'événement.
  - <https://techcrunch.com/2011/11/25/walmart-black-friday/?guccounter=1>

# Réactivité: Par quels moyens ?

- Dans le cas où vous gérez l'infrastructure physique supportant l'exécution de vos applications
  - Il faut procéder à un **compromis entre scalabilité horizontale et scalabilité verticale**
  - Les critères pour la recherche de compromis sont: Les coûts, l'évolutivité, la topographie des nœuds d'exécution, la fiabilité ou la performance
- Dans le cas où vous déployez sur une infrastructure existante (IaaS), la plupart des fournisseurs ont des offres élastiques tenant compte de différentes exigences: Puissance de calcul, latence, mémoire, stockage, etc.
- Au niveau logiciel, un moyen de contribuer à rendre le système réactif est de respecter le principe d'**isolation** entre composants fonctionnels
  - En effet, la dégradation de la réactivité d'une fonction, ou même son arrêt, ne doit pas impacter la réactivité des autres fonctions
- Le respect du principe d'isolation a un impact fort sur l'architecture du système
  - Il est en effet nécessaire d'assurer un **découplage fort** des composants du système pour y arriver
  - On parle alors de service, ou plus précisément de **micro-service**
- En plus du découplage fort, il est recommandé de **dupliquer** les composants sur différents nœuds physiques

# Scalabilité verticale

- La mise à l'échelle verticale consiste à ajouter des ressources à un serveur ou à un système existant afin d'en augmenter la capacité. Il peut s'agir d'une mise à niveau de l'unité centrale, de la mémoire vive, du stockage ou d'autres composants d'une seule machine. La mise à l'échelle verticale est généralement plus simple à mettre en œuvre car elle n'implique pas la complexité de la coordination entre plusieurs systèmes ou des changements dans l'architecture de l'application.
- **Avantages :**
  - Simplicité : Plus facile à mettre en œuvre et à gérer puisqu'elle ne concerne qu'un seul système.
  - Impact immédiat : La mise à niveau du matériel permet d'améliorer rapidement les performances du système.
  - Compatibilité : Moins de risques de devoir modifier le code de l'application.
- **Inconvénients :**
  - Limites physiques : Les contraintes physiques et technologiques limitent les possibilités de mise à niveau d'un système unique.
  - Temps d'arrêt : La mise à niveau du matériel peut nécessiter des temps d'arrêt, ce qui affecte la disponibilité.
  - Coût : le matériel haut de gamme peut être coûteux et la rentabilité diminue à mesure que l'on s'approche des limites supérieures de la technologie disponible.

# Scalabilité horizontale

- La mise à l'échelle horizontale consiste à ajouter des machines ou des instances à un pool de ressources afin de répartir la charge et d'augmenter la capacité. Cette approche est privilégiée dans les systèmes distribués, tels que les architectures en microservices, où les charges de travail peuvent être réparties sur plusieurs serveurs.
- **Avantages :**
  - Évolutivité : Évolution pratiquement illimitée par l'ajout de machines en fonction des besoins.
  - Flexibilité : Possibilité d'augmenter la capacité pour répondre à la demande et de la réduire lorsque la demande diminue, ce qui permet d'optimiser l'utilisation des ressources et les coûts.
  - Tolérance aux pannes : Amélioration de la résilience et de la disponibilité puisque la défaillance d'un nœud n'entraîne pas l'effondrement de l'ensemble du système.
- **Inconvénients :**
  - Complexité : Plus complexe à mettre en œuvre et à gérer en raison de la nature distribuée de l'architecture.
  - Frais généraux : Nécessite des mécanismes pour l'équilibrage de la charge, la cohérence des données et la communication entre les services.
  - Coût initial : Peut impliquer des coûts d'installation initiaux plus élevés et une plus grande complexité opérationnelle.

# Reactive Manifesto

Selon le "Reactive Manifesto" (<https://www.reactivemanifesto.org/>):

"Les systèmes modernes font face à des contraintes de plus en plus fortes. Pour y répondre, les systèmes doivent être plus robustes, plus résilients, plus flexibles et mieux placés pour répondre aux exigences modernes [...]"

Nous voulons des systèmes réactifs, résilients, élastiques et pilotés par les messages. Nous les appelons systèmes réactifs.

Les systèmes construits en tant que systèmes réactifs sont plus flexibles, faiblement couplés et évolutifs. Ils sont donc plus faciles à développer et à modifier. Ils sont beaucoup plus tolérants à l'égard des défaillances et, lorsque celles-ci se produisent, ils y font face avec élégance plutôt qu'en catastrophe. Les systèmes réactifs offrent aux utilisateurs un retour d'information interactif efficace."

**Un système réactive est: responsive, résilient, élastique, message-driven.**

# Reactive Manifesto

## **Le système doit être responsive :**

Le système réagit en temps voulu, dans la mesure du possible. La réactivité est la pierre angulaire de la convivialité et de l'utilité, mais plus encore, la réactivité signifie que les problèmes peuvent être détectés rapidement et traités efficacement. Les systèmes réactifs s'attachent à fournir des temps de réponse rapides et cohérents, en établissant des limites supérieures fiables afin de fournir une qualité de service constante. Ce comportement cohérent simplifie à son tour le traitement des erreurs, renforce la confiance de l'utilisateur final et l'incite à interagir davantage.

## **Le système doit être résilient :**

Le système reste réactif en cas de défaillance. Cela ne s'applique pas seulement aux systèmes hautement disponibles et critiques - tout système qui n'est pas résilient ne sera pas réactif après une panne. La résilience est obtenue par la réplication, le confinement, l'isolation et la délégation. Les défaillances sont contenues dans chaque composant, isolant les composants les uns des autres et garantissant ainsi que certaines parties du système peuvent tomber en panne et se rétablir sans compromettre le système dans son ensemble. La reprise de chaque composant est déléguée à un autre composant (externe) et la haute disponibilité est assurée par la réplication si nécessaire. Le client d'un composant n'a pas à gérer ses défaillances.



# Reactive Manifesto

## **Le système doit être élastique :**

Le système reste réactif sous une charge de travail variable. Les systèmes réactifs peuvent réagir aux changements du débit d'entrée en augmentant ou en diminuant les ressources allouées pour servir ces entrées. Cela implique des conceptions sans points de contention ni goulots d'étranglement centraux, ce qui permet de partager ou de répliquer les composants et de répartir les entrées entre eux. Les systèmes réactifs prennent en charge les algorithmes de mise à l'échelle prédictifs, ainsi que réactifs, en fournissant des mesures de performance pertinentes en temps réel. Ils assurent l'élasticité de manière rentable sur des plates-formes matérielles et logicielles de base.

## **Le système doit être "Message-driven" :**

Les systèmes réactifs s'appuient sur le passage de messages asynchrones pour établir une frontière entre les composants qui garantit un couplage faible, l'isolation et la transparence de l'emplacement. Cette frontière fournit également les moyens de déléguer les défaillances sous forme de messages. L'utilisation du passage de messages explicite permet la gestion de la charge, l'élasticité et le contrôle du flux en façonnant et en surveillant les files d'attente de messages dans le système et en appliquant une pression de retour si nécessaire. L'utilisation de la messagerie transparente comme moyen de communication permet à la gestion des pannes de fonctionner avec les mêmes constructions et la même sémantique à travers un cluster ou au sein d'un seul hôte. La communication non bloquante permet aux destinataires de ne consommer des ressources que lorsqu'ils sont actifs, ce qui réduit la surcharge du système.

# Micro-services: Présentation

- Les micro-services tentent d'apporter une réponse aux problèmes posés par les applications monolithiques par un style architectural visant:
  - Un couplage faible entre composants
  - Une meilleure cohérence dans l'hétérogénéité des composants
  - Une plus grande réactivité face aux différents événements pouvant survenir pendant tout le cycle de vie du logiciel (panne, pic d'utilisation, ...)
- Les micro-services permettent:
  - Une meilleure évolution dans le temps des applications
  - Une meilleure maintenabilité
  - Une hétérogénéité contrôlée
  - Un passage à l'échelle facilité
  - Une résilience accrue

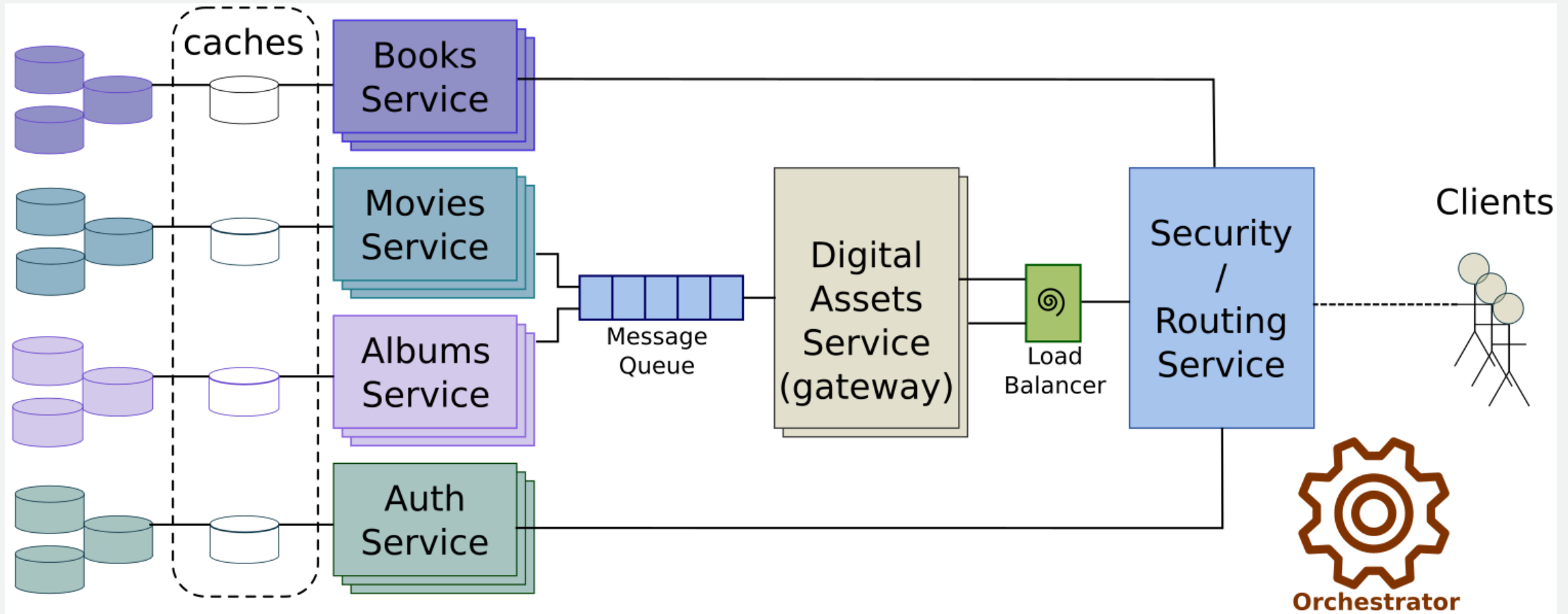
# Avantages des micro-services

- **Évolutivité** : Les micro-services peuvent être mis à l'échelle de manière indépendante pour répondre à la demande. La défaillance du site web de Walmart démontre la nécessité de systèmes capables de s'adapter dynamiquement aux pics de trafic, ce qui est un point fort des micro-services.
- **Résilience** : L'architecture en micro-services peut isoler les défaillances, empêchant ainsi qu'un seul point de défaillance n'entraîne l'arrêt de tout le système. La panne d'Amazon montre le coût élevé des temps d'arrêt, soulignant la valeur d'une architecture résiliente que les micro-services peuvent fournir.
- **Agilité** : Les micro-services permettent un déploiement plus rapide des mises à jour et des corrections. En réponse à des problèmes ou à des demandes changeantes, les entreprises peuvent mettre à jour ou faire évoluer des services individuels sans redéployer l'ensemble de l'application, ce qui permet d'éviter les pannes ou d'y remédier rapidement.

# Avantages des micro-services

- **Diversification des technologies** : Les micro-services favorisent l'utilisation de différentes technologies entre les services, ce qui permet d'adopter la technologie la plus adaptée aux besoins de chaque service. Cela aurait pu contribuer à optimiser les performances et la fiabilité de fonctionnalités spécifiques, telles que le processus d'encaissement de Walmart.
- **Décentralisation** : En décentralisant le contrôle et les données, les micro-services réduisent le risque d'incohérences et de défaillances généralisées des données. Cette approche pourrait atténuer les problèmes tels que ceux rencontrés par Walmart, où des défaillances à l'échelle du système ont entraîné une mauvaise expérience client.
- **Expérience du client** : En fin de compte, l'architecture en micro-services favorise une meilleure expérience client en garantissant que les services sont disponibles, réactifs et à jour. Les incidents d'Amazon et de Walmart illustrent l'impact direct des défaillances du système sur la satisfaction des clients et la réputation de l'entreprise.

# Exemple d'architecture en micro-services



- Cet exemple est plus conforme à ce qu'on attend d'une application réactive - **Pourquoi ? Quels avantages ?**

# Passage aux micro-services

- **Développement:** Le développement d'un micro-service est fortement contraint par les performances; il nécessite un cadre de programmation particulier, permettant de créer des applications réactives, à faible empreinte mémoire. Exemple: Micronaut
- **Conteneurisation:** Une fois le micro-service implanté, il faut le conteneuriser de manière à pouvoir le déployer n'importe où, avec notamment **docker**
- **Communication:** Le manifeste Reactive stipule que les applications doivent être message-driven de manière à assurer un couplage plus lâche et une meilleure isolation. Exemple: Kafka
- **Orchestration:** Les micro-services nécessitent des outils permettant que l'application soit toujours dans un état acceptable; que les micro-services puissent communiquer de manière transparente, qu'ils soient relancés automatiquement en cas de panne ou qu'ils soient dupliqués en cas de pic de charge. Exemple: Kubernetes



# Passage aux micro-services

- **Tests:** De fait, le test d'applications à base de micro-services nécessitent des frameworks particuliers permettant le test de chaque micro-service en mimant son environnement. Exemple: Testcontainers
- **Configuration:** La configuration d'applications à base de micro-services nécessitent une approche centralisée pour plus de cohérence. Exemple: Spring Cloud Config
- **Logging:** Comme pour la configuration, les applications à base de micro-services nécessitent également une approche centralisée permettant d'analyser la chaîne de traitements et remonter à la source du problème. Exemple: Zipkin ou Jaeger
- **Monitoring:** Les décisions de reconfiguration de l'application sont basées sur les observations faites sur l'ensemble des micro-services. Là encore, nous avons besoins de centralisée les traces d'exécution, les visualiser et éventuellement prédire de potentiels problèmes. Exemples: Prometheus / Grafana

# Règles de bonnes pratiques

- La **méthodologie des douze facteurs** est un ensemble de douze bonnes pratiques pour développer des applications destinées à fonctionner comme un service, élaborée à l'origine par Heroku pour les applications déployées en tant que services sur leur cloud et qui s'est avérée suffisamment générique pour tout développement de logiciel en tant que service (SaaS).
- Les douzes bonnes pratiques sont les suivantes:
  - Assurer le suivi de vos changement avec un système de contrôle de versions
  - Gérer de manière explicite toutes les dépendances (maven, pip, ...)
  - Centraliser la configuration des services (Spring cloud, Ansible)
  - Utiliser des interfaces standards afin de s'abstraire des services techniques (JPA pour les BDD)
  - Bien séparer les étapes de construction de l'application (Compilation, test, déploiement, exécution)
  - Les services doivent être stateless
  - Les services doivent être complètement autonomes, et embarquer toutes leur dépendance (ex: Spring / Jetty)
  - La concurrence doit être supportée au niveau des processus, les threads étant vus comme des détails d'implantation
  - L'arrêt d'un service ne doit pas générer d'effet de bord
  - Éviter les gaps entre environnements de développement et de production
  - Gérer les logs de manière standardisée et centralisée
  - Automatiser autant que possible les tâches d'administration par des scripts

# Exemple de success story

- Spotify, le service de streaming musical, a migré d'une architecture monolithique vers une architecture en micro-services en 2013-2014. Avant cela, leur application backend était un gros bloc monolithique Java déployé sur une centaine de serveurs.
- Les problèmes rencontrés avec cette architecture monolithique étaient :
  - Des déploiements longs et risqués à cause de la taille du monolithe
  - Des temps de build très longs
  - Des problèmes de scalabilité, l'architecture n'était pas adaptée à la croissance de Spotify
  - Un couplage fort rendant les changements difficiles et lents
- Spotify a donc décidé de migrer vers une architecture en micro-services, en découpant progressivement son monolithe en petits services autonomes. Chaque service peut être développé, déployé et scalé indépendamment par une équipe dédiée.

# Exemple de success story (Cont'd)

- Les bénéfices observés ont été :
  - Des déploiements beaucoup plus rapides et fréquents (passage de un par semaine à plusieurs par jour)
  - Une bien meilleure scalabilité / résilience
  - Des temps de build réduits permettant un développement plus rapide
  - Plus d'autonomie pour les équipes qui maîtrisent leur service de bout en bout
- Spotify utilise son propre outil de déploiement continu (Helios) pour gérer ses centaines de micro-services dans un cluster.
- Cet exemple montre bien les avantages que peut apporter une architecture en micro-services pour une application à grande échelle comme Spotify, même si la migration depuis un monolithe est un processus long et complexe qui doit être bien planifié.

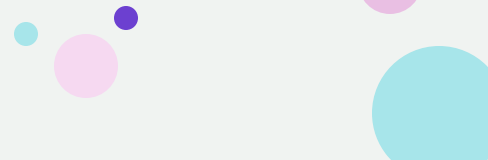
# Les défis de l'adoption

- **Complexité du réseau** : La communication entre les services distribués peut être complexe à gérer, nécessitant des mécanismes avancés de découverte de services et d'équilibrage de charge.
- **Gestion des dépendances** : Les micro-services peuvent avoir des dépendances multiples et variées, rendant la gestion des versions et la coordination entre les services plus complexe.
- **Sécurité et isolation** : Assurer la sécurité et l'isolation appropriées entre les services distribués est crucial mais difficile, en particulier dans des environnements cloud publics.
- **Surveillance et débogage** : Le suivi et le diagnostic des problèmes dans un environnement de micro-services peuvent être difficiles en raison de la nature distribuée et des interactions entre les services.
- **Consistance des données** : Maintenir la consistance des données à travers différents services nécessite des stratégies efficaces de gestion des transactions et de synchronisation.

A cluster of decorative circles in shades of pink, purple, and orange in the top-left corner.

# Prérequis

- Afin de suivre les exercices, il faut disposer des outils suivants
  - Docker
  - Docker-compose
- Clôner les sources sur Github
  - <https://github.com/akoudri/docker-training.git>



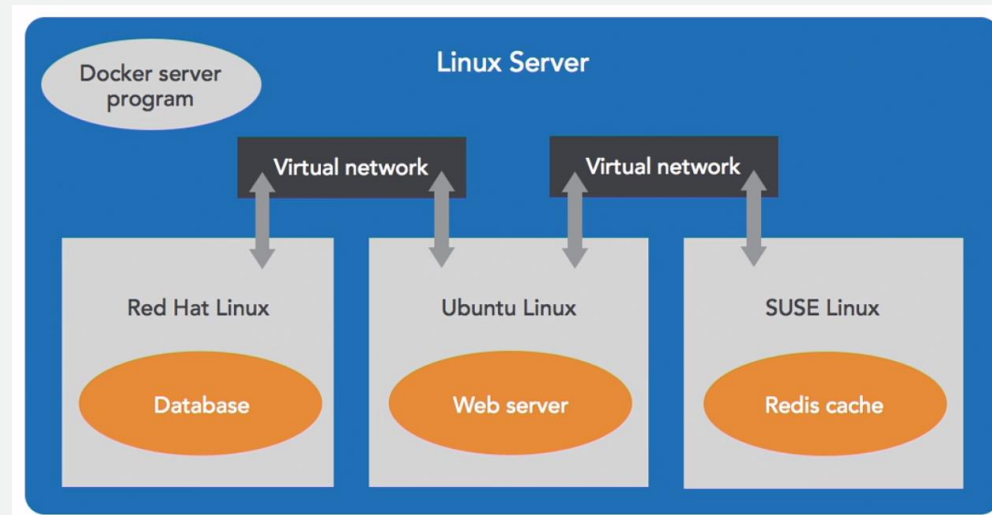


The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of circles in shades of pink, purple, and orange. The top-right corner has circles in blue, purple, and orange. The bottom-right corner is decorated with circles in teal, pink, and purple. The title 'Présentation de Docker' is centered in a dark purple font.

# Présentation de Docker

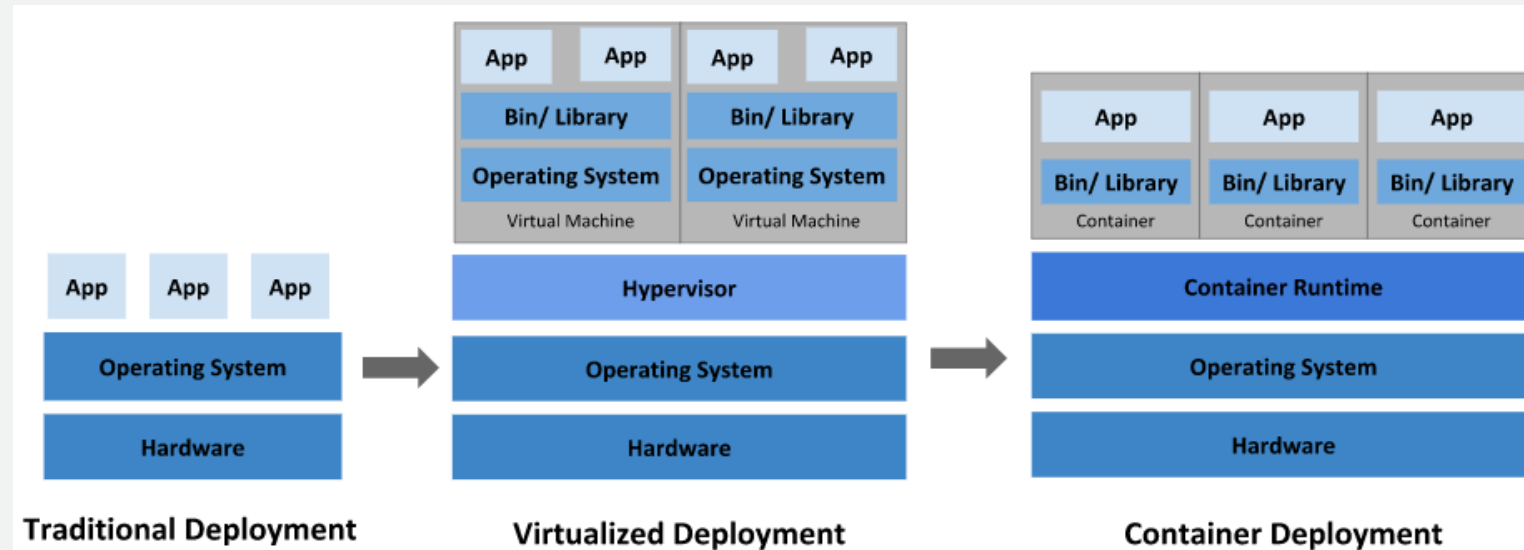
# Introduction

- Docker est une technologie permettant de créer des composants "auto-suffisants"
  - Respectant le principe d'isolation
  - Ne nécessitant aucune dépendance externe, à la différence d'un paquet deb ou rpm
  - Equivalent au snap dans ubuntu
- Docker fournit un ensemble d'outils permettant de créer, de configurer et de gérer le cycle de vie de conteneurs et de leur assemblage
- Docker représente un écosystème de composants configurables et prêts à l'emploi



# Conteneur vs Machine Virtuelle

- Une machine virtuelle embarque tout ce dont elle a besoin pour fonctionner: OS, middleware, librairies
- Parmi les inconvénients de la machine virtuelle, nous pouvons citer:
  - Le temps nécessaire pour construire une image sur mesure
  - La taille et le temps de chargement
- Le conteneur représente un exécutable indépendant embarquant toutes ses dépendances, selon le principe d'isolation, à l'exception des services de base

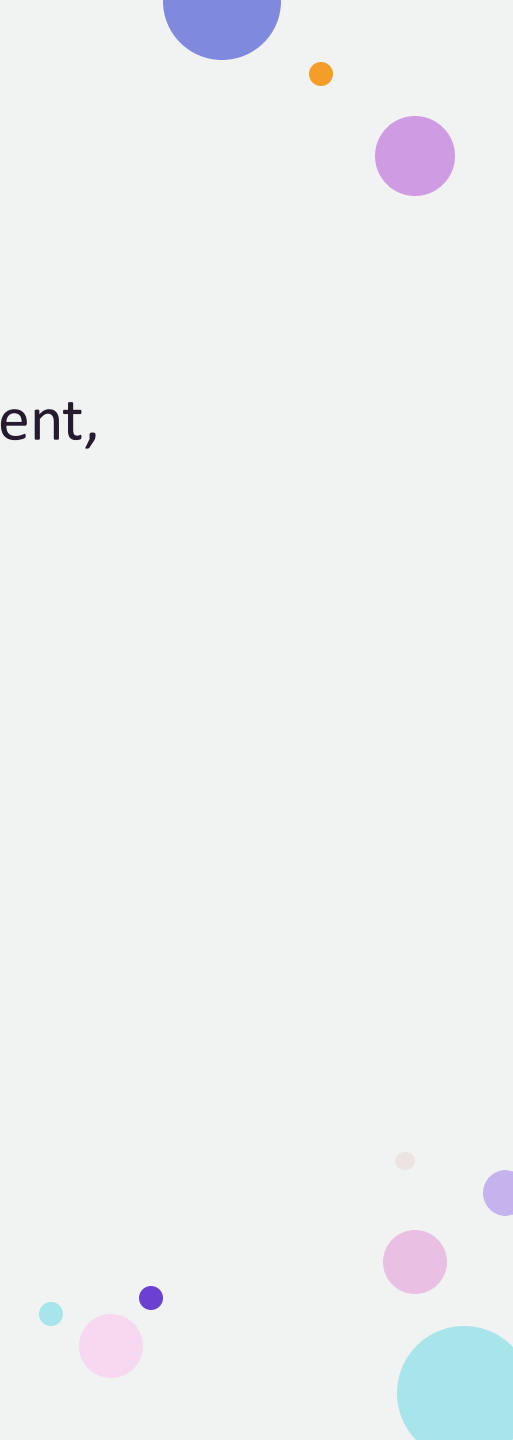


# Définition

- Un conteneur est une unité logicielle standardisée qui encapsule une application et toutes ses dépendances (bibliothèques, outils, configurations, etc.) dans un environnement isolé et portable, permettant son exécution cohérente et fiable sur n'importe quel système compatible.
- Les conteneurs simplifient le déploiement d'applications en éliminant les problèmes liés aux différences d'environnement ("ça marche sur ma machine, mais pas en production").
- Ils sont légers et efficaces, car ils partagent le noyau du système hôte, contrairement aux machines virtuelles qui nécessitent un système d'exploitation complet.



# Objectifs des conteneurs

- **La portabilité** : Fonctionnement identique, quel que soit l'environnement,
  - **L'isolation** : Suppression des conflits entre applications,
  - **L'efficacité** : Optimisation des ressources,
  - **La scalabilité** : Mise à l'échelle facilitée,
  - **La cohérence** : Reproductibilité des environnements,
  - **L'automatisation** : Intégration avec CI/CD.
- 

# Caractéristiques des conteneurs

- **Unité d'exécution isolée** : Un conteneur est une unité logicielle qui encapsule une application et ses dépendances, permettant de l'exécuter de manière isolée de l'environnement hôte et des autres conteneurs.
- **Léger et portable** : Contrairement aux machines virtuelles, les conteneurs partagent le noyau du système d'exploitation hôte, ce qui les rend plus légers et plus rapides à démarrer. Ils sont également facilement déplaçables d'un environnement à un autre.
- **Environnement cohérent** : Les conteneurs garantissent que l'application fonctionne de la même manière, quel que soit l'environnement (développement, test, production), ce qui réduit les problèmes liés aux différences d'environnement.



# Caractéristiques des conteneurs

- **Basé sur des images** : Un conteneur est instancié à partir d'une image, qui est un modèle immuable contenant le code de l'application, les bibliothèques et les configurations nécessaires. Les images sont souvent stockées dans des registres comme Docker Hub.
- **Isolation des ressources** : Les conteneurs utilisent des mécanismes comme les namespaces et les cgroups pour isoler les ressources (CPU, mémoire, réseau) et limiter l'impact d'un conteneur sur les autres.
- **Éphémère et scalable** : Les conteneurs sont conçus pour être éphémères, c'est-à-dire qu'ils peuvent être créés, arrêtés et supprimés rapidement. Cela facilite la mise à l'échelle horizontale des applications.
- **Écosystème riche** : Docker, Kubernetes et d'autres outils fournissent un écosystème complet pour gérer le cycle de vie des conteneurs, du développement au déploiement en production.

# Histoire de Docker

- **Fondation** : Docker a été fondé en 2013 par Solomon Hykes en France, dans le cadre de la société dotCloud, une plateforme de cloud computing.
- **Naissance du projet Docker** : Docker est né d'un projet interne chez dotCloud pour simplifier le déploiement d'applications. Il a rapidement gagné en popularité et est devenu un projet open-source en 2013.
- **Croissance rapide** : Docker a révolutionné le développement logiciel en popularisant l'utilisation des conteneurs, devenant un standard de facto dans l'industrie.



# Qu'est-ce que Docker ?

- **Une entreprise** : Docker, Inc. est la société qui développe et soutient les outils Docker.
- **Une technologie** : Docker est une plateforme open-source qui permet de créer, déployer et gérer des applications dans des conteneurs.
- **Un écosystème** : Docker a créé un écosystème complet autour des conteneurs, incluant des outils comme Docker Engine, Docker Compose, Docker Hub, et bien d'autres.

# Produits de Docker

- **Docker Engine :**
  - Le cœur de Docker, un runtime qui permet de créer et d'exécuter des conteneurs.
  - Il inclut un démon (Docker Daemon), une API et une interface en ligne de commande (CLI).
- **Docker Hub :**
  - Un registre public (et privé) pour stocker et partager des images Docker.
  - Les développeurs peuvent y trouver des images officielles (comme Ubuntu, MySQL, Node.js) ou publier leurs propres images.
- **Docker Compose :**
  - Un outil pour définir et gérer des applications multi-conteneurs à l'aide d'un fichier YAML.
  - Idéal pour les environnements de développement local.

# Produits de Docker

- **Docker Desktop :**
  - Une application pour les systèmes Windows et macOS qui permet de développer et tester des conteneurs localement.
  - Il inclut Docker Engine, Kubernetes et d'autres outils.
- **Docker Swarm :**
  - Un outil d'orchestration de conteneurs intégré à Docker, permettant de gérer un cluster de conteneurs.
  - Moins complexe que Kubernetes, mais moins puissant.

# Impact sur l'industrie

- **Standardisation des conteneurs** : Docker a popularisé et standardisé l'utilisation des conteneurs, en simplifiant leur adoption.
- **Adoption massive** : Docker est devenu un outil incontournable pour les développeurs, les équipes DevOps et les entreprises.
- **Écosystème Kubernetes** : Bien que Docker ait initialement dominé l'orchestration de conteneurs, Kubernetes (projet open-source soutenu par Google) est devenu le standard pour l'orchestration. Docker a intégré Kubernetes dans Docker Desktop pour rester compétitif.

# Modèle économique

- **Open-source** : Les outils de base de Docker (comme Docker Engine) sont open-source et gratuits.
- **Services payants** : Docker propose des services payants pour les entreprises, comme :
  - **Docker Hub Pro** : Pour les équipes de développement.
  - **Docker Enterprise** : Une solution pour les entreprises, incluant des fonctionnalités de sécurité et de gestion.
  - **Docker Desktop** : Gratuit pour les particuliers, mais payant pour les grandes entreprises.

# Évolutions récentes

- **Refocus sur les développeurs** : Ces dernières années, Docker s'est recentré sur les développeurs individuels et les petites équipes, en simplifiant ses outils et en améliorant l'expérience utilisateur.
- **Concurrence** : Docker fait face à une concurrence accrue, notamment de la part de Kubernetes, Podman (Red Hat) et d'autres solutions de conteneurisation.
- **Partenariats** : Docker collabore avec des acteurs majeurs comme Microsoft (intégration avec Azure) et Amazon Web Services (AWS).



# Le choix de Docker

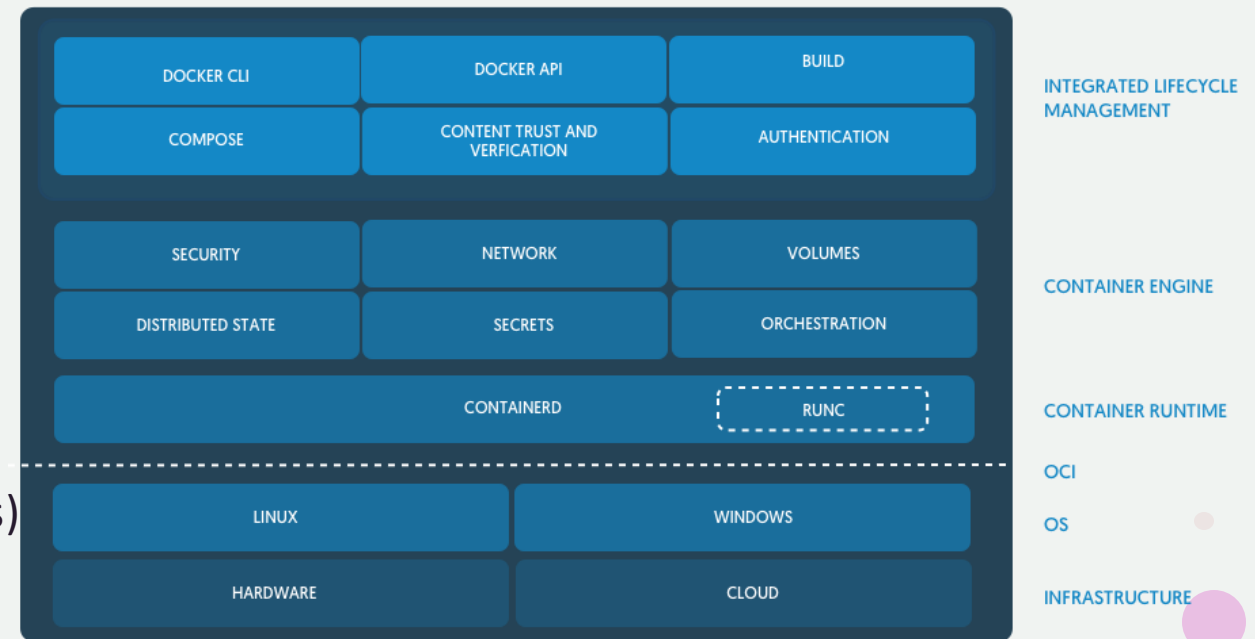
- Docker a révolutionné la manière dont les applications sont développées, déployées et gérées.
- En popularisant les conteneurs, Docker a permis aux entreprises d'accélérer leur transformation numérique, d'améliorer leur efficacité et de réduire leurs coûts.
- Bien que la concurrence soit forte, Docker reste un acteur clé dans l'écosystème des conteneurs.
- La popularité de Docker repose sur les caractéristiques suivantes:
  - **Simplicité** : Docker a rendu les conteneurs accessibles à tous, même aux débutants.
  - **Portabilité** : Les applications Docker fonctionnent de la même manière sur n'importe quel environnement.
  - **Écosystème riche** : Docker propose une suite complète d'outils pour chaque étape du cycle de vie des applications.

# La question du droit

- **Docker Engine** (le runtime qui permet de créer et d'exécuter des conteneurs) est open-source et distribué sous la licence Apache 2.0.
  - Vous pouvez utiliser Docker Engine gratuitement, même pour des projets commerciaux.
  - Vous pouvez modifier et redistribuer le code source de Docker Engine, à condition de respecter les termes de la licence Apache 2.0 (comme inclure une notice de licence et les mentions de copyright).
- Licences des **images tierces** :
  - Les images disponibles sur Docker Hub peuvent avoir des licences différentes. Par exemple, certaines images officielles (comme MySQL, PostgreSQL) sont open-source, mais d'autres peuvent avoir des restrictions commerciales.
  - Toujours vérifier la licence d'une image avant de l'utiliser dans un projet commercial.
- **Docker Desktop** pour les grandes entreprises :
  - Si vous êtes une grande entreprise, vous devez acheter une licence pour Docker Desktop. L'utilisation non autorisée peut entraîner des problèmes juridiques.

# Containerd

- La notion de conteneur n'est pas un concept natif des OS
- Containerd est une couche d'abstraction entre les outils de gestion de conteneurs et le système d'exploitation
- Il implante tous les concepts liés aux conteneurs et à leur gestion en cachant les détails bas niveau liés à l'appel des primitives de l'OS
- Il offre des services de haut niveau pour gérer:
  - L'orchestration
  - La gestion des volumes
  - La sécurité
  - L'authentification
  - Les réseaux
  - ...
- Il est défini et utilisé par docker et kubernetes (entre autres)



# Containerd

- À l'origine, le "Docker Engine" (le démon dockerd) était une application monolithique.
- Dockerd gère tout :
  - L'interface en ligne de commande (docker build, docker run, etc.).
  - Le processus de build des images (lecture des Dockerfiles).
  - Le stockage et la gestion des images.
  - La gestion des volumes et des réseaux.
  - L'exécution et la supervision effective des conteneurs (création des namespaces, cgroups, lancement des processus).
- Avec la montée en puissance des orchestrateurs comme Kubernetes, il est devenu clair qu'une séparation des responsabilités était nécessaire.
  - Les orchestrateurs avaient surtout besoin d'une couche fiable et standardisée pour exécuter des conteneurs, sans forcément dépendre de toutes les fonctionnalités "haut niveau" de Docker (comme le build ou la gestion avancée du réseau Docker).

# Containerd

- Docker Inc. a alors décidé d'extraire la partie centrale responsable de l'exécution des conteneurs de son démon monolithique. Ce composant extrait est devenu containerd.
- containerd se concentre sur le cycle de vie complet du conteneur :
  - Gestion des images (pull, push, stockage).
  - Gestion de l'exécution des conteneurs (démarrer, arrêter via un runtime OCI comme runc).
  - Gestion du stockage bas niveau et des attachements réseau pour les conteneurs.
- Il expose une API pour que d'autres outils puissent l'utiliser.
- Pour favoriser l'adoption et la standardisation, Docker Inc. a donné containerd à la Cloud Native Computing Foundation (CNCF), où il est maintenant un projet "Graduated" (mature et largement adopté).
- Aujourd'hui, lorsque vous installez et utilisez Docker, le démon dockerd ne gère plus directement l'exécution des conteneurs. Il délègue cette tâche à containerd.

# Containerd et kubernetes

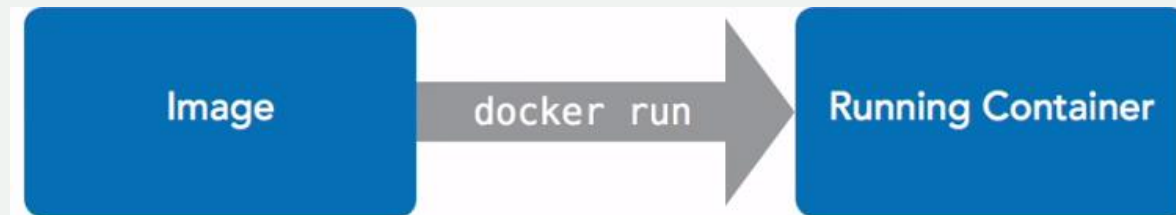
- Containerd est plus léger que l'ensemble du Docker Engine, car il ne contient pas les fonctionnalités de build, de gestion de réseau complexe Docker, etc. C'est idéal pour les nœuds workers d'un cluster où l'on veut juste exécuter des conteneurs efficacement.
- Containerd implémente l'interface CRI (Container Runtime Interface) de Kubernetes. Cela permet à kubelet (l'agent Kubernetes sur chaque nœud) de communiquer directement avec containerd pour gérer les pods et les conteneurs, sans passer par l'intermédiaire du démon Docker complet. C'est la raison pour laquelle le "dockershim" (la couche de compatibilité entre kubelet et Docker Engine) a été déprécié puis retiré de Kubernetes.
- Étant un projet CNCF, containerd est indépendant de la plateforme Docker complète.

# Installation

- Sur Linux (de préférence)
  - La plupart des distributions inclut docker dans leur store
  - Sur ubuntu:
    - `sudo apt install docker.io docker-compose-plugin`
    - `sudo usermod -aG docker $USER`
  - Sur linux de manière générale :
    - `curl -o get-docker.sh https://get.docker.com`
    - `chmod +x get-docker.sh`
    - `sudo ./get-docker.sh`
    - `sudo usermod -aG docker $USER`
- Sur Windows, il faut au préalable installer WSL2
- Sur Mac, une archive dmg est téléchargeable sur le site officiel de docker

# Concepts Essentiels

- Une image docker représente un programme sur le système de fichier
- Un conteneur docker représente une instance de programme en mémoire (exécution)
- Il est possible d'instancier (run) plusieurs fois la même images
  - Chaque conteneur gère son propre état
- Lister les images présentes: `$docker images`
- Lister les instances: `$docker ps`
- **Exercice:** lancer la commande `$docker run hello-world`, que se passe-t-il ?
  - Lister les conteneurs
  - Relancer la commande, que se passe-t-il ?





# Anatomie d'une commande docker

- `$ docker [options] commande [arguments...]`
- Il faut garder à l'esprit que la commande docker sert pour l'essentiel à gérer des ressources de différentes natures: images, conteneurs, réseaux ou volumes
- De fait, une commande docker adhère globalement à un pattern:
  - `$ docker <type de ressource> <sous-commande>`
  - On retrouve beaucoup de sous-commandes communes aux différents types de ressources: liste (ls), inspection (inspect), création (create), suppression (rm) ou nettoyage (prune)
  - Exemples:
    - `$docker volume ls`
    - `$docker network create mynet`
    - `$docker image prune`

# Quelques Précisions

- Pour avoir des informations sur le moteur docker:
  - *\$docker info*
- Le client et le moteur ne sont pas nécessairement sur la même machine
  - *\$ssh-add [clef privée]*
  - *\$export DOCKER\_HOST=ssh://utilisateur@adresse-ip-distante*

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of circles in shades of pink, purple, and orange. The top-right corner has circles in blue, orange, and purple. The bottom-right corner is decorated with circles in teal, pink, and purple. The main title is centered in a dark purple font.

# Gestion des images

## Partie 1

# Origine des images

- Les images Docker par défaut proviennent principalement de Docker Hub, le registre public de conteneurs géré par Docker, Inc.
- Docker Hub est la plateforme centrale où sont stockées et partagées les images Docker.
  - C'est en quelque sorte un GitHub pour les images Docker
  - Les images y sont hébergées et accessibles à tous (sans avoir à être authentifié)
- Vous avez également la possibilité de :
  - Utiliser d'autres registres, comme par exemple celui de microsoft (<https://mcr.microsoft.com/>)
  - Utiliser des registres privés (comme AWS ECR, Google Container Registry, ou un registre auto-hébergé).
- **Exercice:** Se rendre sur docker hub (<https://hub.docker.com/>) et rechercher l'image pour postgres
  - Qu'observez-vous ?
  - Remarque: vous avez la possibilité de faire la recherche en lignes de commande:
    - Docker search postgres

# Recommandation sur les images

- **Préférez les images officielles:** Les images officielles sont maintenues par les éditeurs du logiciel ou des organisations de confiance. Elles suivent les meilleures pratiques en matière de sécurité et de configuration.
- **Vérifiez la provenance et la popularité de l'image:** Les images tierces peuvent contenir des vulnérabilités, des logiciels malveillants ou des configurations non optimisées.
- **Choisissez des images à jour et légères:** Les images obsolètes peuvent contenir des vulnérabilités non corrigées ou des versions de logiciels incompatibles. Les images légères réduisent le temps de téléchargement, d'exécution et la surface d'attaque.
- **Scannez les images pour détecter les vulnérabilités:** Les images Docker peuvent contenir des vulnérabilités connues dans leurs packages ou dépendances.

# Recommandation sur les images

- **Utilisez des tags spécifiques** (évitez latest, surtout en production): Le tag latest peut changer sans préavis, ce qui peut introduire des incompatibilités.
- **Vérifiez les licences:** Certaines images peuvent inclure des logiciels sous licence restrictive, ce qui peut poser des problèmes pour une utilisation commerciale.
- **Personnalisez les images si nécessaire:** Les images officielles ou tierces peuvent ne pas répondre exactement à vos besoins.
- **Stockez les images personnalisées dans des registres privés:** Si vous créez des images personnalisées, il est préférable de les stocker dans un registre privé pour des raisons de sécurité et de contrôle.
- **Documentez et versionnez les images:** Une bonne documentation et un versionnement clair facilitent la maintenance et la collaboration.

# Constitution d'une image

- Une image Docker est une unité logicielle qui contient tout ce dont une application a besoin pour s'exécuter de manière isolée et portable.
- Une image Docker est constituée de :
  - Couches empilables (layers) représentant des modifications successives.
  - Un système de fichiers complet incluant l'application et ses dépendances.
  - Une distribution Linux minimale (ou une image vide).
  - Des métadonnées (labels, variables d'environnement, commandes par défaut).
  - Un Dockerfile qui décrit comment construire l'image.

# Constitution d'une image

- **Les couches (layers)**

- Concept : Une image Docker est composée de couches empilables (layers). Chaque couche représente une modification apportée à l'image (ajout de fichiers, installation de logiciels, etc.).
- Immuabilité : Les couches sont immuables, c'est-à-dire qu'une fois créées, elles ne peuvent pas être modifiées. Toute modification crée une nouvelle couche.
- Partage : Les couches sont partagées entre les images. Par exemple, si deux images utilisent la même couche de base (comme ubuntu), cette couche n'est téléchargée qu'une seule fois.

- **Le système de fichiers**

- Contenu : Une image Docker inclut un système de fichiers complet (fichiers binaires, bibliothèques, configurations, etc.) nécessaire pour exécuter l'application.
- Structure : Le système de fichiers est organisé comme un système Linux standard, avec des répertoires comme /bin, /usr, /etc, etc.
- Minimalisme : Les images officielles sont souvent minimalistes pour réduire leur taille (par exemple, alpine utilise seulement 5 Mo).



# Constitution d'une image

- **La distribution de base**

- Base : Une image Docker inclut généralement une distribution Linux minimale (comme alpine, ubuntu, debian) ou une image vide (scratch).
- Pas de noyau : Contrairement à une machine virtuelle, une image Docker ne contient pas de noyau. Elle utilise le noyau du système hôte.

- **Les dépendances de l'application**

- Bibliothèques : Les images incluent les bibliothèques nécessaires pour exécuter l'application (par exemple, glibc pour les applications C, node\_modules pour Node.js).
- Outils : Elles peuvent également inclure des outils de développement ou de débogage (comme curl, vim, git), bien que cela soit déconseillé en production; notamment pour réduire la taille de l'image ou pour des questions de sécurité

- **Les fichiers de l'application**

- L'image contient le code source ou les binaires de l'application.
- Fichiers de configuration : Les fichiers de configuration (comme nginx.conf, env, etc.) sont souvent inclus dans l'image.
- Données statiques : Les fichiers statiques (comme des images, des fichiers HTML, etc.) peuvent également être inclus.

# Constitution d'une image

- **Les métadonnées**

- Labels : Les images peuvent inclure des métadonnées sous forme de labels (par exemple, la version de l'application, le mainteneur, la licence).
- Environnement : Les variables d'environnement par défaut peuvent être définies dans l'image.
- Commandes par défaut : L'image peut spécifier une commande par défaut à exécuter au démarrage du conteneur (via CMD ou ENTRYPOINT dans le Dockerfile).

- **Le Dockerfile**

- Définition : Une image est généralement construite à partir d'un Dockerfile, un fichier texte qui décrit les étapes pour créer l'image.
- Instructions courantes :
  - FROM : Spécifie l'image de base.
  - RUN : Exécute des commandes pour installer des logiciels ou configurer l'image.
  - COPY ou ADD : Ajoute des fichiers ou répertoires à l'image.
  - ENV : Définit des variables d'environnement.
  - CMD ou ENTRYPOINT : Spécifie la commande à exécuter au démarrage du conteneur.

# Constitution d'une image

- Exécutez la commande *\$docker pull python*
  - Qu'observez-vous ?
  - Comment l'expliquer ?
  - Quels sont les avantages ?
- Exécutez la commande *\$docker history python*
  - Qu'observez-vous ?
- Réexécutez la commande avec l'option *--no-trunc*
  - Qu'observez-vous ?
- Exécutez la commande *\$docker inspect python*
  - Qu'observez-vous ?

# Tags des images - Principes

- **Définition** : Un tag est une étiquette associée à une image Docker pour identifier une version ou une variante spécifique.
- **Format** : Un tag est généralement composé d'un nom d'image suivi d'une étiquette, séparés par deux points (:). Par exemple : `nginx:1.23.4` ou `python:3.9-slim`.
- **Cas spécial** : Le tag *latest* est utilisé par défaut si aucun tag n'est spécifié.

# Tags des images - Utilisation

- **Gestion des versions :**

- Les tags permettent de spécifier une version précise d'une image (par exemple, node:18 ou mysql:8.0).
- Cela garantit que votre application utilise toujours la même version du logiciel, même si de nouvelles versions sont publiées.

- **Variantes d'images :**

- Les tags permettent de choisir entre différentes variantes d'une image, comme :
  - Des versions légères (alpine, slim),
  - Des versions avec des outils supplémentaires (-dev, -full),
  - Des versions pour des architectures spécifiques (arm64, amd64).

- **Stabilité et reproductibilité :**

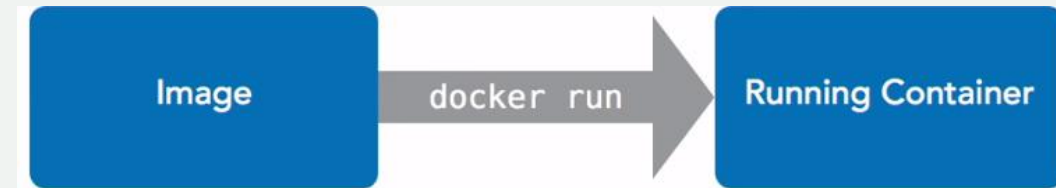
- En utilisant des tags spécifiques, vous assurez que votre application fonctionne de la même manière dans tous les environnements (développement, test, production).
- Note: Le tag latest peut changer sans préavis, ce qui peut introduire des incompatibilités. Utiliser des tags spécifiques évite ce problème.

# Gestion des conteneurs

# Cycle de vie d'un conteneur

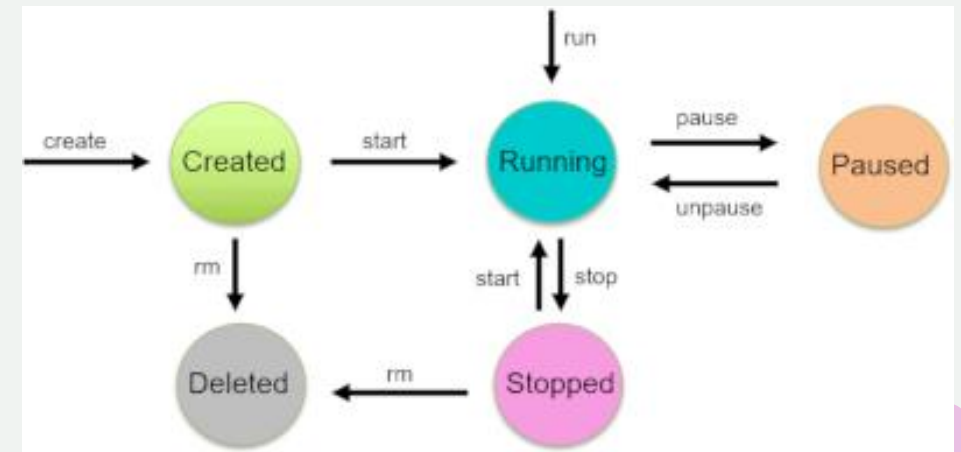
- **Initialisation du conteneur :**

- Description : Un conteneur est une instance d'une image.
- Processus : Il est créé lorsque vous exécutez une image à l'aide de commandes telles que docker run. Cette étape initialise le conteneur en configurant son système de fichiers, son réseau et les autres configurations nécessaires définies dans l'image.
- **Note**: La commande run télécharge l'image si cette dernière est absente



- **Exécution du conteneur :**

- Description : Le conteneur est une instance d'image en cours d'exécution.
- Processus : Il s'exécute de manière isolée mais peut communiquer avec d'autres conteneurs et le système hôte. Vous pouvez interagir avec le conteneur en cours d'exécution, exécuter des commandes à l'intérieur de celui-ci, ou le conteneur peut simplement exécuter un service en arrière-plan.



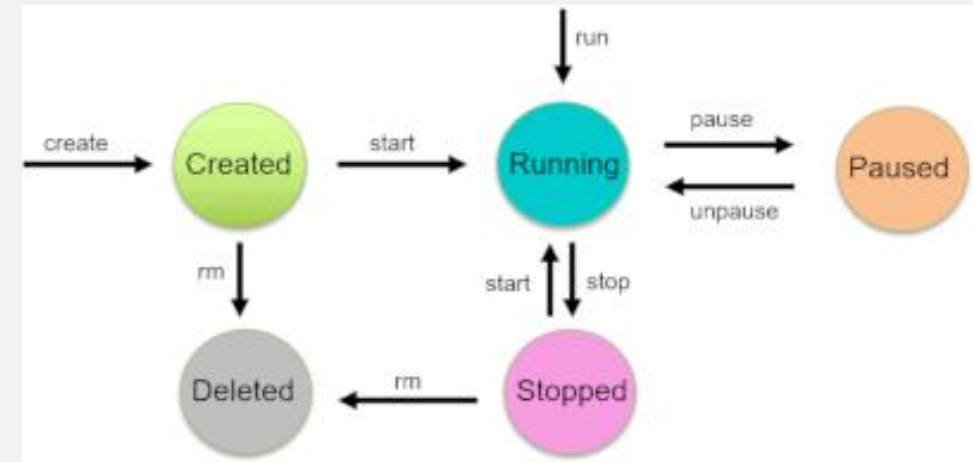
# Cycle de vie d'un conteneur

- **Arrêt des conteneurs :**

- Description : Les conteneurs peuvent être arrêtés et redémarrés
- Processus : Vous pouvez arrêter un conteneur à l'aide de commandes telles que `docker stop`, qui envoie un `SIGTERM` et, après une période donnée, un `SIGKILL`, aux processus s'exécutant dans le conteneur.

- **Suppression du conteneur :**

- Description : Une fois qu'un conteneur a rempli sa fonction, il peut être supprimé.
- Processus : Cette opération s'effectue à l'aide de commandes telles que `docker rm`. Cette étape supprime l'instance du conteneur mais pas l'image sous-jacente. Cela signifie que vous pouvez créer un nouveau conteneur à partir de la même image chaque fois que vous en avez besoin.





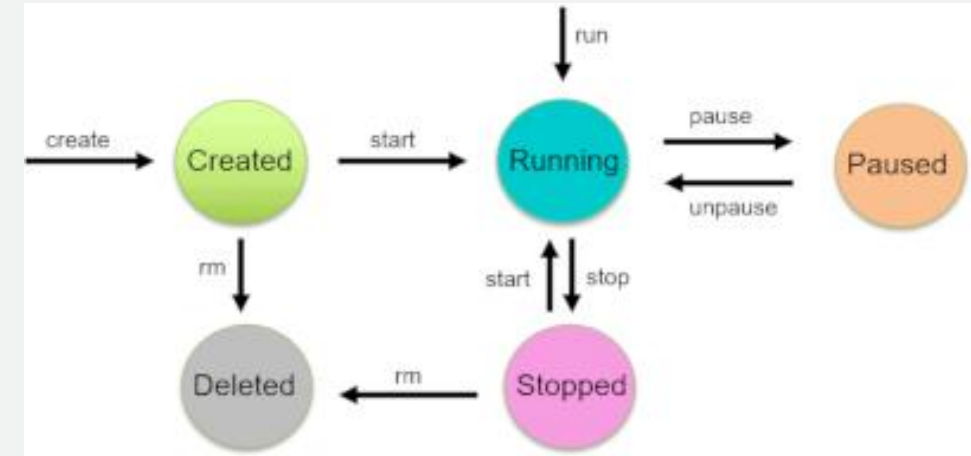
# Cycle de vie d'un conteneur

- **Persistence et gestion des données :**

- Description : Il est parfois nécessaire de conserver des données au-delà de la durée de vie d'un seul conteneur.
- Processus : Docker fournit des volumes pour le stockage et la gestion des données persistantes. Les volumes sont stockés dans une partie du système de fichiers de l'hôte qui est gérée par Docker (/var/lib/docker/volumes/ par défaut).

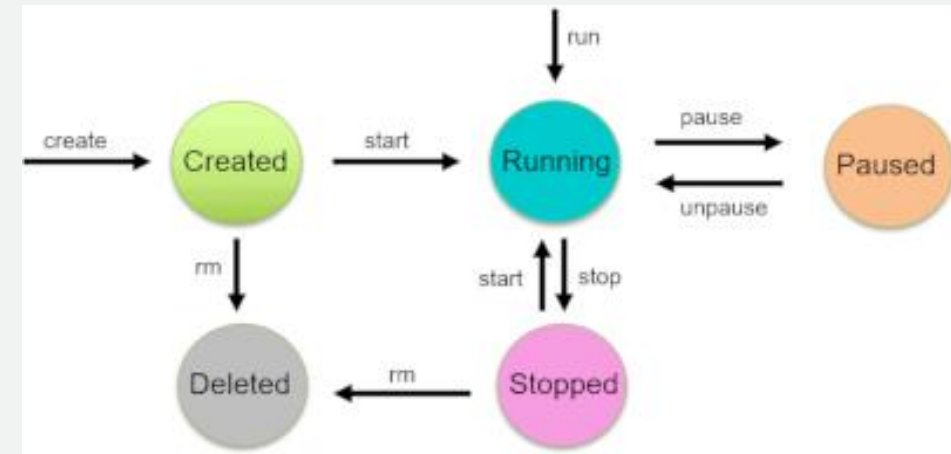
- **Orchestration des conteneurs (facultatif) :**

- Description : Dans les systèmes plus complexes, il peut être nécessaire de gérer le cycle de vie de nombreux conteneurs.
- Processus : C'est là que les outils d'orchestration de conteneurs tels que Kubernetes entrent en jeu. Ils permettent d'automatiser le déploiement, la mise à l'échelle et le fonctionnement des conteneurs.



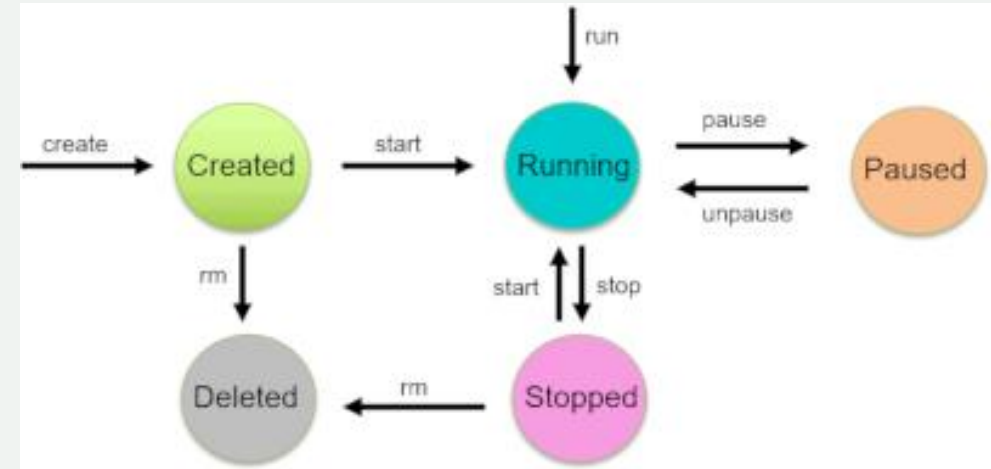
# Commandes associées au cycle de vie

- **docker create :**
  - Fait passer un conteneur de l'état inexistant à l'état créé.
  - Cette commande crée un conteneur mais ne le démarre pas.
- **docker start :**
  - Fait passer un conteneur de l'état *Created* à l'état *Running*.
  - Elle peut également faire passer un conteneur de l'état *Stopped* à l'état *Running*.
- **docker run :**
  - Combinaison de *docker create* et *docker start*, cette commande fait passer un conteneur de l'état non-existant à l'état en cours d'exécution.
- **docker pause :**
  - Fait passer un conteneur de l'état *Running* à l'état *Paused*.
  - Elle suspend tous les processus du conteneur.



# Commandes associées au cycle de vie

- **docker unpause :**
  - Déplace un conteneur de l'état *Paused* à l'état *Running*.
  - Il reprend tous les processus dans le conteneur.
- **docker stop :**
  - Fait passer un conteneur de l'état *Running* à l'état *Stopped*.
  - Il envoie un SIGTERM et, après une période de grâce, un SIGKILL au conteneur, si nécessaire.
- **docker rm :**
  - Lorsqu'il est utilisé sur un conteneur dans l'état *Created*, il supprime le conteneur, le faisant passer à l'état *Deleted*.
  - Lorsqu'il est utilisé sur un conteneur dans l'état *Stopped*, il supprime également le conteneur, le faisant passer à l'état *Deleted*.



# TP: Création d'un conteneur

- La commande run permet de lancer une instance
  - `$docker run ubuntu:latest`
- Si l'image passée n'est pas présente dans le système de fichier, l'image est préalablement chargée
- Il y a plusieurs manières d'exécuter une image
  - En mode attaché: un conteneur est créé et nous donne la main pour interagir avec
    - `$docker run -ti ubuntu:latest`
  - En mode détaché: une fois créé, le conteneur s'exécute comme une tâche de fond
    - `$docker run -d postgres`
- **Exercice:**
  - Exécuter la commande `$docker run -ti ubuntu:latest bash`
  - Lister les conteneurs à partir d'un autre terminal, qu'observez-vous ?
  - Quitter l'invite de commande (Ctrl + D) du conteneur et lister de nouveau les conteneurs, qu'observez-vous ?

# Options d'instanciation de base

- **--name** : Donne un nom au conteneur.
- **-d** ou **--detach** : Exécute le conteneur en arrière-plan (mode détaché).
- **-it** : Combine **-i** (interactif) et **-t** (terminal) pour exécuter le conteneur en mode interactif avec un terminal.
- **--rm** : Supprime automatiquement le conteneur après son arrêt.
- **--memory** ou **-m** : Limite la mémoire utilisable par le conteneur.
- **--cpus** : Limite le nombre de CPU utilisables par le conteneur.
- **--cpu-shares** : Définit la priorité CPU relative du conteneur (valeur par défaut: 1024)
  - Un conteneur avec 2048 shares recevra deux fois plus de temps CPU qu'un conteneur avec 1024 shares, en période de forte contention.
  - Un conteneur avec 512 shares recevra moitié moins de temps CPU qu'un conteneur avec 1024 shares, dans les mêmes conditions.
- **-e** ou **--env** : Définit une variable d'environnement.
- **--env-file** : Charge des variables d'environnement depuis un fichier.

# TP: Cycle de vie d'un conteneur

- **Exercice:**
  - Instancier une image ubuntu en mode interactif
  - Créer un fichier à la racine du conteneur et quitter le conteneur (Ctrl + D)
  - Réinstancier une image ubuntu en mode interactif et lister les fichiers. Qu'observez-vous ?
- Le fait de stopper un conteneur ne supprime pas les fichiers ajoutés, il est juste endormi
  - La commande "`$docker ps`" liste uniquement les processus actifs
  - Il faut ajouter l'option "-a" pour lister les processus endormis également
- Pour relancer un processus endormi, il suffit de lancer la commande `$docker start mon_conteneur`
- **Exercice:**
  - Exécuter cette dernière commande et lister les processus actifs. Qu'observez-vous ?
  - Pour quitter en gardant le conteneur en tâche de fond **Ctrl + P + Q**
  - Pour reprendre la main sur un conteneur passé en tâche de fond: `$docker attach mon_conteneur`

# TP: Cycle de vie d'un conteneur

- Chaque conteneur gère son cycle de vie
- Une fois le conteneur supprimé, tout ce qui peut être lié à son état (fichiers, processus, ...) disparaît avec
- Afin de persister l'état d'un conteneur plusieurs options sont à considérer:
  - Dans le cas d'installations supplémentaires, il est possible de créer une image à partir d'un conteneur
  - En cas de fichiers persistés, il est possible de monter un volume indépendant du conteneur
    - En lecture / écriture
- Afin de créer une image à partir d'un conteneur, docker fournit la commande "commit"
  - Qui peut être vue comme l'opération inverse de la commande "run"
  - `$docker commit container-id #`renvoie l'identifiant de l'image créée à partir du conteneur
  - `$docker tag image-id image-name #`associe un nom à cette nouvelle image
- **Exercice:** créer une nouvelle image à partir d'un conteneur existant et l'instancier

# TP: Cycle de vie d'un conteneur

- Afin de lancer un conteneur en mode détaché, il suffit d'ajouter l'option "-d" à la commande run
  - `$docker run -d -ti ubuntu:latest`
- Lorsqu'un conteneur est créé, il exécute un processus principal (main)
  - `$docker run --rm -ti ubuntu:latest sleep 5`
- Il est possible d'ajouter de nouveaux processus au processus principal par la commande "exec"
  - `$docker exec -ti container-name bash`
  - Pratique pour le debug par exemple
- Pour lister les processus au sein d'un conteneur:
  - `$docker top container-name`
- **Exercice:**
  - Lancer un conteneur avec un bash
  - Associer un second bash
  - Lister les processus actifs du conteneur



# TP: Gestion des problèmes

- Chaque conteneur conserve l'historique de ses logs
- La commande "`$docker logs container-name`" permet d'afficher les logs associés à un conteneur
- **Exercice:**
  - Exécuter un conteneur avec une mauvaise commande
  - Observer les logs suite à l'exécution de la commande
- Attention: les logs sont conservés pour tout le cycle de vie du conteneur
- Options de logging et de santé:
  - **--log-driver** : Spécifie le pilote de logging (par exemple, json-file, syslog, journald).
  - **--log-opt** : Configure des options supplémentaires pour le logging.
  - **--health-cmd** : Définit une commande pour vérifier la santé du conteneur.
  - **--health-interval** : Définit l'intervalle entre les vérifications de santé.
  - **--restart** : Définit la politique de redémarrage du conteneur (no, on-failure, always, unless-stopped).

# Options de restart

- **no (par défaut) :**

- Description : C'est le comportement par défaut. Le conteneur ne sera pas redémarré automatiquement en cas d'arrêt, qu'il soit dû à une erreur, à une sortie volontaire (via docker stop), ou à un redémarrage du système Docker.
- Utilisation : Convient aux conteneurs qui ne doivent pas être relancés automatiquement, comme des tâches ponctuelles ou des outils de ligne de commande.

- **on-failure :**

- Description : Le conteneur sera redémarré automatiquement uniquement si son processus principal se termine avec un code de sortie différent de zéro (indiquant généralement une erreur). Vous pouvez éventuellement spécifier un nombre maximal de tentatives de redémarrage après lesquelles le conteneur ne sera plus redémarré (par exemple, on-failure:5). Si aucun nombre n'est spécifié, Docker tentera de redémarrer indéfiniment.
- Utilisation : Utile pour les services qui peuvent parfois rencontrer des erreurs temporaires et devraient être redémarrés pour tenter de se rétablir (par exemple, une application qui dépend d'une ressource externe temporairement indisponible).

# Options de restart

- **always :**

- Description : Le conteneur sera toujours redémarré automatiquement. Cela inclut les cas d'arrêt dus à une erreur, à une sortie volontaire (via docker stop), ou à un redémarrage du système Docker. La seule façon d'empêcher un conteneur configuré avec always de redémarrer est de le supprimer explicitement (docker rm -f).
- Utilisation : Convient aux services critiques qui doivent être disponibles en permanence et se rétablir automatiquement en cas de problème (par exemple, une base de données, un serveur web).

- **unless-stopped :**

- Description : Le conteneur sera redémarré automatiquement dans tous les cas d'arrêt (erreur, redémarrage du système Docker), sauf s'il a été explicitement arrêté par l'utilisateur (via docker stop). Si le conteneur était arrêté manuellement, il ne sera pas redémarré automatiquement, même après un redémarrage du système Docker.
- Utilisation : C'est souvent le choix le plus judicieux pour de nombreux services. Il assure la reprise automatique en cas de pannes inattendues ou de redémarrage du système, tout en permettant un arrêt manuel persistant lorsque nécessaire (par exemple, pour une maintenance planifiée).

# Exemple avancé d'une instantiation

```
docker run -ti -d \  
--name nginx-container \  
--restart unless-stopped \  
--health-cmd="curl -f http://localhost/ || exit 1" \  
--health-interval=30s \  
--health-timeout=10s \  
--health-retries=3 \  
--health-start-period=5s \  
--log-driver=json-file \  
--log-opt max-size=10m \  
--log-opt max-file=3 \  
nginx
```

**Exercice:** Expliquer ce que fait cette commande

# Autres Commandes Utiles

- `$ docker container ls -as` => toutes les informations concernant les conteneurs
- `$ docker container stats` => statistiques générales
- `$ docker container inspect container_id` => Informations détaillées sur un conteneur
- `$ docker container rename container_name new_name`
- `$ docker container prune` => supprime tous les conteneurs stoppés

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of circles in shades of pink, purple, and orange. The top-right corner has circles in blue, orange, and purple. The bottom-right corner is decorated with circles in teal, pink, and purple. The main title is centered in a dark purple font.

# Gestion des images

## Partie 2

# Cycle de vie d'une image

- **Création d'images :**

- Description : Une image de conteneur est un paquet exécutable léger et autonome qui comprend tout ce qui est nécessaire à l'exécution d'un logiciel, y compris le code, le moteur d'exécution, les outils système, les bibliothèques et les paramètres.
- Processus : Les images sont généralement construites à partir d'un fichier Docker, qui est un script composé de diverses commandes pour créer l'image.

```
FROM python:3.8-slim-buster
LABEL description="My simple server"

WORKDIR /python-docker
COPY . .
RUN pip3 install -r requirements.txt
CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0"]
```

- **Distribution de l'image :**

- Description : Une fois qu'une image est créée, elle peut être stockée et distribuée via un registre de conteneurs, un système de stockage et de distribution de contenu qui contient des images Docker nommées, disponibles dans différentes versions étiquetées.
- Processus : Les registres les plus courants sont Docker Hub, Google Container Registry et Amazon Elastic Container Registry.

# Travaux Pratiques

- Les commandes `images` / `image` permettent de gérer les images
- La commande `"$docker pull"` permet de tirer une image sans l'exécuter
- La commande `"$docker push"` permet de partager de nouvelles images dans un registre central
- La commande `"$docker rmi image-name:version"` permet de supprimer une image
- La commande `"$docker image prune"` permet de supprimer toutes les images non utilisées
- **Exercice:**
  - A partir de l'image `ubuntu`, créer 1 nouvelle image
  - Lister les images avec la commande `"$docker images"`
  - Que peut-on observer sur la taille de l'image créée ?



# Provenance des images

- La plupart des images proviennent de registres qui centralisent et distribuent les images
  - Par défaut, docker est connecté au registre docker-hub
- Vous pouvez utiliser des registres publics ou privés
- Docker fournit les commandes pour requêter les registres
  - `$docker search image-name`
- **Exercice:** Faites une recherche sur redis. Qu'observez-vous ?
- Il est souvent nécessaire de se rendre sur le site, car vous disposez en plus d'instructions qui permettent d'instancier correctement une image
- Pour se connecter à un autre registre, vous pouvez utiliser la commande "`$docker login url`"
- Pour créer son propre registre, il faut installer et configurer docker-registry
- Autres options:
  - Docker Trusted Registry, Amazon ECR, Google CR, Azure CR
  - Docker met également à disposition les commandes `save` et `load` pour archiver / charger des images localement

# Dockerfile

- Docker vous fournit la possibilité de créer des images docker custom
- La configuration d'une image docker est spécifiée dans un fichier **Dockerfile**
- Un fichier Dockerfile représente en quelque sorte un programme qui spécifie comment une image doit être construite par des commandes spécifiques
- Lorsqu'un fichier Dockerfile a été complètement spécifié, la commande "\$docker build" peut être appelée pour générer l'image correspondante
  - Exemple: \$docker build -t tag-name . #le point désigne le chemin où se trouve le Dockerfile
- Il y a des conventions à respecter lorsque l'on tag une image
  - Registry-url/organization/image-name:version-tag
- Les images sont construites en couches. Chaque ligne produit une nouvelle image.
- Syntaxe: <https://docs.docker.com/engine/reference/builder/>

# Commandes Dockerfile

- **FROM** (Obligatoire - Première instruction non commentée)
  - Syntaxe : `FROM <image>[:<tag>] [AS <nom_d_étape>]`
  - Rôle : Spécifie l'image de base (parent) à partir de laquelle votre nouvelle image sera construite. C'est le point de départ de votre Dockerfile.
  - Exemple :
    - `FROM ubuntu:latest` (utilise la dernière version de l'image Ubuntu)
    - `FROM node:16-alpine AS builder` (utilise Node.js version 16 basé sur Alpine Linux et nomme cette étape "builder" pour les builds multi-étapes)
- **RUN**
  - Syntaxe:
    - Forme shell : `RUN <commande>` (la commande est exécutée dans un shell, généralement `/bin/sh` sous Linux)
    - Forme exec : `RUN ["executable", "param1", "param2"]` (évite l'invocation d'un shell, plus explicite et potentiellement plus sécurisée)
  - Rôle : Exécute des commandes dans le contexte de l'image en cours de construction. C'est l'instruction principale pour installer des logiciels, configurer l'environnement, etc. Chaque commande RUN crée une nouvelle couche dans l'image.
  - Exemples :
    - `RUN apt-get update && apt-get install -y --no-install-recommends curl wget` (forme shell)
    - `RUN ["npm", "install", "--production"]` (forme exec)

# Commandes Dockerfile

## • COPY

- Syntaxe : `COPY [--chown=<user>:<group>] <src>... <dest>`
- Rôle : Copie des fichiers ou des dossiers de votre machine hôte (ou du contexte de build) vers le système de fichiers de l'image à une destination spécifiée.
- `<src>` : Chemin(s) vers le fichier ou le dossier source. Peut accepter des motifs globaux (wildcards).
- `<dest>` : Chemin de destination à l'intérieur de l'image. Doit être un chemin absolu ou relatif au `WORKDIR`.
- `--chown` (optionnel) : Change la propriété des fichiers/dossiers copiés à l'utilisateur et au groupe spécifiés à l'intérieur de l'image.
- Exemples :
  - `COPY package.json package-lock.json /app/` (copie les fichiers de dépendances dans le répertoire `/app` de l'image)
  - `COPY ./src /app/src/` (copie le contenu du dossier `src` local vers `/app/src` dans l'image)
  - `COPY --chown=nobody:nogroup myapp.jar /opt/myapp.jar`

## • ADD

- Syntaxe : `ADD [--chown=<user>:<group>] <src>... <dest>`
- Rôle : Similaire à `COPY`, mais avec des fonctionnalités supplémentaires :
  - Peut décompresser automatiquement les fichiers tar (gzip, bzip2, xz) lorsqu'ils sont copiés d'une source locale vers un dossier de destination dans l'image.
  - Peut télécharger des fichiers à partir d'URL.
- Exemples :
  - `ADD my-archive.tar.gz /app/` (décompressera `my-archive.tar.gz` dans `/app/`)
  - `ADD https://example.com/myfile.zip /tmp/` (téléchargera le fichier ZIP dans `/tmp/`)

# Commandes Dockerfile

## • WORKDIR

- Syntaxe : WORKDIR <chemin>
- Rôle : Définit le répertoire de travail pour les instructions RUN, CMD, ENTRYPOINT, COPY et ADD qui suivent dans le Dockerfile. Si le répertoire n'existe pas, il est créé. Vous pouvez utiliser plusieurs instructions WORKDIR pour changer de répertoire, et les chemins sont interprétés de manière relative au WORKDIR précédent.
- Exemples :
  - WORKDIR /app (définit /app comme répertoire de travail)
  - WORKDIR src (si le WORKDIR précédent était /app, le nouveau WORKDIR sera /app/src)

## • CMD

- Syntaxe:
  - Forme exec (recommandée) : CMD ["executable", "param1", "param2"] (devient la commande par défaut exécutée lorsque le conteneur démarre)
  - Forme shell : CMD <commande> (la commande est exécutée dans un shell)
- Rôle : Spécifie la commande à exécuter lorsque le conteneur est lancé. Il ne peut y avoir qu'une seule instruction CMD dans un Dockerfile. Si vous en spécifiez plusieurs, seule la dernière sera prise en compte.
- Important : CMD fournit une commande par défaut qui peut être surchargée en spécifiant une autre commande lors de l'exécution du conteneur avec docker run <image> <nouvelle\_commande>.
- Exemples :
  - CMD ["node", "server.js"] (exécute server.js avec Node.js au démarrage du conteneur)
  - CMD ["/bin/bash"] (ouvre un shell Bash dans le conteneur)
  - CMD ["-h"] (fournit l'argument -h à l'instruction ENTRYPOINT)

# Commandes Dockerfile

## ENTRYPOINT

- Syntaxe:
  - Forme exec (recommandée) : `ENTRYPOINT ["executable", "param1", "param2"]` (configure l'exécutable principal qui sera lancé lorsque le conteneur démarre)
  - Forme shell : `ENTRYPOINT <commande>` (la commande est exécutée dans un shell)
- Rôle : Configure l'exécutable qui sera exécuté comme processus PID 1 au démarrage du conteneur. Similaire à `CMD`, mais avec des différences importantes :
  - Les arguments spécifiés dans `docker run <image> <arguments>` sont ajoutés à la fin de la commande `ENTRYPOINT` (en forme exec).
  - `ENTRYPOINT` rend le conteneur plus "exécutable". Il définit le but principal du conteneur.
- Pour surcharger l'`ENTRYPOINT` lors de l'exécution, vous pouvez utiliser l'option `--entrypoint` de `docker run`.
- Combinaison avec `CMD` : Il est courant d'utiliser `ENTRYPOINT` pour définir l'exécutable principal et `CMD` pour fournir les arguments par défaut à cet exécutable.
- Exemples :
  - `ENTRYPOINT ["/app/my-script.sh"]`
  - `ENTRYPOINT ["java", "-jar", "myapp.jar"]`
  - `ENTRYPOINT ["nginx", "-g", "daemon off;"]` (empêche Nginx de se mettre en arrière-plan)
  - `ENTRYPOINT ["my-app"]`
  - `CMD ["--config", "/etc/myapp/config.json"]` (au démarrage, `my-app --config /etc/myapp/config.json` sera exécuté)

# Commandes Dockerfile

## • ENV

- Syntaxe : ENV <clé>=<valeur> ...
- Rôle : Définit des variables d'environnement qui seront disponibles à l'intérieur du conteneur en cours d'exécution.
- Exemples :
  - ENV JAVA\_HOME /usr/lib/jvm/default-java
  - ENV PATH "\$PATH:/usr/local/bin"
  - ENV MY\_API\_KEY=abcdef123

## • EXPOSE

- Syntaxe : EXPOSE <port> [<port>/<protocol>...]
- Rôle : Déclare les ports sur lesquels l'application à l'intérieur du conteneur écoute le réseau. Cette instruction est principalement une documentation et n'expose pas réellement les ports de l'hôte. Pour exposer les ports à l'hôte, vous devez utiliser l'option -p de docker run ou la section ports dans Docker Compose.
- Exemples :
  - EXPOSE 80
  - EXPOSE 8080/tcp 8080/udp

# Commandes Dockerfile

## • VOLUME

- Syntaxe : `VOLUME ["/path/in/container"]`
- Rôle : Déclare un ou plusieurs points de montage de volumes nommés ou anonymes. Les volumes sont utiles pour persister des données au-delà du cycle de vie d'un conteneur et pour partager des données entre conteneurs.
  - L'instruction `VOLUME` déclare simplement l'intention de créer un point de montage pour un volume à cet emplacement dans le conteneur. Le montage réel du volume (où les données seront stockées sur l'hôte) se produit lors de la création ou du démarrage du conteneur (avec `docker run -v` ou dans la section volumes de Docker Compose).
  - Il est généralement considéré comme une bonne pratique de définir les emplacements où votre application stocke des données dynamiques (bases de données, logs, uploads) comme des `VOLUME` dans votre Dockerfile. Cela rend plus clair pour les utilisateurs de votre image quels répertoires sont destinés à la persistance.
  - Les modifications apportées aux données dans un volume ne sont pas incluses dans les mises à jour de l'image.
- Exemples :
  - `VOLUME ["/data"]`
  - `VOLUME /var/log/myapp`

## • USER

- Syntaxe : `USER <utilisateur>[:<groupe>]` ou `USER <UID>[:<GID>]`
- Rôle : Spécifie l'utilisateur (et optionnellement le groupe) sous lequel les commandes `RUN`, `CMD` et `ENTRYPOINT` suivantes seront exécutées à l'intérieur du conteneur. Il est recommandé de ne pas exécuter les processus en tant que `root` pour des raisons de sécurité.
- Exemples :
  - `USER nobody`
  - `USER appuser:appgroup`
  - `USER 1001:1001`



# Commandes Dockerfile

## • ARG

- Syntaxe : ARG <nom>[=<valeur\_par\_défaut>]
- Rôle : Définit une variable que l'utilisateur peut passer à l'image lors de la construction à l'aide de l'option --build-arg de docker build. Les variables ARG sont uniquement disponibles pendant le processus de construction.
- Exemples :
  - ARG VERSION=1.0
  - ARG USERNAME
  - docker build --build-arg USERNAME=myuser -t myimage .

## • STOPSIGNAL

- Syntaxe : STOPSIGNAL <signal>
- Rôle : Spécifie le signal système qui sera envoyé au processus principal du conteneur pour l'arrêter. Par défaut, c'est SIGTERM.
- Exemple :
  - STOPSIGNAL SIGQUIT

# Commandes Dockerfile

## • HEALTHCHECK

- Syntaxe :
  - Forme shell : HEALTHCHECK [OPTIONS] CMD <commande>
  - Forme exec : HEALTHCHECK [OPTIONS] CMD ["executable", "param1"]
- Rôle : Configure une commande à exécuter périodiquement pour vérifier la santé du conteneur. Si la commande renvoie un code de sortie 0, le conteneur est considéré comme sain ; un code non nul indique qu'il est malade.
- OPTIONS :
  - --interval=DURATION (par défaut: 30s)
  - --timeout=DURATION (par défaut: 30s)
  - --start-period=DURATION (délai pour les premières vérifications)
  - --retries=N (nombre de tentatives en cas d'échec avant de déclarer le conteneur malade)
- Exemple :
  - HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost:8080/health

## • SHELL

- Syntaxe : SHELL ["executable", "options"]
- Rôle : Modifie le shell par défaut utilisé pour la forme shell des commandes RUN, CMD et ENTRYPOINT suivantes. Utile principalement pour les systèmes d'exploitation Windows où le shell par défaut est cmd.exe.
- Exemple :
  - SHELL ["powershell", "-Command"]

# Fichier .dockerignore

- Le fichier .dockerignore fonctionne de manière similaire au .gitignore pour git
- Il est placé à la racine du contexte de build et est analysé avant l'envoi des fichiers au démon Docker.
- Tous les fichiers correspondant aux motifs spécifiés sont exclus du contexte de build et ne seront donc pas inclus dans l'image finale.
- Ce fichier est particulièrement utile pour les projets avec de nombreuses dépendances ou des artefacts de build volumineux qui ne sont pas nécessaires dans l'image finale.

# Utilité du fichier .dockerignore

- **Optimisation des performances**

- Réduction de la taille des images : En excluant les fichiers et répertoires non nécessaires (comme `node_modules`, `logs`, etc.), vous obtenez des images plus légères.
- Accélération du processus de build : La réduction du contexte de build envoyé au démon Docker diminue considérablement le temps de construction.
- Transferts plus rapides : Les images plus petites sont plus rapides à télécharger (pull) et à envoyer (push) vers les registres.

- **Sécurité renforcée**

- Prévention des fuites de données sensibles : Il permet d'éviter d'inclure accidentellement des fichiers contenant des informations confidentielles comme des variables d'environnement (`.env`) ou des fichiers de configuration.
- Réduction de la surface d'attaque : Moins de fichiers dans l'image signifie moins de vulnérabilités potentielles.

# Fichier .dockerignore - Exemple

```
# Dépendances
node_modules
npm-debug.log
yarn-debug.log
yarn-error.log
# Environnement et configuration
.env
.env.local
.env.development
.env.test
.env.production
*.env
```

# Travaux Pratiques

- **Exercice:**

- Construire une image à partir du Dockerfile ci-dessous
- Qu'observez-vous sur la sortie standard ?

```
#requirements.txt file
flask==2.2.2
werkzeug==2.2.2
```

```
#app.py file
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return '<h1>Hello from Flask and Docker</h1>'

if __name__ == "__main__":
    app.run(debug=True)
```

```
#Dockerfile file
FROM python:3.8-slim-buster
LABEL description="My simple server"

WORKDIR /python-docker
COPY . .
RUN pip3 install -r requirements.txt
EXPOSE 5000
CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0", "--port=5000"]
```

# Build multi-étapes

- Le build multi-étapes (multi-stage build) est une technique puissante de Docker qui permet de créer des images plus légères et plus sécurisées en séparant le processus de construction en plusieurs étapes.
- Le build multi-étapes utilise plusieurs instructions FROM dans un même Dockerfile, où chaque instruction commence une nouvelle étape. Vous pouvez sélectivement copier des artefacts d'une étape à l'autre, en laissant derrière tout ce qui n'est pas nécessaire dans l'image finale.
- Avantages principaux
  - **Images plus légères** : L'image finale ne contient que les artefacts nécessaires à l'exécution
  - **Sécurité améliorée** : Réduction de la surface d'attaque en éliminant les outils de build et dépendances de développement
  - **Séparation des préoccupations** : Distinction claire entre environnement de build et environnement d'exécution
  - **Optimisation du cache** : Meilleure utilisation du cache Docker pour accélérer les builds

# Build multi-étapes - Exemple

```
# Étape de build
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
# Étape de production
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```



# Importer / Exporter des images

- Dans le cas où vous n'auriez pas accès à un registre (réseaux fermés par exemple), il est tout à fait possible de créer une archive d'une image que l'on peut copier sur différents supports de stockage
  - `docker save nginx:alpine | gzip -9 > custom-nginx.tar.gz`
- Il est facile ensuite de désarchiver cette image pour la rendre accessible sur une quelconque machine avec Docker installé
  - `gunzip -c custom-nginx.tar.gz | docker load`

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, a medium purple circle, and a tiny dark purple dot. The top-right corner has a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner is decorated with a small teal circle, a medium pink circle, a tiny dark purple dot, a medium purple circle, a large pink circle, and a large teal circle. The text 'Gestion du réseau' is centered in a dark purple, sans-serif font.

# Gestion du réseau

# Présentation

- Docker propose plusieurs types de réseaux pour connecter les conteneurs entre eux, avec l'hôte ou avec l'extérieur. Ces réseaux permettent de gérer la communication et l'isolation des conteneurs de manière flexible.
- Voici en résumé, les différents types de réseau et leur utilité:
  - **Bridge** : Par défaut pour la plupart des cas.
  - **Host** : Pour des performances maximales (sans isolation).
  - **Overlay** : Pour des applications distribuées sur plusieurs hôtes.
  - **Macvlan/IPvlan** : Pour des conteneurs nécessitant une présence réseau directe.
  - **None** : Pour des conteneurs sans besoin de réseau.
  - **User-defined bridge** : Pour isoler des groupes de conteneurs.

# Configuration réseau

- **Réseau bridge** (par défaut)

- Description : Le réseau bridge est le réseau par défaut utilisé par Docker. Il permet aux conteneurs de communiquer entre eux sur le même hôte Docker.
- Fonctionnement :
  - Docker crée un pont réseau virtuel (docker0) sur l'hôte.
  - Chaque conteneur connecté à ce réseau reçoit une adresse IP interne.
  - Les conteneurs peuvent communiquer entre eux via ces adresses IP.
- Utilisation :
  - Idéal pour les applications mono-hôte.
  - Les conteneurs ne sont pas accessibles depuis l'extérieur par défaut (sauf via le mapping de ports).

- **Réseau host**

- Description : Le réseau host supprime l'isolation réseau entre le conteneur et l'hôte Docker. Le conteneur utilise directement les interfaces réseau de l'hôte.
- Fonctionnement :
  - Le conteneur partage l'espace de réseau de l'hôte.
  - Aucune adresse IP distincte n'est attribuée au conteneur.
- Utilisation :
  - Idéal pour des performances réseau maximales (pas de NAT ni de pont).
  - Non recommandé pour des environnements multi-conteneurs, car il n'y a pas d'isolation.

# Configuration Réseau

- **Réseau overlay**

- Description : Le réseau overlay permet de connecter des conteneurs sur plusieurs hôtes Docker, ce qui est essentiel pour les clusters (par exemple, avec Docker Swarm ou Kubernetes).
- Fonctionnement :
  - Utilise un réseau virtuel qui s'étend sur plusieurs hôtes.
  - Les conteneurs peuvent communiquer entre eux comme s'ils étaient sur le même hôte.
- Utilisation :
  - Idéal pour les applications distribuées ou les environnements de production multi-hôtes.

- **Réseau macvlan**

- Description : Le réseau macvlan permet d'attribuer une adresse MAC unique à chaque conteneur, ce qui le fait apparaître comme un appareil physique sur le réseau.
- Fonctionnement :
  - Le conteneur est directement connecté au réseau physique de l'hôte.
  - Chaque conteneur a sa propre adresse IP sur le réseau local.
- Utilisation :
  - Idéal pour les applications nécessitant une présence réseau directe (par exemple, des serveurs réseau virtuels).

# Configuration Réseau

- **Réseau none**

- Description : Le réseau none désactive entièrement la pile réseau du conteneur. Le conteneur n'a accès à aucune interface réseau.
- Fonctionnement :
  - Le conteneur est complètement isolé du réseau.
  - Utile pour des conteneurs qui n'ont pas besoin de communication réseau.
- Utilisation :
  - Idéal pour des conteneurs qui effectuent des tâches locales (par exemple, des traitements par lots).

- **Réseau personnalisé (user-defined bridge)**

- Description : Docker permet de créer des réseaux bridge personnalisés pour isoler des groupes de conteneurs.
- Fonctionnement :
  - Similaire au réseau bridge par défaut, mais avec une isolation supplémentaire.
  - Les conteneurs sur un réseau personnalisé peuvent communiquer entre eux, mais pas avec les conteneurs sur d'autres réseaux.
- Utilisation :
  - Idéal pour segmenter des applications ou des environnements (par exemple, développement, test, production).

# Configuration Réseau

- **Réseau IPvlan**

- Description : Le réseau IPvlan est similaire à macvlan, mais permet à plusieurs conteneurs de partager la même adresse MAC tout en ayant des adresses IP différentes.
- Fonctionnement :
  - Les conteneurs sont connectés au réseau physique via une interface parente.
  - Chaque conteneur a sa propre adresse IP, mais partage la même adresse MAC.
- Utilisation :
  - Idéal pour les environnements où les adresses MAC uniques sont limitées.

# Configuration Réseau

- Exemples:

- `$ docker run -d --name conteneur1 nginx #bridge`
- `$ docker run -d --network host nginx #host`
- `$ docker network create --driver overlay mon-reseau-overlay #overlay`
- `$ docker service create --network mon-reseau-overlay --name mon-service nginx`
- `$ docker network create -d macvlan --subnet=192.168.1.0/24 --gateway=192.168.1.1 -o parent=eth0 mon-reseau-macvlan #macvlan`
- `$ docker run --network mon-reseau-macvlan --ip 192.168.1.100 nginx`
- `$ docker run -d --network none mon-conteneur-sans-reseau #none`
- `$ docker network create mon-reseau-bridge #user bridge`
- `$ docker run -d --network mon-reseau-bridge --name conteneur1 nginx`



# Commandes usuelles

Commande	Description
<b>docker network ls</b>	Liste tous les réseaux Docker disponibles
<b>docker network create</b>	Crée un nouveau réseau Docker
<b>docker network create -d &lt;driver_type&gt; &lt;network_name&gt;</b>	Crée un réseau avec un driver spécifique
<b>docker network connect &lt;network_name&gt; &lt;container_id&gt;</b>	Connecte un conteneur à un réseau existant
<b>docker network disconnect &lt;network_name&gt; &lt;container_id&gt;</b>	Déconnecte un conteneur d'un réseau
<b>docker network inspect &lt;network_name&gt;</b>	Affiche des informations détaillées sur un ou plusieurs réseaux
<b>docker network rm &lt;network_name&gt;</b>	Supprime un ou plusieurs réseaux
<b>docker network prune</b>	Supprime tous les réseaux non utilisés

# Travaux Pratiques

- Les conteneurs ont été conçus pour fonctionner ensemble dans un réseau virtuel, créé par docker
- Un conteneur peut exposer des ports pour:
  - Communiquer avec d'autres conteneurs
  - Être accessible depuis la machine hôte
- Exemple:
  - `docker run -d -e POSTGRES_USER=training -e POSTGRES_PASSWORD=training -e POSTGRES_DB=training --name mydb -p 6543:5432 postgres`
  - `psql -U training -h localhost -p 6543 -W training #en local`
- Cet exemple illustre au passage la configuration de variables d'environnement

# Travaux Pratiques

- La commande "\$docker network" permet de gérer les réseaux
- \$docker network ls #liste les réseaux
- \$docker network create training #crée un réseau training
- \$docker run --rm -ti -d --name m1 --net training my-ubuntu bash #crée un conteneur m1 sur le réseau training
- \$docker network connect training m2 #ajouter un conteneur existant dans le réseau
- **Exercice:**
  - Lancer deux conteneurs ubuntu sur le même réseau
  - Installer sur l'un des deux conteneurs la commande ping
  - \$apt update && apt install iputils-ping
  - Pinger l'autre machine
  - Qu'observez-vous ?
- L'option --link permet d'établir des relations à sens unique, elle permet d'accéder aux variables d'environnement du conteneur ciblé

# Travaux Pratiques

- Par défaut, les ports sont exposés en TCP
- Pour les exposer en UDP, il faut utiliser la syntaxe suivante: `$docker run -p 1234:1234/udp`
- Lire attentivement la documentation de chaque image avant d'en lancer des instances
- **Exercice:**
  - Exécuter la commande pour lancer un serveur postgresql puis lister les conteneurs.
  - Utiliser un client postgres pour vous y connecter.
  - Lancer une instance d'une image Ubuntu et installez-y un client postgres
  - Utiliser le client pour vous connecter à la même base.

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, a medium purple circle, and a tiny dark purple dot. The top-right corner has a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner includes a small teal circle, a medium pink circle, a tiny dark purple dot, a medium purple circle, a large pink circle, and a large teal circle.

# Gestion des Volumes

# Gestion des volumes

- Comme mentionné précédemment, docker permet le partage de données entre conteneur à travers des espaces de stockage virtuels: les volumes
- Les volumes sont un élément essentiel de Docker lorsqu'il s'agit de gérer des données persistantes.
  - Ils permettent de séparer le cycle de vie des données du cycle de vie des conteneurs, ce qui est une bonne pratique pour les applications conteneurisées.
  - En utilisant des volumes, vous vous assurez que vos données restent intactes même lorsque les conteneurs sont arrêtés, supprimés ou reconstruits.
- La sauvegarde des données est très importante et il faut être attentif au pilote de volume et aux autorisations de fichiers afin d'éviter toute perte de données ou tout problème d'accès.
- Grâce aux commandes de gestion des volumes, vous pouvez gérer efficacement les données dans votre environnement Docker.

# Commandes usuelles

<b>docker volume create</b>	Crée un volume.
<b>docker volume ls</b>	Liste les volumes.
<b>docker volume inspect</b>	Affiche les détails d'un volume.
<b>docker volume rm</b>	Supprime un volume.
<b>docker volume prune</b>	Supprime tous les volumes inutilisés.
<b>docker run -v</b>	Monte un volume ou un répertoire dans un conteneur.
<b>docker volume create --opt</b>	Crée un volume avec des options spécifiques.
<b>docker inspect &lt;conteneur&gt;</b>	Affiche les volumes utilisés par un conteneur.

# Gestion des volumes

- Il existe deux types de volumes
  - Les volumes persistants: existent indépendamment des conteneurs
  - Les volumes éphémères: existent tant qu'un conteneur continue de les pointer
- Il est également possible de monter des dossiers / fichiers permettant un partage entre l'hôte et les conteneurs
  - `$docker run --rm -ti -v /tmp/myfolder:/myfolder[:ro] --name m1 my-ubuntu bash`
  - Ajouter :ro dans le cas d'un montage en lecture seule
- **Exercice:**
  - Tester la commande précédente
  - Editer un nouveau fichier dans le dossier partagé et quitter le conteneur
  - Observer sur l'hôte le contenu du dossier partagé
- Le partage d'un fichier se fait exactement de la même manière
- **Question:** quel intérêt voyez-vous au partage de fichier ?



# Volumes éphémères

- Un volume éphémère est créé si aucun mapping n'est spécifié
- Le partage entre conteneurs se fait par l'option "--volume-from"
- Exemple:
  - `$docker run --rm -ti -v /myfolder -d --name m1 my-ubuntu bash`
  - `$docker run --rm -ti --volumes-from m1 --name m2 my-ubuntu bash`
- Commandes essentielles
  - `$docker volume create mon_volume` # crée un volume "mon\_volume"
  - `$docker volume ls` # liste les volumes
  - `$docker volume inspect mon_volume` # donne des informations détaillées sur le volume "mon\_volume"
  - `$docker volume rm mon_volume` # supprime le volume "mon\_volume"
- Instancier une image avec un volume:
  - `$docker run -d -v mon_volume:/chemin/complet ubuntu:latest`

# Exercice

- Compiler le projet React-2048
  - Vous aurez besoin de l'outil npm => Utiliser l'image node:20-alpine pour la compilation
  - \$npm ci
  - \$npm run build
  - Cela va nécessiter un partage de volume
- À l'aide d'un Dockerfile, spécifier une image docker qui embarque le résultat de la compilation
- Lancer une instance de l'image (conteneur) qui écoute sur le port local 80
  - Cela va nécessiter un mapping de ports
- Observez le résultat
- Observez les logs
- Monitorisez l'état de santé du conteneur

# Docker cp

- La commande **docker cp** permet de copier des fichiers ou des répertoires entre un conteneur Docker et le système hôte.
- Elle fonctionne de manière similaire à la commande Unix cp, mais avec quelques différences et limitations.
- **Syntaxe** : `docker cp <src> <dest>`
- **Exemples**:
  - `docker cp container_name:/chemin/dans/conteneur /chemin/sur/hote`
  - `docker cp /chemin/sur/hote container_name:/chemin/dans/conteneur`



# Docker cp

- Fonctionne avec des conteneurs en cours d'exécution ou arrêtés
- Permet de copier et renommer simultanément
- Préserve les permissions des fichiers quand c'est possible
- Ne suit pas les liens symboliques par défaut (utiliser l'option -L pour cela)
- Ne permet pas de copier directement entre deux conteneurs (il faut passer par l'hôte)
- **Question:** Quels usages voyez-vous à cette commande ?

The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of pink, purple, and orange circles. The top-right corner has blue, purple, and orange circles. The bottom-right corner is decorated with pink, purple, and teal circles. The title 'Composition de Services' is centered in a dark blue, sans-serif font.

# Composition de Services

# Docker-compose

- Configurer un réseau de conteneurs à la main peut s'avérer chronophage et sujet à erreur
- Afin de faciliter cette tâche, docker fournit la commande docker-compose qui prend en entrée un fichier de configuration et qui permet d'orchestrer l'ensemble des conteneurs
- Docker-compose peut être vu comme une surcouche à docker pour faciliter l'orchestration
- Le fichier de configuration est écrit en Yaml et doit respecter un certain schéma
- **Docker-compose n'a pas été conçu être utilisé en environnement de production**
  - Il n'a été conçu que pour des tests en local
- Il existe d'autres solutions pour les environnements de production distribués
  - Docker-swarm
  - **Kubernetes**

# Docker-compose

- Docker-compose est un outil supplémentaire qui doit être installé à part
  - <https://docs.docker.com/compose/install/>
- La syntaxe est très simple à comprendre quand les principes de docker sont bien assimilés
  - <https://docs.docker.com/compose/compose-file/>

version: "3"

volumes:  
authdata:

services:  
ldap-server:  
image: rg.fr-par.scw.cloud/ecollab/ldap:latest  
build: ./images/ldap  
container\_name: ldap-server  
restart: unless-stopped  
environment:

- LDAP\_ORGANISATION=cenotelie
- LDAP\_DOMAIN=cenotelie.org
- LDAP\_ADMIN\_PASSWORD=eCollab

keycloak-postgres:  
image: postgres:alpine  
container\_name: keycloak-postgres  
restart: unless-stopped  
environment:

- POSTGRES\_USER=keycloak
- POSTGRES\_PASSWORD=keycloak
- POSTGRES\_DB=keycloak

volumes:

- authdata:/var/lib/postgresql/data

keycloak:  
image: rg.fr-par.scw.cloud/ecollab/keycloak:latest  
build: ./images/keycloak  
container\_name: keycloak  
environment:

- DB\_VENDOR=POSTGRES
- DB\_ADDR=keycloak-postgres
- DB\_DATABASE=keycloak
- DB\_USER=keycloak
- DB\_SCHEMA=public
- DB\_PASSWORD=keycloak
- PROXY\_ADDRESS\_FORWARDING=true

restart: unless-stopped  
depends\_on:

- nginx
- keycloak-postgres
- ldap-server

# Exercice

- Instancier un serveur wordpress
  - Tenir compte du fait qu'un serveur wordpress requiert un serveur de bases de données MySQL
  - Le faire manuellement dans un premier temps
  - L'automatiser avec un fichier docker-compose.yml dans un second temps
- Compléter le fichier docker-compose.yml existant pour y intégrer les conteneurs flask et nginx personnalisés
- Observer le contenu du volume db-data
- Lister les différents services avec la commande docker-compose
- Arrêter le service nginx avec la commande docker-compose



The slide features a light gray background with decorative elements in the corners. The top-left corner contains a cluster of pink, purple, and orange circles. The top-right corner has blue, purple, and orange circles. The bottom-right corner is decorated with pink, purple, and teal circles.

# Gestion de la Sécurité

# Risques

- **Vulnérabilités des images**

- Les images peuvent contenir des vulnérabilités dans les bibliothèques ou dépendances intégrées
- Les images malveillantes peuvent être déguisées en images légitimes pour tromper les administrateurs
- Les secrets codés en dur dans les images constituent un risque majeur, même si l'image est ancienne ou inutilisée

- **Risques liés aux conteneurs**

- Évasion de conteneur (container breakout) permettant à un attaquant d'accéder au système hôte
- Compromission du système hôte pouvant entraîner l'exécution de code arbitraire
- Propagation de logiciels malveillants entre conteneurs
- Communications non restreintes entre conteneurs facilitant les mouvements latéraux

# Bonnes pratiques

- **Sécurité des images**

- Utiliser uniquement des images officielles provenant de sources fiables
- Analyser régulièrement les images pour détecter les vulnérabilités (avec des outils comme Trivy, Wiz, Clair)
- Éviter d'utiliser le tag "latest" et spécifier plutôt une version d'image précise
- Mettre en place la signature d'images (Docker Content Trust) pour vérifier leur authenticité
- Maintenir un ensemble d'images de base internes ("golden images")

- **Configuration sécurisée**

- Ne pas exécuter les conteneurs en tant que root
- Limiter les privilèges des conteneurs en utilisant le principe du moindre privilège
- Réduire la surface d'attaque en supprimant les packages et services inutiles
- Configurer correctement les paramètres réseau pour éviter les expositions non désirées
- Utiliser des systèmes de fichiers en lecture seule lorsque c'est possible (:ro)

# Bonnes pratiques

- **Gestion des secrets**

- Éviter de stocker des secrets dans les Dockerfiles ou les variables d'environnement
- Utiliser des outils de gestion de secrets comme Docker Secrets (dans le cas de Docker Swarm) ou des solutions tierces (Hashicorp Vault par exemple)
- Scanner les couches des images Docker pour détecter les secrets cachés

- **Contrôle d'accès**

- Implémenter le contrôle d'accès basé sur les rôles (RBAC) pour les registres d'images
- Activer l'authentification multifacteur (MFA) pour l'accès aux registres
- Appliquer des politiques strictes pour les extractions d'images

# Options principales

- **--user** : Exécute le conteneur avec un utilisateur spécifique (UID ou nom).
  - `$docker run --user 1000 nginx`
- **--read-only** : Monte le système de fichiers du conteneur en lecture seule.
  - `$docker run --read-only alpine`
- **--cap-add** et **--cap-drop** : Ajoute ou supprime des capacités Linux au conteneur.
  - `$docker run --cap-add NET_ADMIN alpine`
- **--security-opt** : Configure des options de sécurité spécifiques (par exemple, SELinux).
  - `$docker run --security-opt seccomp=unconfined alpine`
- **--privileged** : Donne au conteneur un accès privilégié à l'hôte (à utiliser avec précaution).
  - `$docker run --privileged alpine`

# Exercice

- Reprendre la définition des services afin de les sécuriser (Dockerfile et docker-compose.yml)
- Quelles sont les différentes failles de ces fichiers ?
- De quels leviers disposez-vous ?
- Tips:
  - Vérifier sur Docker Hub le niveau de vulnérabilité des images
  - Utiliser le digest SHA des images utilisées
  - Utiliser des builds multi-étages
  - Ajouter une vérification de santé pour détecter les états défectueux de l'application

# Docker Rootless

- Utiliser Docker en mode rootless permet d'exécuter des conteneurs sans avoir besoin de privilèges root, ce qui améliore la sécurité en réduisant les risques liés à l'utilisation de privilèges élevés.
- Le démon Docker (dockerd) s'exécute avec des privilèges root. Si un attaquant parvient à exploiter une vulnérabilité dans Docker, il pourrait obtenir un accès root sur le système hôte.
- Les conteneurs exécutés avec des privilèges root peuvent également accéder à des ressources sensibles du système hôte, ce qui augmente les risques en cas de compromission.
- **Prérequis**
  - Docker version 20.10 ou ultérieure : Le mode rootless est officiellement supporté à partir de cette version.
  - Système d'exploitation compatible : Linux avec un noyau récent (5.11 ou ultérieur est recommandé pour une meilleure expérience).
  - User namespaces activés : Vérifiez que les user namespaces sont activés sur votre système.
    - `grep CONFIG_USER_NS /boot/config-$(uname -r)`
  - L'installation est celle mentionnée

# Docker Rootless - Installation

- Installer le mode **rootless**
  - `sudo apt install -y uidmap`
  - `dockerd-rootless-setup tool.sh install`
  - `export DOCKER_HOST=unix:///run/user/1000/docker.sock`
  - `sudo loginctl enable-linger $USER` # pour le maintien des services utilisateurs après déconnexion
- Activer le mode **rootless**
  - `$sudo systemctl stop docker` # arrêt de docker
  - `$systemctl --user start docker` # démarrage en mode user
  - `$systemctl --user enable docker` # démarrage automatique
- Désactiver le mode **rootless** :
  - Opérations inverses
  - Supprimer les répertoires cachés "`~/local/share/docker`" et "`~/docker`"



# Limitations du mode Rootless

- **Performance :**

- Le mode rootless peut être légèrement moins performant en raison de l'utilisation de user namespaces.

- **Fonctionnalités limitées :**

- Certaines fonctionnalités avancées (comme --privileged) ne sont pas disponibles en mode rootless.
- Les réseaux personnalisés (bridge, overlay) peuvent être limités.
- Les ports inférieurs à 1024 ne sont pas accessibles par défaut (utilisez des ports supérieurs).

# Déploiement sur AWS

# Déploiement en production

- Docker et docker-compose permettent de prototyper / tester des configurations localement
- Pour un déploiement en production d'une architecture en micro-services, il n'est absolument pas recommandé de les utiliser
- Nous avons besoin d'un outil de plus haut niveau permettant de gérer des déploiements sur plusieurs nœuds physiques
  - Au regard de toutes les contraintes rencontrées en production
    - Routage
    - Gestion des mots de passes
    - ...
- Amazon ECS fournit toutes ces facilités
- **Question:** quels sont les avantages des conteneurs d'un point de vue Devops ?

# Actions préalables

- Installer le client AWS sur votre machine
  - Exécuter le script "aws-cli-install.sh" si vous êtes sur ubuntu
- Créer un utilisateur avec les droits suivants(Servie IAM):
  - AmazonEC2ContainerRegistryFullAccess
  - AmazonECS\_FullAccess
- Créer une clef d'accès pour le cas d'usage CLI
  - Enregistrer le couple access key / secret access key

# Registre privé - Service ECR

- Créer un registre docker privé "docker-training"
  - Laisser les paramètres par défaut
  - Enregistrer l'URI du registre (en enlevant la dernière composante) dans la variable d'environnement `DOCKER_REGISTRY`
- Créer l'image docker à pousser en respectant la convention de nommage suivante:
  - `DOCKER_REGISTRY_URI/docker-training:APP_VERSION`
- Lancer la commande "aws config" et entrer les informations demandées
  - Pour la région, entrer la région du cluster (exple: eu-west-3)
- Loguer vous sur votre registre
  - `aws ecr get-login-password | docker login --username AWS --password-stdin $DOCKER_REGISTRY`
- Pousser votre image
  - `docker push $DOCKER_REGISTRY/docker-training:APP_VERSION`
- Observer le résultat sur l'interface web AWS

# Instanciación – Service ECS

- Créer un cluster "ProdCluster"
  - Sélectionner Fargate comme infrastructure
  - Laisser les paramètres par défaut
- Créer ensuite une tâche
  - Sélectionner "Create new task definition with JSON"
  - Adapter le fichier taskdef.json et copier / coller le contenu dans l'interface de spécification de la tâche"
- Créer un service
  - Retourner ensuite sur le cluster
  - Dans "Deployment Configuration", sélectionner la famille et la révision
  - Donner un nom au service
  - Laisser les autres paramètres par défaut
  - Exécuter et observer le résultat

The slide features a light blue background with decorative elements in the corners. The top-left corner contains a large pink circle, a small orange circle, a small light purple circle, a medium purple circle, and a tiny dark purple dot. The top-right corner has a large blue circle, a small orange circle, and a medium purple circle. The bottom-right corner is decorated with a small teal circle, a medium pink circle, a tiny dark purple dot, a medium purple circle, a large pink circle, and a large teal circle.

# Considérations transverses

# Les tests

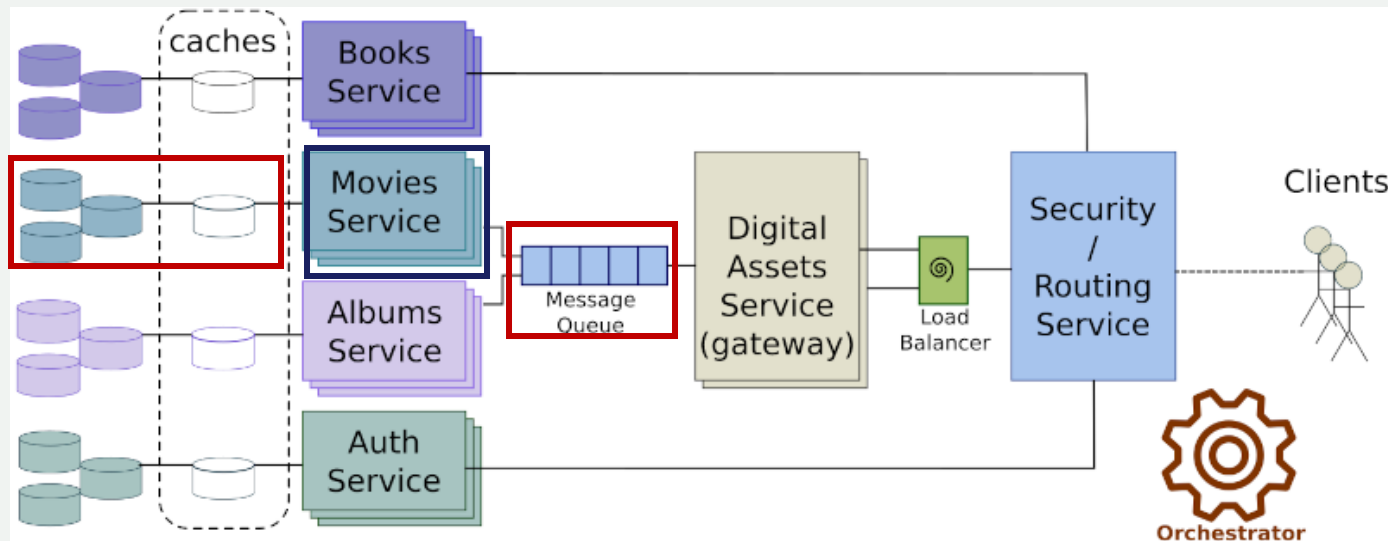
- L'avantage des micro-services, comparés aux applications monolithiques, est indéniable
- Cependant, l'intégration des micro-services peuvent poser des problèmes à cause des comportements émergents
  - Même avec l'adoption d'interfaces standards (ex: HTTP + REST)
- Sur un plan théorique, il est quasiment impossible de démontrer la correction fonctionnelle d'une application
  - Ni même ses performances
- Le moyen le plus efficace dont nous disposons pour tenter de maîtriser les comportements reste les tests et le monitoring
  - Tests unitaires
  - Tests d'intégration
  - Tests E2E
  - Tests de performance

} Suppose de disposer d'une chaîne d'intégration / déploiement continue
- De ce fait, il est impératif que le déploiement d'une application basée sur une architecture en micro-services dispose d'un système de logs et d'exploitation des logs (monitoring) efficace



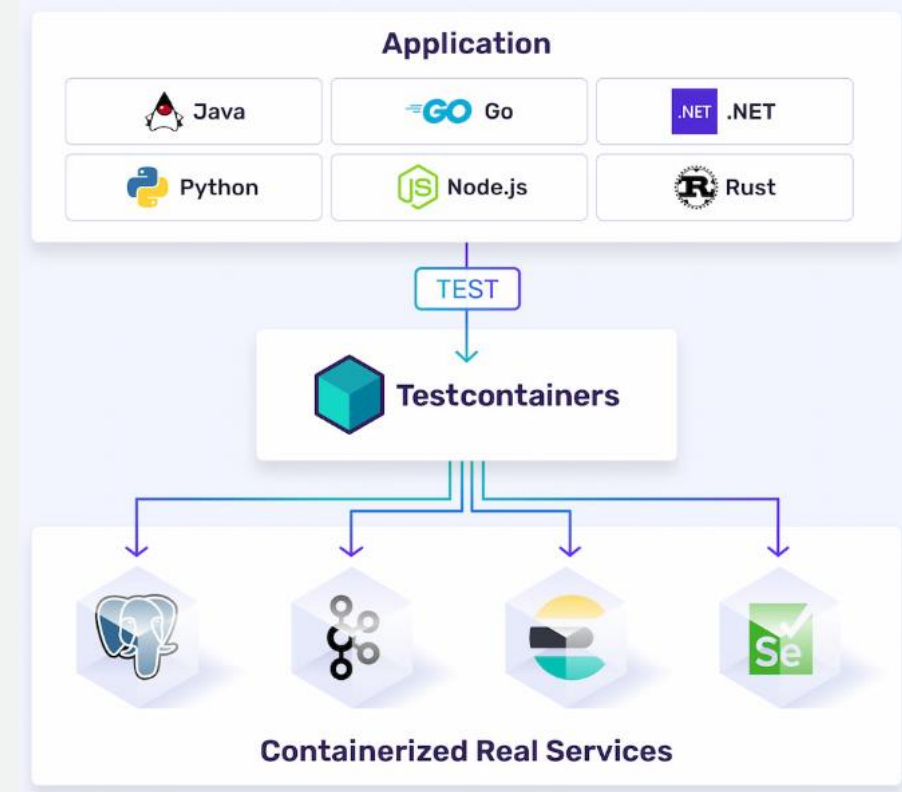
# Les tests

- Au sein du micro-service, il peut être intéressant de mimer l'environnement d'un service – Pourquoi ?
  - Pour cela, on peut utiliser un framework comme Mockito
- Entre les micro-services, il peut être intéressant de mimer l'environnement du micro-service – Pourquoi ?
  - Pour cela, on peut utiliser un framework comme TestContainers



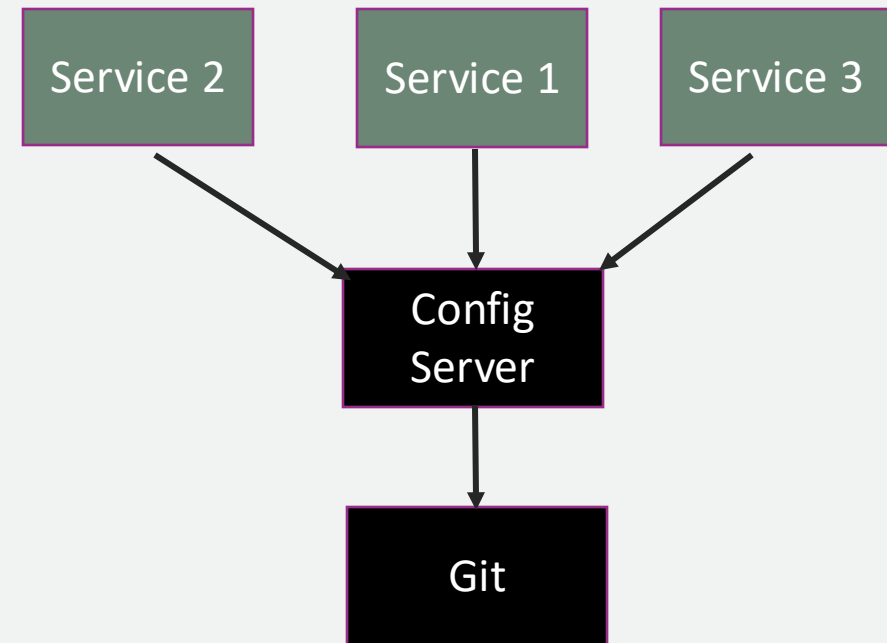
# TestContainers

- Dans les architectures micro-services, où les services interagissent souvent avec divers systèmes externes (bases de données, courtiers de messages, autres micro-services), il est crucial de s'assurer que les tests d'intégration reflètent correctement ces interactions. Les Testcontainers offrent plusieurs avantages dans ce contexte :
  - **Environnement de test réaliste** : Il permet de créer un environnement de test réaliste en lançant des instances réelles de services externes dans des conteneurs Docker. Ceci est particulièrement utile pour tester les interactions des micro-services avec les bases de données, d'autres micro-services ou des services tiers.
  - **Tests d'intégration** : Les conteneurs de test facilitent les tests d'intégration en fournissant un environnement programmable pour tester la façon dont les micro-services interagissent avec leurs dépendances, en s'assurant que les points d'intégration fonctionnent comme prévu.
  - **Compatibilité avec l'intégration continue** : Grâce à sa capacité à créer des environnements isolés et cohérents, Testcontainers est bien adapté à une utilisation dans les pipelines d'intégration continue, garantissant que les tests sont fiables et reproductibles dans différents environnements.
  - **Efficacité du développement** : Les développeurs peuvent rapidement écrire et exécuter des tests d'intégration sans avoir à configurer manuellement des dépendances externes, ce qui accélère le cycle de développement et de test.



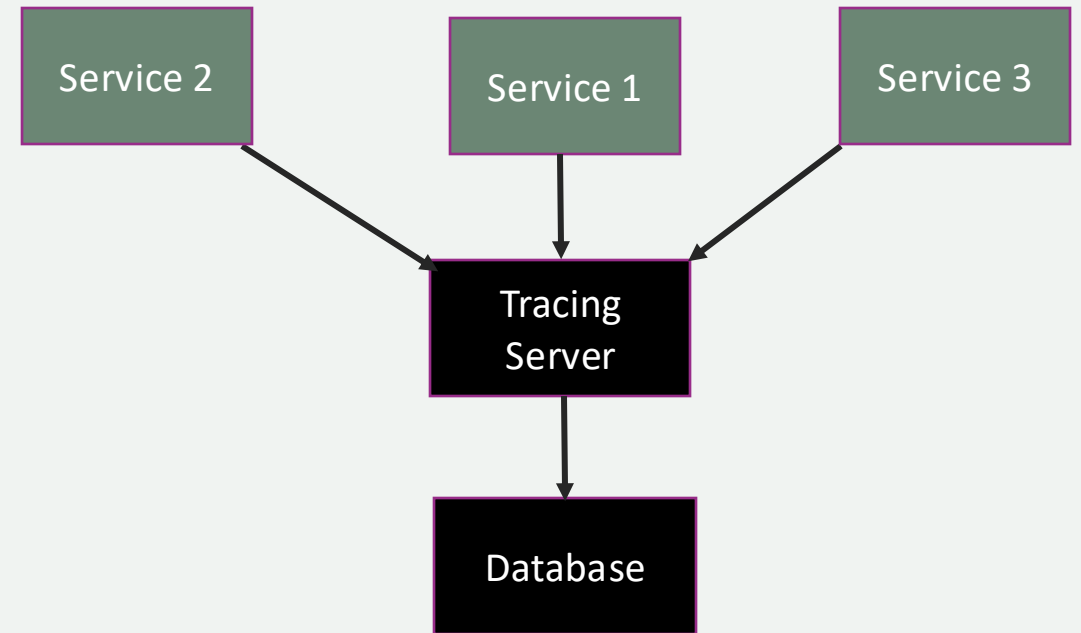
# Centralisation de la configuration avec Spring Cloud Config

- **Cohérence entre les services** : En centralisant la gestion de la configuration, on s'assure que tous les micro-services disposent d'un ensemble cohérent de configurations, ce qui réduit la probabilité de problèmes liés à la configuration.
- **Facilité de gestion de l'environnement** : Il simplifie la gestion des configurations dans différents environnements, ce qui facilite le déploiement et la gestion des micro-services dans les environnements de développement, de test et de production.
- **Agilité et flexibilité** : La possibilité de mettre à jour dynamiquement les configurations sans redémarrer les services permet aux équipes d'être plus agiles et réactives face aux changements. Ceci est particulièrement utile dans les pipelines de déploiement continu et d'intégration continue (CI/CD).
- **Découplage entre la configuration et le code** : En externalisant la configuration, Spring Cloud Config aide à découpler la configuration du code de l'application. Cette séparation des préoccupations rend l'application plus modulaire et plus facile à maintenir.



# Traces distribuées avec Zipkin

- Zipkin est un système de trace distribué open-source qui aide à rassembler les données temporelles nécessaires pour résoudre les problèmes de latence dans les architectures de micro-services.
- Il a été développé à l'origine par Twitter et est basé sur le document Google Dapper.
- Zipkin permet de suivre les requêtes au fur et à mesure qu'elles circulent dans les différents micro-services, ce qui facilite la compréhension du cheminement des requêtes et l'identification des latences.



# Logging et Monitoring

- Dans une application à base de micro-services, chaque service produit des logs
- Il est nécessaire, afin d'avoir une vue d'ensemble, de pouvoir agréger tous les logs à des fins de visualisation / analyse en temps réel
- Il existe des outils permettant de centraliser les logs sur la base de requêtes
  - Elastic Search / Logstash
  - Prometheus
- À combiner avec d'autres outils pour visualiser ces logs
  - Le plus souvent sous la forme de séries temporelles
  - Kibana (avec ES)
  - Grafana (avec Prometheus ou ES)
- Il peut être intéressant de combiner ces outils de monitoring à des outils de ML afin d'anticiper les problèmes (maintenance préventive) ou d'automatiser sa résolution (ne serait-ce que par sa planification avec des outils dédiés comme PagerDuty)

# cAdvisor

- cAdvisor (Container Advisor) est un outil open source développé par Google pour surveiller et analyser les performances des conteneurs.
  - Il collecte des métriques en temps réel sur l'utilisation des ressources (CPU, mémoire, réseau, etc.) des conteneurs et les expose via une interface web ou des points de terminaison (endpoints) compatibles avec des outils de monitoring comme Prometheus.
  - C'est une solution très simple à mettre en œuvre
  - Mais conçu pour fonctionner **localement**, sans système de notification
- **Avantages:**
  - Facile à déployer : cAdvisor est disponible sous forme d'image Docker, ce qui le rend très simple à déployer.
  - Open source : Gratuit et entièrement open source.
  - Intégration facile : Compatible avec des outils populaires comme Prometheus et Grafana.
  - Visibilité en temps réel : Fournit des métriques en temps réel pour une surveillance proactive.

# CAdvisor - Fonctionnalités

- **Surveillance des conteneurs :**
  - Collecte des métriques sur l'utilisation du CPU, de la mémoire, du disque et du réseau.
  - Fournit des informations détaillées sur chaque conteneur, y compris les limites de ressources et l'utilisation actuelle.
- **Interface web :**
  - Une interface web intuitive pour visualiser les métriques en temps réel.
  - Affiche des graphiques et des tableaux pour chaque conteneur.
- **Intégration avec Prometheus :**
  - Expose les métriques via un endpoint compatible avec Prometheus, ce qui permet une intégration facile avec des outils de surveillance comme Grafana.
- **Support multi-hôte :**
  - Peut surveiller les conteneurs sur plusieurs hôtes Docker.
- **Faible overhead :**
  - Léger et conçu pour fonctionner avec un impact minimal sur les performances du système.

# CAdvisor – docker-compose

```
cadvisor:  
  image: gcr.io/cadvisor/cadvisor:latest  
ports:  
  - "8080:8080"  
volumes:  
  - /:/rootfs:ro  
  - /var/run:/var/run:rw  
  - /sys:/sys:ro  
  - /var/lib/docker:/var/lib/docker:ro
```



# Comparaison avec Podman

# Architecture et Sécurité

- **Architecture sans démon (daemonless) :**
  - Podman : Contrairement à Docker, Podman n'utilise pas de démon central pour gérer les conteneurs. Chaque commande Podman est exécutée directement par l'utilisateur, ce qui réduit les risques de sécurité et les points de défaillance uniques.
  - Docker : Docker repose sur un démon (dockerd) qui doit être en cours d'exécution pour gérer les conteneurs. Cela peut introduire des vulnérabilités de sécurité et une complexité supplémentaire.
- **Sécurité :**
  - Podman : Podman fonctionne en mode rootless par défaut, ce qui signifie que les conteneurs peuvent être exécutés par des utilisateurs non privilégiés. Cela réduit les risques liés à l'exécution de conteneurs avec des privilèges élevés.
  - Docker : Bien que Docker ait ajouté le support rootless, il est plus couramment utilisé avec des privilèges root, ce qui peut poser des problèmes de sécurité si le démon est compromis.

# Intégration et Compatibilité

- **Intégration avec systemd :**

- Podman : Podman est conçu pour s'intégrer étroitement avec systemd, permettant de gérer les conteneurs comme des services systemd. Cela facilite la gestion des conteneurs dans des environnements basés sur systemd.
- Docker : Docker peut également être intégré avec systemd, mais cette intégration est moins native et peut nécessiter des configurations supplémentaires.

- **Compatibilité avec Docker :**

- Podman : Podman est compatible avec les images Docker et peut utiliser les mêmes fichiers de configuration (comme Dockerfile) pour construire des images. Il peut également utiliser les registres Docker pour pull et push des images.
- Docker : Docker est le standard de facto pour les conteneurs, donc la compatibilité est native. Cependant, Podman peut souvent remplacer Docker sans modification majeure des workflows existants.

# Orchestration et Communauté

- **Orchestration :**

- Podman : Podman propose une solution d'orchestration appelée Podman Compose, qui est similaire à Docker Compose, ainsi que Podman Pods pour gérer des groupes de conteneurs. Cependant, il n'a pas de solution native pour l'orchestration à grande échelle comme Kubernetes.
- Docker : Docker propose Docker Swarm pour l'orchestration, ainsi qu'une intégration étroite avec Kubernetes. Docker Compose est également largement utilisé pour l'orchestration de conteneurs.

- **Communauté et support :**

- Podman : Podman est soutenu par Red Hat et est de plus en plus adopté dans les environnements Linux, en particulier ceux qui utilisent des distributions Red Hat comme RHEL et Fedora.
- Docker : Docker a une communauté très large et est largement adopté dans l'industrie. Il bénéficie d'un support étendu et d'une documentation abondante.

# Conclusion

# Points essentiels à retenir

- Les conteneurs sont des environnements éphémères et isolés. Ils sont idéaux pour les flux de travail d'intégration et de livraison continues (CI/CD).
- Le cycle de vie des conteneurs est géré par diverses commandes et configurations Docker.
- La persistance des données dans Docker est gérée séparément du cycle de vie du conteneur par le biais de volumes ou de montages bind.
- Les outils d'orchestration peuvent gérer les cycles de vie de nombreux conteneurs simultanément dans des systèmes plus importants => Kubernetes
- Règles de bonnes pratiques:
  - Images de confiance : Utilisez des images officielles et vérifiées pour réduire les risques de vulnérabilités.
  - Scanning des images : Effectuez régulièrement des analyses de sécurité sur les images Docker.
  - Principe de moindre privilège : Exécutez les conteneurs avec le minimum de permissions nécessaires.

# Le mot de la fin

- Les architectures en micro-services ont beaucoup gagné en popularité ces dernières années
  - Ils apportent réactivité et flexibilité
- Leur développement et leurs tests nécessitent l'utilisation de frameworks dédiés
- Les conteneurs permettent de rapprocher les métiers du développement et ceux de l'opérationnalisation et de la maintenance
  - Chaines CI/CD plus fluides, plus efficaces
- Cependant, le déploiement dans le cloud peut poser bien des soucis en termes de qualité de service ou tolérance aux pannes
  - ECS et Kubernetes permettent aux développeurs de s'affranchir d'une grande partie de ces problèmes



# Mes formations en lien avec cette présentation

- Programmation réactive avec Reactor (en java ou Kotlin)
- Développement de micro-services avec Micronaut
- Le projet Spring Cloud
- Orchestration de micro-services avec Kubernetes (incl. Helm)
- IaS avec Terraform