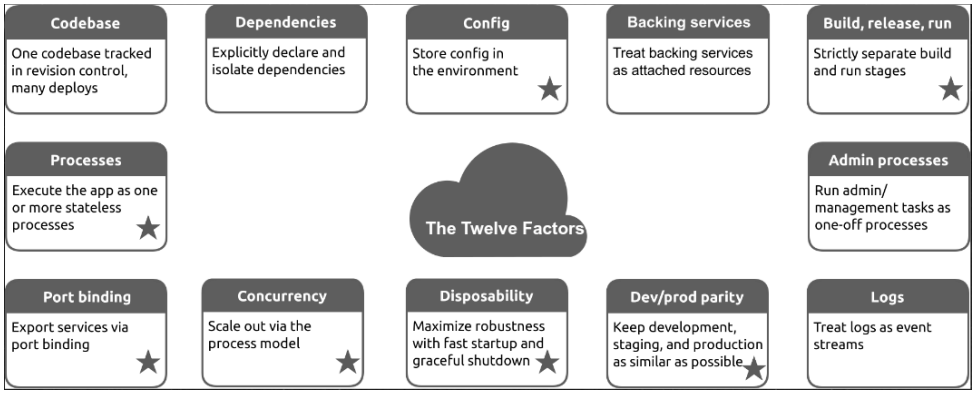
Chapitre 3
Introduction à Cloud Ready et Cloud Native

1. Cloud Ready

Cloud Ready, littéralement **prêt pour le cloud** en français, désigne un ensemble de principes de développement d'applications pour les rendre aptes à être déployées dans le cloud, ou dans un orchestrateur de conteneurs.

Lorsqu'on parle d'applications Cloud Ready, on cite souvent les ***12 factor app principles*** qui sont un ensemble de douze principes (ou facteurs) pour le développement d'applications à déployer dans le cloud ou Kubernetes. Ces principes ont été définis par la société Heroku et sont disponibles à l'URL : https://12factor.net/fr/



Ces douze facteurs sont les suivants :

1. **Base de code** : avoir une base de code unique avec un système de contrôle de version pour tous les déploiements.

2. **Dépendances** : déclarer explicitement et isoler les dépendances.

3. **Configuration** : stocker la configuration dans l'environnement.

4. **Services externes** : traiter les services externes comme des ressources attachées.

5. **Assembler, publier, exécuter** : séparer strictement les étapes d'assemblage et d'exécution.

6. **Processus** : exécuter l'application comme un ou plusieurs processus sans état.

7. **Associations de ports** : exporter les services via des associations de ports.

8. **Concurrence** : grossir (***scale***) à l'aide du modèle de processus.

9. **Jetable** : maximiser la robustesse avec des démarrages rapides et des arrêts gracieux.

10. **Parité dev/prod** : garder le développement, la validation et la production aussi proches que possible.

11. **Logs** : traiter les logs comme des flux d'événements.

12. **Processus d'administration** : lancer les processus d'administration et de maintenance comme des processus uniques (***one-off-processes***).

Les applications déployées dans le cloud sont aussi fortement sujettes aux aléas du réseau et généralement constituées de plusieurs composants distribués. Pour fonctionner correctement, elles se basent le plus souvent sur les techniques de développement suivantes :

– **Tolérance à la panne** : la panne d'un service externe ne doit pas mettre en danger l'application.
– **Résilience** : l'application est capable de revenir à un état normal après un dysfonctionnement.
– **Observabilité** : on peut observer l'état d'un système de l'extérieur avec des données qu'il produit (métriques, logs et traces).

Le chapitre Cloud Ready présentera les principales extensions permettant la mise en place de ces principes avec Quarkus.

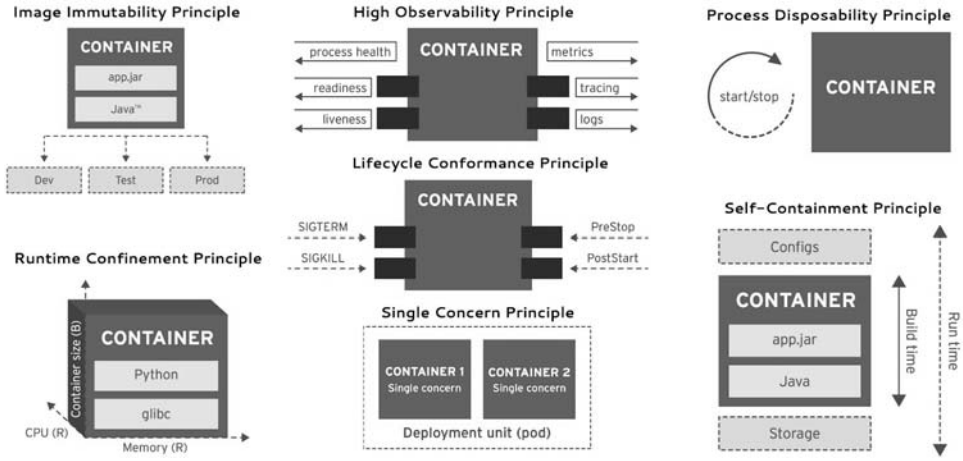
2. Applications prévues pour les conteneurs

Les applications Cloud Ready vont, la plupart du temps, être déployées dans un conteneur ou dans une solution de type Function-as-a-Service (FaaS).

Il est donc utile, dans le développement d'une application, de prendre en compte son fonctionnement au sein d'un conteneur. Bonne nouvelle, comme nous l'avons déjà précisé, Quarkus a été conçu pour fonctionner dans un conteneur.

Un conteneur va démarrer une application dans un processus isolé, à partir d'une image de conteneur généralement créé lors de la phase d'intégration ou de déploiement continu d'une application. La technologie de conteneur la plus courante est Docker, mais les orchestrateurs modernes de conteneurs ainsi que les fournisseurs cloud supportent généralement n'importe quel conteneur obéissant au standard OCI (*Open Container Initiative*).

Red Hat a écrit un livre blanc qui liste sept principes pour le développement d'applications pensées pour les conteneurs. Il est schématisé dans cette infographie :



Source https://kubernetes.io/blog/2018/03/principles-of-container-app-design/ - CC BY 4.O

Ces principes sont les suivants :

– **Une seule préoccupation** : chaque conteneur répond à une seule préoccupation et le fait bien.

– **Auto-contenance** : un conteneur repose uniquement sur la présence du noyau Linux. Des bibliothèques supplémentaires sont ajoutées lorsque le conteneur est construit.

– **Images immuables** : les applications conteneurisées sont censées être immuables et, une fois construites, elles ne sont pas censées changer entre différents environnements.

– **Haute observabilité** : chaque conteneur doit mettre en œuvre toutes les API nécessaires pour permettre à la plateforme de les observer de la meilleure façon possible.

– **Conformité du cycle de vie** : un conteneur doit se conformer aux événements standards de cycle de vie de la plateforme.

– **Processus jetables** : les applications conteneurisées doivent être aussi éphémères que possible et prêtes à être remplacées par une autre instance de conteneur à tout moment.
– **Confinement au moment de l'exécution** : chaque conteneur doit déclarer ses besoins en ressources et limiter l'utilisation des ressources aux besoins indiqués.

Mettre en place ces principes permet de garantir que l'application fonctionnera correctement dans un orchestrateur de conteneurs tel que Kubernetes.

Le chapitre Conteneurisation et Kubernetes présentera comment construire une image Docker d'une application et la déployer dans Kubernetes.

3. Cloud Native

Nous utiliserons dans ce livre le terme anglais ***Cloud Native*** et pas **Cloud Natif** car il désigne une approche et une architecture reconnues.

Pour reprendre la définition de Wikipédia (https://en.wikipedia.org/wiki/Cloud-native_computing) :

Cloud Native est une approche de développement logiciel qui utilise le cloud pour « créer et exécuter des applications évolutives dans des environnements modernes et dynamiques tels que les clouds publics, privés et hybrides ».

Les technologies telles que les conteneurs, les micro-services, les fonctions serverless et l'infrastructure immuable [...] sont des éléments communs de ce style architectural.

Certaines fonctionnalités offertes par un fournisseur cloud sont dites *serverless* ou sans serveur en français. C'est bien sûr un abus de langage, il y a bien des serveurs quelque part (dans les centres de données du fournisseur de cloud), mais ceux-ci ne sont pas visibles pour l'utilisateur qui va utiliser un service totalement géré par le fournisseur cloud proposant de la mise à l'échelle à 0 et du paiement à l'utilisation.

Selon sa définition, une application est cloud native si elle utilise les fonctionnalités d'un fournisseur de cloud dans son architecture. C'est le cas par exemple si elle fait usage d'une base de données gérée par le fournisseur cloud, ou serverless, à la place d'une base de données gérée par l'application ; ou encore si elle se sert des solutions de stockage ou de messaging offertes par le fournisseur cloud.

Une application cloud native va généralement être déployée dans une solution de type Function-as-a-Service (FaaS) ou Container-as-a-Service (CaaS).

Les chapitres Déployer dans Amazon Web Services, Déployer dans Google Cloud Platform et Déployer dans Microsoft Azure présenteront les principales fonctionnalités cloud native offertes par Quarkus pour chacun des trois grands fournisseurs cloud.