KUBERNETES

Maîtrisez l'orchestrateur
des infrastructures du futur

Chez le même éditeur

Docker - *Pratique des architectures à base de conteneurs*, P-Y. Cloux, T. Garlot, J. Kohler (2e édition) 336 pages, 2019.

Découvrir DevOps - *L'essentiel pour tous les métiers*, S. Goudeau, S. Metias (2e édition) 240 pages, 2018.

Mettre en œuvre DevOps - *Comment évoluer vers une DSI agile*, A. Sacquet, C. Rochefolle, (2e édition) 288 pages, 2018.

Scrum - *Pour une pratique vivante de l'agilité*, C. Aubry, (5e édition) 384 pages, 2018.

KUBERNETES

Maîtrisez l'orchestrateur des infrastructures du futur

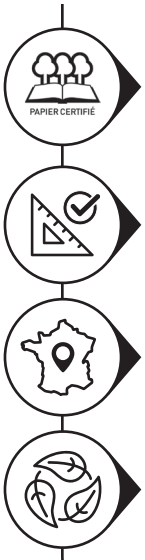
Kelsey Hightower
Brendan Burns
Joe Beda
Lachlan Everson

Traduit de l'anglais par Dominique Maniez

DUNOD

Authorized French translation of material from the English edition
of *Kubernetes : Up an d Running*
ISBN : 978-2-49-193567-5

NOUS NOUS ENGAGEONS EN FAVEUR DE L'ENVIRONNEMENT :



Nos livres sont imprimés sur des papiers certifiés pour réduire notre impact sur l'environnement.

Le format de nos ouvrages est pensé afin d'optimiser l'utilisation du papier.

Depuis plus de 30 ans, nous imprimons 70 % de nos livres en France et 25 % en Europe et nous mettons tout en œuvre pour augmenter cet engagement auprès des imprimeurs français.

Nous limitons l'utilisation du plastique sur nos ouvrages (film sur les couvertures et les livres).

© Dunod, 2023
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-082868-5

Préface

Kubernetes souhaiterait remercier tous les administrateurs système qui se sont réveillés à 3 heures du matin pour redémarrer un processus, mais aussi tous les développeurs qui ont mis leur code en production pour constater qu'il ne fonctionnait pas comme sur leur ordinateur portable, et puis encore tous les architectes système qui ont à tort lancé un test de charge sur un serveur de production en raison d'un nom d'hôte inutilisé qu'ils n'avaient pas mis à jour. Ce sont toutes ces souffrances, ces heures indues et ces erreurs étranges qui ont inspiré le développement de Kubernetes. Si l'on veut résumer notre propos en une seule phrase : Kubernetes a pour objectif de simplifier radicalement les tâches de création, de déploiement et de maintenance des systèmes distribués. Il a été inspiré par des décennies de pratique de conception de systèmes fiables et il a été élaboré à partir des usages de la base pour en faire une expérience, si ce n'est euphorique, tout du moins agréable. Nous espérons que vous allez apprécier ce livre !

◆ *À qui s'adresse ce livre ?*

Que vous soyez néophyte dans les systèmes distribués ou que vous déployiez des systèmes cloud-native depuis des années, les conteneurs et Kubernetes peuvent vous aider à réaliser des progrès en matière de vitesse, d'agilité, de fiabilité et d'efficacité. Cet ouvrage décrit l'orchestrateur de clusters Kubernetes et la façon dont ses outils et son API peuvent être utilisés pour améliorer le développement, la livraison, la sécurité et la maintenance des applications distribuées. Bien qu'aucune expérience antérieure avec Kubernetes ne soit requise, il est préférable d'être à l'aise avec la création et le déploiement des applications basées sur un serveur si vous voulez tirer le meilleur parti de ce livre. La familiarité avec des concepts comme l'équilibrage de charge et le stockage réseau sera utile, mais pas nécessaire. De la même manière, une expérience de Linux, des conteneurs Linux, et de Docker, n'est pas essentielle, mais elle vous aidera à profiter au maximum des ressources de cet ouvrage.

◆ *Pourquoi avons-nous écrit ce livre ?*

Nous avons été impliqués dans Kubernetes depuis son tout début. Cela a vraiment été extraordinaire d'assister à la transformation de cette technologie qui est passée d'une curiosité largement utilisée pour des expériences à une infrastructure cruciale de production qui alimente les applications à grande échelle dans des domaines variés qui vont de l'apprentissage automatique (*machine learning*) aux services en ligne. Quand cette transition s'est produite, il est devenu de plus en plus clair qu'un livre qui décrirait à la fois la façon d'utiliser les concepts de base de Kubernetes et les motivations sous-jacentes à l'élaboration de ces concepts serait une contribution importante pour le développement d'applications cloud-native. Nous espérons qu'à la lecture de ce livre, non seulement vous allez apprendre à créer des applications fiables et évolutives en vous appuyant sur Kubernetes, mais vous allez aussi comprendre les défis majeurs des systèmes distribués qui ont conduit au développement de Kubernetes.

◆ *Pourquoi avons-nous mis à jour ce livre ?*

L'écosystème Kubernetes a continué à croître et à évoluer depuis les deux premières éditions de ce livre. Plusieurs versions de Kubernetes ont vu le jour, de nombreux outils et modèles d'utilisation de Kubernetes s'étant imposés comme des standards. Dans la troisième édition, nous nous sommes concentrés sur l'ajout de sujets qui ont pris de l'importance dans l'écosystème Kubernetes, notamment la sécurité, l'accès à Kubernetes à partir des langages de programmation, ainsi que les déploiements d'applications multiclusters. Nous avons également mis à jour tous les chapitres existants afin de refléter les changements et l'évolution de Kubernetes depuis la première édition. Nous nous attendons d'ailleurs à réviser à nouveau ce livre dans quelques années (et nous sommes impatients de le faire), car Kubernetes continue d'évoluer.

◆ *Un mot sur les applications cloud-native actuelles*

Des premiers langages de programmation, à la programmation orientée objet, en passant par le développement de la virtualisation et des infrastructures dans le cloud, l'histoire de l'informatique est une histoire du développement d'abstractions qui cachent la complexité et vous permettent de créer des applications toujours plus sophistiquées. Malgré cela, le développement d'applications fiables et évolutives est encore beaucoup plus difficile qu'il ne devrait l'être. Ces dernières années, les conteneurs et les API d'orchestration de conteneurs comme Kubernetes se sont révélés être une abstraction importante qui simplifie radicalement le développement de systèmes distribués fiables et évolutifs. Les conteneurs et les orchestrateurs permettent aux développeurs de créer et de déployer des applications avec une vitesse, une agilité et une fiabilité qui seraient passées pour de la science-fiction il y a seulement quelques années.

◆ *Organisation de ce livre*

Ce livre est organisé de la manière suivante. Le premier chapitre présente les principaux avantages de Kubernetes sans rentrer trop profondément dans les détails. Si vous n'avez jamais utilisé Kubernetes, c'est l'occasion idéale pour commencer à comprendre pourquoi vous devriez lire le reste du livre.

Le chapitre 2 fournit une introduction détaillée aux conteneurs et au développement d'applications dans des conteneurs. Si vous n'avez jamais vraiment testé Docker auparavant, ce chapitre constitue une introduction utile. Si vous êtes déjà un expert Docker, ce sera sans doute une révision.

Le chapitre 3 couvre la procédure de déploiement de Kubernetes. Bien que la plus grande partie de ce livre se concentre sur la façon d'utiliser Kubernetes, vous devez disposer d'un cluster opérationnel avant de commencer à l'utiliser. Bien que l'exécution d'un cluster pour la production dépasse la portée de ce livre, ce chapitre présente quelques méthodes simples pour créer un cluster afin que vous puissiez comprendre comment utiliser Kubernetes. Le chapitre 4 couvre une sélection de commandes courantes utilisées pour interagir avec un cluster Kubernetes.

À partir du chapitre 5, nous nous plongeons dans les détails du déploiement d'une application à l'aide de Kubernetes. Nous couvrons les pods (chapitre 5), les étiquettes

et les annotations (chapitre 6), les services (chapitre 7), Ingress (chapitre 8), et les ReplicaSets (chapitre 9). Ces éléments constituent les bases essentielles de ce dont vous avez besoin pour déployer votre service dans Kubernetes. Nous couvrons ensuite les déploiements (chapitre 10), qui relient tous les éléments du cycle de vie d'une application complète.

Après ces chapitres, nous étudions certains objets plus spécialisés de Kubernetes : les DaemonSets (chapitre 11), les jobs (chapitre 12), les ConfigMaps et les secrets (chapitre 13). Bien que ces chapitres soient essentiels pour de nombreuses applications de production, vous pouvez les ignorer si vous démarrez l'apprentissage de Kubernetes, et y revenir plus tard après avoir acquis plus d'expérience et d'expertise.

Nous présentons ensuite le contrôle d'accès basé sur les rôles (chapitre 14) et couvrons les maillages de services (chapitre 15) et l'intégration du stockage (chapitre 16) dans Kubernetes. Nous abordons l'extension de Kubernetes (chapitre 17) et l'accès à Kubernetes à partir des langages de programmation (chapitre 18). Nous nous concentrons ensuite sur la sécurisation des pods (chapitre 19) ainsi que sur la gouvernance de Kubernetes (chapitre 20).

Enfin, nous terminons par quelques exemples sur la façon de développer et de déployer des applications multiclusters (chapitre 21) et nous abordons la manière d'organiser vos applications avec un système de contrôle des sources (chapitre 22).

◆ *Ressources en ligne*

Il est souhaitable d'installer Docker (https://docker.com) et de vous familiariser avec sa documentation si ce n'est déjà fait.

De la même manière, il est préférable d'installer l'outil en ligne de commande `kubectl` (https://kubernetes.io). Vous voudrez sans doute aussi vous abonner à la chaîne Slack Kubernetes (http://slack.kubernetes.io), où vous trouverez une grande communauté d'utilisateurs qui sont tout disposés à répondre à vos questions à presque n'importe quelle heure du jour et de la nuit.

Enfin, quand vous aurez bien progressé, vous pourrez participer au dépôt open source Kubernetes sur GitHub (https://github.com/kubernetes/kubernetes).

◆ *Conventions utilisées dans ce livre*

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique les nouveaux termes, les URL, les adresses électroniques, les noms de fichiers et les extensions de fichier.

`Police à espacement fixe`

Utilisée pour les listings de code des programmes, ainsi que dans les paragraphes pour faire référence à des éléments de programme comme des noms de variables ou de fonctions, des bases de données, des types de donnée, des variables d'environnement, des instructions et des mots-clés.

`Police à espacement fixe en gras`

Affiche les commandes ou tout autre texte qui doit être saisi littéralement par l'utilisateur.

`Police à espacement fixe en italique`

Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.

Cette icône représente une astuce, une suggestion ou une note.

Cette icône indique un avertissement ou une mise en garde.

◆ *Utilisation des exemples de code*

Les ressources supplémentaires (exemples de code, exercices, etc.) sont téléchargeables à https://github.com/kubernetes-up-and-running/examples.

Ce livre est conçu pour faciliter votre travail. En général, si un exemple de code est proposé avec ce livre, vous pouvez l'utiliser dans vos programmes et votre documentation. Vous n'avez pas besoin de nous contacter pour une autorisation à moins que vous ne reproduisiez une partie importante du code. Par exemple, l'écriture d'un programme qui utilise plusieurs extraits de code de ce livre ne nécessite pas d'autorisation. En revanche, la vente ou la distribution d'un CD-ROM d'exemples extraits d'ouvrages de chez O'Reilly exige une autorisation. Si vous répondez à une question en citant ce livre et en citant un exemple de code, vous n'avez pas besoin d'une d'autorisation, mais si vous incorporez beaucoup d'exemples de code tirés de ce livre dans la documentation de votre produit, il vous faudra demander une autorisation.

Nous apprécions, sans que cela constitue une exigence, d'être référencés quand vous citez notre travail. Une référence inclut généralement le titre de l'ouvrage, son auteur, l'éditeur et le numéro ISBN. Par exemple : « Kubernetes: Up and Running, 3rd edition, par Brendan Burns, Joe Beda, Kelsey Hightower, et Lachlan Evenson (O'Reilly). Copyright 2019 Brendan Burns, Joe Beda, Kelsey Hightower, et Lachlan Evenson, 978-1-098-11020-8. » pour la version originale, et Kubernetes Maîtriser l'orchestrateur des infrastructures du futur, des mêmes auteurs, Dunod, 2023 - 9782100828685 pour la version en français.

Si vous pensez que votre utilisation d'exemples de code dépasse le cadre juridique de la courte citation, n'hésitez pas à nous contacter à permissions@oreilly.com.

Nous avons une page Web pour la version originale de ce livre en anglais, où nous répertorions les errata, les exemples et toutes les informations supplémentaires : https://learning.oreilly.com/library/view/~/9781098110192/ et sur www.dunod.com pour cette version en français.

Table des matières

Introduction

Kubernetes est un orchestrateur open source pour le déploiement d'applications en conteneurs. Il a été développé à l'origine par Google, qui s'est inspiré de sa longue expérience du déploiement de systèmes évolutifs et fiables dans des conteneurs via des API orientées application[1].

Depuis son introduction en 2014, Kubernetes est devenu l'un des projets open source les plus importants et les plus populaires au monde. Il est aujourd'hui l'API standard pour la création d'applications cloud native, qui est présente dans presque tous les clouds publics. Kubernetes est une infrastructure éprouvée pour les systèmes distribués qui convient aux développeurs cloud-native, qu'ils travaillent sur un cluster de Raspberry Pi ou sur une ferme de serveurs avec des ordinateurs à la pointe de la technologie. Kubernetes fournit le logiciel nécessaire pour construire et déployer avec succès des systèmes distribués fiables et évolutifs.

Vous vous demandez peut-être ce que nous entendons par « systèmes distribués fiables et évolutifs ». De plus en plus de services sont livrés sur le réseau via des API. Ces API sont souvent livrées par un *système distribué*, les différents éléments qui implémentent ces API fonctionnant sur différentes machines, connectées via le réseau et coordonnant leurs actions via des communications réseau. Dans la mesure où nous comptons de plus en plus sur ces API pour gérer tous les aspects de notre vie quotidienne (par exemple, trouver l'itinéraire vers l'hôpital le plus proche), ces systèmes doivent être extrêmement *fiables*. Ils ne peuvent pas se permettre de tomber en panne, même si une partie du système plante ou fonctionne mal. De la même manière, ils doivent rester *disponibles* même pendant les déploiements de logiciels ou les opérations de maintenance. Enfin, comme de plus en plus de monde est présent sur Internet et utilise de tels services, ils doivent être très *évolutifs* afin de pouvoir accroître leur

1. Brendan Burns et al., "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade", *ACM Queue*, vol. 14 (2016): 70–93, disponible à https://oreil.ly/ltE1B.

capacité pour satisfaire une utilisation en augmentation continue sans avoir à modifier radicalement la conception du système distribué qui implémente les services. Dans de nombreux cas, cela signifie également qu'il faut augmenter automatiquement (et réduire) la capacité, afin que votre application soit la plus efficace possible.

Quelle que soit votre expérience en matière de conteneurs, de systèmes distribués, et de Kubernetes (vous envisagez peut-être de créer votre application sur une infrastructure de cloud public, dans des data centers privés ou dans un environnement hybride), nous pensons que ce livre vous permettra de tirer le meilleur parti de Kubernetes.

Il existe de nombreuses raisons qui poussent les gens à utiliser des conteneurs et des API conteneur comme Kubernetes, mais nous estimons qu'ils cherchent tous à bénéficier au moins de l'un de ces avantages :

- ✓ Vitesse de développement
- ✓ Évolutivité (du logiciel et des équipes)
- ✓ Abstraction de l'infrastructure
- ✓ Efficacité
- ✓ Écosystème cloud native

Dans les paragraphes suivants, nous décrivons la manière dont Kubernetes peut vous aider à profiter de chacune de ces fonctionnalités.

1.1 VITESSE

La rapidité est le facteur clé aujourd'hui dans presque tous les développements informatiques. L'industrie du logiciel est passée de la livraison de produits sous la forme de CD ou de DVD à des logiciels livrés grâce au réseau via des services Web mis à jour toutes les heures. Cette évolution du paysage signifie que ce qui permet de vous différencier de vos concurrents est bien souvent la vitesse à laquelle vous pouvez développer et déployer de nouveaux composants, ou la vitesse à laquelle vous pouvez répondre aux innovations développées par d'autres.

Il est cependant important de noter que cette rapidité n'est pas simplement définie en termes de vitesse brute. Même si vos utilisateurs sont toujours à la recherche d'améliorations itératives, ils resteront plus intéressés par un service extrêmement fiable. Par le passé, on tolérait qu'un service soit en maintenance à minuit tous les soirs, mais aujourd'hui, nos utilisateurs s'attendent à bénéficier d'une disponibilité constante, même si le logiciel qu'ils exécutent est mis à jour en permanence.

Par conséquent, la vitesse n'est pas mesurée en fonction du nombre de mises à jour que vous pouvez livrer en une heure ou en un jour, mais plutôt en fonction du nombre de choses que vous pouvez livrer tout en maintenant un service hautement disponible.

C'est dans ce but que les conteneurs et Kubernetes peuvent fournir les outils dont vous avez besoin pour réagir rapidement, tout en restant disponibles. Les concepts de base qui permettent cela sont au nombre de quatre :

- ✓ L'immutabilité
- ✓ La configuration déclarative
- ✓ Les systèmes d'auto-guérison en ligne
- ✓ Les bibliothèques et les outils réutilisables partagés

Ces idées sont toutes interdépendantes afin d'améliorer radicalement la rapidité avec laquelle vous pouvez déployer de manière fiable de nouveaux logiciels.

1.1.1 La valeur de l'immutabilité

Les containers et Kubernetes encouragent les développeurs à créer des systèmes distribués qui respectent les principes d'infrastructure immutable. Avec une infrastructure immutable, une fois qu'un artefact est créé dans le système, les modifications de l'utilisateur ne peuvent pas le faire changer.

Traditionnellement, les ordinateurs et les systèmes logiciels sont considérés comme des infrastructures *mutables*. Avec une infrastructure mutable, les modifications sont appliquées en tant que mises à jour incrémentielles à un système existant. Ces mises à jour peuvent se produire en une seule fois ou s'étaler sur une longue période. Une mise à niveau du système via l'outil `apt-get update` est un bon exemple de mise à jour d'un système mutable. En exécutant `apt`, on télécharge séquentiellement tous les fichiers binaires mis à jour, on les copie au-dessus des fichiers binaires plus anciens, et on effectue des mises à jour incrémentielles des fichiers de configuration. Avec un système mutable, l'état actuel de l'infrastructure n'est pas représenté comme un artefact unique, mais plutôt comme une accumulation de mises à jour et de modifications incrémentielles au fil du temps. Sur de nombreux systèmes, ces mises à jour incrémentielles proviennent non seulement des mises à niveau du système, mais aussi des modifications de l'exploitant. En outre, dans tout système géré par une grande équipe, il est fort probable que ces modifications auront été effectuées par de nombreuses personnes différentes et, dans de nombreux cas, n'auront pas été documentées.

En revanche, dans un système immutable, au lieu d'une série de mises à jour et de modifications incrémentielles, une image entièrement nouvelle et complète est créée, et la mise à jour remplace simplement en une seule opération la totalité de l'image par l'image la plus récente. Il n'y a pas de modifications incrémentielles. Comme vous pouvez l'imaginer, cela constitue un changement significatif dans l'univers plus traditionnel de la gestion des configurations.

Pour illustrer notre propos dans le monde des conteneurs, prenez l'exemple de deux manières différentes de mettre à niveau votre logiciel :

1. Vous pouvez vous connecter à un conteneur, exécuter une commande pour télécharger votre nouveau logiciel, supprimer l'ancien serveur et démarrer le nouveau.
2. Vous pouvez créer une nouvelle image de conteneur, la pousser dans un registre de conteneurs, supprimer le conteneur existant et en démarrer un nouveau.

À première vue, il peut sembler difficile de faire la différence entre ces deux approches. En quoi le fait de créer un nouveau conteneur peut-il améliorer la fiabilité ?

La différence principale est l'artefact que vous créez, et l'enregistrement de la façon dont vous l'avez créé. Ces enregistrements permettent d'identifier exactement les différences entre les versions et, si la nouvelle version a un souci, il est facile de déterminer ce qui a changé et comment résoudre le problème.

En outre, la construction d'une nouvelle image ne modifie pas une image existante, ce qui signifie que l'ancienne image est toujours présente, et qu'il est possible de l'utiliser rapidement pour une restauration si une erreur se produit. En revanche, une fois que

vous copiez votre nouveau fichier binaire sur un fichier binaire existant, une telle restauration est presque impossible.

Les images de conteneurs immutables sont au cœur de tout ce que vous allez créer dans Kubernetes. Il est possible de changer grâce à des instructions les conteneurs en cours d'exécution, mais il s'agit là d'un anti-modèle à n'employer que dans les cas extrêmes où il n'y a pas d'autres options (par exemple, si c'est la seule façon de réparer temporairement un système critique de production). Et même dans ce cas-là, les modifications doivent aussi être enregistrées par le biais d'une mise à jour de configuration déclarative à un moment ultérieur, quand la crise est terminée.

1.1.2 Configuration déclarative

L'immutabilité s'étend au-delà des conteneurs en cours d'exécution dans votre cluster et s'applique aussi à la façon dont vous décrivez votre application à Kubernetes. Tout élément dans Kubernetes est un *objet de configuration déclarative* qui représente l'état désiré du système. C'est le travail de Kubernetes de s'assurer que l'état réel du monde correspond à cet état désiré.

Comme dans l'opposition entre infrastructure mutable et infrastructure immutable, la configuration déclarative est une alternative à la configuration *impérative*, où l'état du monde est défini par l'exécution d'une série d'instructions plutôt que par une déclaration de l'état désiré du monde. Alors que les commandes impératives définissent des actions, les configurations déclaratives définissent un état.

Pour comprendre ces deux approches, prenez l'exemple de la tâche consistant à produire trois réplicas d'un logiciel. Avec une approche impérative, la configuration dirait : « exécuter A, exécuter B, et exécuter C ». La configuration déclarative correspondante serait : « les réplicas sont au nombre de trois ».

Comme elle décrit l'état du monde, la configuration déclarative n'a pas à être exécutée pour qu'on la comprenne. Son impact est déclaré concrètement. Étant donné que les effets de la configuration déclarative peuvent être compris avant son exécution, la configuration déclarative est beaucoup moins sujette aux erreurs. En outre, les outils traditionnels de développement de logiciels, comme le contrôle de code source, l'examen de codes et les tests unitaires, peuvent être utilisés dans la configuration déclarative selon des modalités qui sont impossibles avec les instructions impératives. L'idée de stocker la configuration déclarative dans le contrôle de la source est souvent appelée « infrastructure en tant que code ».

Ces derniers temps, le concept de GitOps a commencé à formaliser la pratique de l'infrastructure en tant que code avec le contrôle de la source comme source de vérité. Lorsque vous adoptez GitOps, les changements apportés à la production sont entièrement réalisés via des envois (*push*) vers un dépôt Git, qui sont ensuite répercutés dans votre cluster via l'automatisation. En effet, votre cluster Kubernetes de production est considéré comme un environnement en lecture seule. En outre, GitOps est de plus en plus intégré dans les services Kubernetes fournis par le cloud comme le moyen le plus simple de gérer de manière déclarative votre infrastructure cloud native.

La combinaison de l'état déclaratif stocké dans un système de contrôle de versions avec la capacité de Kubernetes à faire correspondre la réalité à cet état déclaratif

facilite grandement la restauration d'une modification. Il s'agit simplement de confirmer à nouveau l'état déclaratif précédent du système. Avec des systèmes impératifs, ceci est généralement impossible ; en effet, bien que les instructions impératives décrivent comment aller d'un point *A* à un point *B*, elles incluent rarement les instructions pour faire l'opération en sens inverse.

1.1.3 Systèmes d'auto-guérison

Kubernetes est un système d'auto-guérison en ligne. Lorsqu'il reçoit une configuration d'état désiré, il ne prend pas simplement un ensemble de mesures pour que l'état actuel corresponde à l'état désiré à un instant donné, il agit de manière continue pour s'assurer que l'état actuel correspond en permanence à l'état désiré. Cela signifie que non seulement Kubernetes va initialiser votre système, mais en plus qu'il le protégera contre toute panne ou perturbation qui pourrait le déstabiliser et affecter sa fiabilité.

Une réparation plus traditionnelle de l'exploitant implique une série d'étapes correctives manuelles, ou une intervention humaine effectuée en réponse à une alerte du système. Ce type de réparation impérative est plus chère (puisqu'elle exige généralement qu'un opérateur d'astreinte soit disponible pour assurer la réparation). Elle est aussi habituellement plus lente, car il faut souvent réveiller une personne qui doit se connecter pour traiter la demande d'intervention. En outre, elle est moins fiable puisque la série impérative d'opérations de réparation souffre de tous les problèmes de gestion impérative que nous avons décrits dans la section précédente. Les systèmes d'auto-guérison comme Kubernetes réduisent la charge de travail imposée aux opérateurs et améliorent la fiabilité globale du système en effectuant des réparations sûres plus rapidement.

Pour prendre un exemple concret de ce comportement d'auto-guérison, si vous déclarez un état désiré de trois réplicas à Kubernetes, il ne crée pas seulement trois réplicas : il s'assure en permanence qu'il y a exactement trois réplicas. Si vous créez manuellement un quatrième réplica, Kubernetes va en détruire un pour ramener à trois le nombre de réplicas. Si vous détruisez manuellement un réplica, Kubernetes va en créer un pour retourner à nouveau à l'état désiré de trois.

Les systèmes d'auto-guérison en ligne améliorent la productivité du développeur, car le temps et l'énergie que vous ne consacrez pas aux opérations de maintenance peuvent être investis pour développer et tester de nouvelles fonctionnalités.

Dans une forme plus avancée d'auto-guérison, un travail important a été réalisé récemment avec le paradigme d'*opérateur* pour Kubernetes. Avec les opérateurs, la logique plus avancée nécessaire à la maintenance, à la mise à l'échelle et à la guérison d'un logiciel spécifique (par exemple, MySQL) est encodée dans une application opérateur qui s'exécute en tant que conteneur dans le cluster. Le code de l'opérateur est responsable de la détection de l'état de santé et de la guérison, d'une manière plus ciblée et plus sophistiquée, que ce qui peut être réalisé via l'auto-guérison générique de Kubernetes. Ces fonctions sont souvent regroupées sous la forme d'opérateurs, qui sont abordés au chapitre 17.

1.2 ÉVOLUTIVITÉ DE VOTRE SERVICE ET DE VOS ÉQUIPES

Au fur et à mesure de l'évolution de la croissance de votre produit, il est inévitable que vous ayez besoin de redimensionner à la fois votre logiciel et les équipes qui le développent. Heureusement, Kubernetes peut vous aider à atteindre ces deux objectifs. Kubernetes parvient à l'évolutivité en favorisant les architectures *découplées*.

1.2.1 Découplage

Dans une architecture découplée, chaque composant est séparé des autres composants par des API définies et des équilibreurs de charge de services. Les API et les équilibreurs de charge isolent chaque partie du système des autres parties. Les API fournissent une mémoire tampon entre l'implémenteur et le consommateur, et les équilibreurs de charge fournissent une mémoire tampon entre les instances en cours exécution de chaque service.

Les composants de découplage via les équilibreurs de charge facilitent l'évolutivité des programmes qui composent votre service, car l'augmentation de la taille (et donc de la capacité) du programme peut se faire sans ajuster ni reconfigurer les autres couches de votre service.

Le découplage des serveurs via les API facilite l'évolutivité des équipes de développement car chaque équipe peut se concentrer sur un seul *micro-service* plus petit avec une surface plus facile à appréhender. Les API claires et nettes entre les microservices fluidifient la communication nécessaire entre les équipes pour créer et déployer les logiciels. L'inflation de la communication entre les équipes est souvent le principal facteur qui limite leur évolutivité.

1.2.2 Faciliter l'évolutivité des applications et des clusters

Concrètement, lorsque vous avez besoin de faire évoluer votre service, la nature immutable et déclarative de Kubernetes facilite l'implémentation de cette mise à l'échelle. Étant donné que vos conteneurs sont immutables et que le nombre de réplicas n'est qu'un chiffre dans une configuration déclarative, l'évolutivité de votre service est simplement une question de modification d'un nombre dans un fichier de configuration, et de déclaration de ce nouvel état à Kubernetes qui va s'occuper du reste tout seul. Vous pouvez également configurer l'évolutivité automatique et laisser Kubernetes gérer cela à votre place.

Bien sûr, ce type d'évolutivité suppose qu'il y ait des ressources disponibles dans votre cluster, et parfois vous aurez besoin d'agrandir le cluster lui-même. Une nouvelle fois, Kubernetes facilite cette tâche. Comme de nombreuses machines dans un cluster sont parfaitement identiques à d'autres machines de cet ensemble, et que les applications elles-mêmes sont découplées des détails de la machine grâce à des conteneurs, l'ajout de ressources supplémentaires dans un cluster n'est qu'une simple question de création d'image d'une nouvelle machine de la même classe à joindre au cluster. Ceci peut être accompli grâce à quelques commandes simples ou par l'emploi d'une image de machine préconstruite.

L'un des défis de l'évolutivité des ressources machine est de prévoir leur utilisation. Si vous utilisez une infrastructure physique, le temps d'obtention d'une nouvelle machine se mesure en jours ou en semaines. Sur les infrastructures physiques et à la fois sur les infrastructures dans le cloud, il est difficile de prévoir les coûts car il n'est pas aisé d'estimer les besoins de croissance et d'évolutivité des applications spécifiques.

Kubernetes peut simplifier la prévision des coûts de calcul. Pour comprendre comment cela fonctionne, imaginez que vous souhaitiez faire évoluer trois équipes. L'expérience vous a prouvé que la croissance de chaque équipe est très variable et donc difficile à prévoir. Si vous provisionnez des machines individuelles pour chaque service, vous n'avez pas d'autre choix que de faire une prévision en fonction de la croissance maximale attendue pour chaque service, puisque les machines dédiées à une équipe ne peuvent pas être utilisées par une autre équipe. Si au lieu de cela, vous utilisez Kubernetes pour découpler les équipes des machines spécifiques qu'elles utilisent, vous pouvez prévoir une croissance basée sur la croissance globale des trois services. La combinaison de trois taux de croissance variables en un seul taux de croissance réduit le bruit statistique et produit une prévision plus fiable de la croissance attendue. En outre, le découplage des équipes à partir de machines spécifiques signifie que les équipes peuvent partager des parties fractionnaires des machines des autres équipes, ce qui réduit encore plus la surcharge associée à la prévision de la croissance des ressources informatiques.

Enfin, Kubernetes permet de réaliser une mise à l'échelle automatique des ressources (pour diminuer ou augmenter les capacités). En particulier dans un environnement dans le cloud où de nouvelles machines peuvent être créées via des API, le fait de pouvoir combiner Kubernetes et la mise à l'échelle automatique, tant pour les applications que pour les clusters eux-mêmes, signifie que vous pouvez toujours adapter vos coûts à la charge actuelle.

1.2.3 Évolutivité des équipes de développement grâce à des micro-services

Comme de nombreuses études l'indiquent, la taille idéale d'une équipe est celle qui se contente de deux pizzas[2], soit environ six à huit personnes, parce qu'un groupe de cette taille se traduit souvent par un bon partage des connaissances, une prise de décision rapide, et un sens commun de l'objectif à atteindre. Les plus grosses équipes ont tendance à souffrir de problèmes hiérarchiques, d'un manque de visibilité et de querelles qui entravent l'agilité et le succès.

Cependant, de nombreux projets nécessitent un accroissement significatif de leurs ressources pour réussir et atteindre leurs objectifs. Par conséquent, il existe une tension entre la taille idéale pour que l'équipe soit agile et la taille nécessaire pour qu'elle puisse satisfaire les objectifs du produit.

La solution courante à cette tension a été le développement d'équipes découplées, orientées services, qui construisent chacune un microservice unique. Chaque petite équipe est responsable de la conception et de la livraison d'un service qui est consommé par d'autres petites équipes. L'agrégation de tous ces services fournit au bout du compte l'implémentation du périmètre global du produit.

---

2. NDT : on attribue cette expression à Jeff Bezos, le fondateur d'Amazon.

Kubernetes fournit de nombreuses abstractions et des API qui facilitent la construction de ces architectures de microservice découplées.

- ✓ Les *pods*, qui sont des groupes de conteneurs, peuvent regrouper des images de conteneurs développées par différentes équipes en une seule unité déployable.
- ✓ Les services Kubernetes fournissent l'équilibrage de charge, le nommage et la découverte pour isoler un microservice d'un autre.
- ✓ Les espaces de noms fournissent l'isolation et le contrôle d'accès, de sorte que chaque microservice peut contrôler le degré avec lequel il interagit avec les autres services.
- ✓ Les objets Ingress fournissent une interface facile à utiliser qui peut combiner plusieurs microservices en une seule surface d'API externalisée.

Enfin, le découplage de l'image du conteneur d'application et de la machine signifie que différents microservices peuvent cohabiter sur la même machine sans interférer les uns avec les autres, ce qui réduit la surcharge et le coût des architectures de microservices. Les fonctionnalités de contrôle d'intégrité et de mise en œuvre de Kubernetes assurent une approche cohérente du déploiement et de la fiabilité des applications, ce qui garantit que la prolifération des équipes de microservices n'entraîne pas une multiplication de différentes approches concernant le cycle de vie et les opérations de production des services.

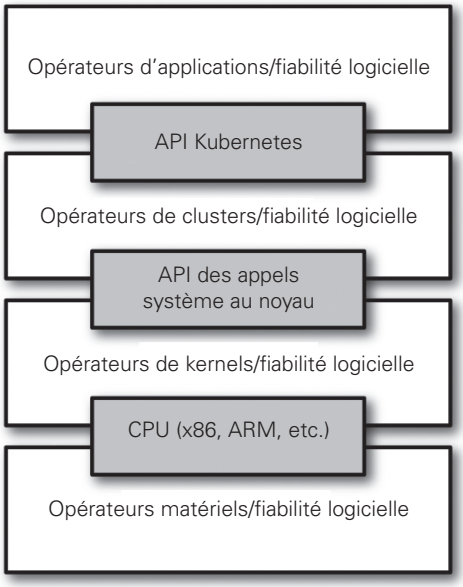
1.2.4 Séparation des problèmes en matière de cohérence et d'évolutivité

En plus de la cohérence que Kubernetes apporte à vos opérations, le découplage et la séparation des problèmes produits par la pile Kubernetes engendrent une cohérence beaucoup plus grande des niveaux inférieurs de votre infrastructure. Cela vous permet de faire évoluer les opérations d'infrastructure pour gérer de nombreuses machines avec une seule équipe réduite et concentrée sur ses objectifs. Nous avons longuement parlé du découplage du conteneur d'applications et de la machine/système d'exploitation, mais un aspect important de ce découplage est que l'API d'orchestration de conteneurs devient un contrat clair qui sépare les responsabilités de l'opérateur d'applications de celles de l'opérateur d'orchestration de clusters. On appelle ça la ligne « c'est pas mon problème ». Le développeur d'applications s'appuie sur l'accord de niveau de service (SLA) fourni par l'API de l'orchestration de conteneurs, sans se soucier des détails de la façon dont ce SLA est atteint. De la même manière, l'ingénieur responsable de la fiabilité de l'API d'orchestration des conteneurs se concentre sur la fourniture du SLA de l'API d'orchestration sans se soucier des applications qui s'exécutent au-dessus.

Ce découplage des problèmes signifie qu'une petite équipe qui exploite un cluster Kubernetes peut être responsable de centaines, voire de milliers d'équipes exécutant des applications dans ce cluster (figure 1.1). De la même manière, une petite équipe peut être responsable de l'exploitation de dizaines (ou plus) de clusters à travers le monde. Il est important de noter que le même découplage de conteneurs et de systèmes d'exploitation permet aux ingénieurs responsables de la fiabilité du système d'exploitation de se concentrer sur le SLA de l'OS des machines individuelles. Cela permet un autre degré de partage des responsabilités, les opérateurs Kubernetes

s'appuyant sur les SLA des OS, et les opérateurs du système d'exploitation ayant pour seule préoccupation de fournir ce SLA. Encore une fois, cela vous permet de gérer une petite équipe d'experts des OS capable d'exploiter une flotte de milliers de machines.

Figure 1.1 – Illustration de la manière dont les différentes équipes d'exploitation sont découplées à l'aide des API.



Bien évidemment, le fait de consacrer une petite équipe, si réduite soit-elle, à la gestion d'un système d'exploitation dépasse les capacités de nombreuses organisations. Dans ces environnements, une architecture KaaS (Kubernetes-as-a-Service) proposée par un fournisseur de cloud public est une excellente option. Comme Kubernetes est devenu de plus en plus omniprésent, la disponibilité de KaaS a également augmenté, au point qu'il est désormais proposé sur presque tous les clouds publics. Bien évidemment, l'utilisation de KaaS présente certaines limites, puisque l'opérateur prend à votre place des décisions sur la façon dont les clusters Kubernetes sont créés et configurés. Par exemple, de nombreuses plateformes KaaS désactivent les fonctionnalités alpha car elles peuvent déstabiliser le cluster géré.

En plus d'un service Kubernetes entièrement géré, il existe un écosystème florissant d'entreprises et de projets qui fournissent une assistance à l'installation et à la gestion de Kubernetes. Il existe un éventail complet de solutions qui vont de l'installation manuelle où vous vous débrouillez tout seul à un service entièrement géré.

La décision d'utiliser KaaS, ou de gérer Kubernetes vous-même, est par conséquent un choix que chaque utilisateur doit faire en fonction des compétences et des exigences de sa situation. Pour les petites organisations, KaaS fournit souvent une solution facile à utiliser qui leur permet de concentrer leur temps et leur énergie à la création du logiciel plutôt qu'à la gestion d'un cluster. Pour une organisation plus grande qui peut se permettre d'avoir une équipe dédiée à la gestion de son cluster Kubernetes, il

peut être judicieux de l'administrer par ses propres moyens, car cela permet une plus grande souplesse en termes de capacité et d'exploitation du cluster.

1.3 ABSTRACTION DE VOTRE INFRASTRUCTURE

L'objectif du cloud public est de fournir une infrastructure libre-service et facile à utiliser pour les développeurs. Toutefois, trop souvent, les API cloud sont orientées vers une mise en miroir des infrastructures demandée par les DSI (par exemple, des machines virtuelles), et non pas vers des concepts (par exemple, des applications) que les développeurs veulent exploiter. En outre, dans de nombreux cas, le cloud est livré avec des détails particuliers dans la mise en œuvre ou les services qui sont spécifiques au fournisseur de cloud. L'exploitation directe de ces API rend difficile l'exécution de votre application dans plusieurs environnements, ou la propagation entre le cloud et les environnements physiques.

Le passage aux API de conteneurs orientées application comme Kubernetes a deux avantages concrets. Tout abord, comme nous l'avons décrit précédemment, cela sépare les développeurs des machines spécifiques. Cela simplifie non seulement le rôle des services informatiques orientés machine, puisque les machines peuvent être ajoutées simplement dans l'agrégat pour faire évoluer le cluster, mais dans le contexte du cloud, cela permet aussi un degré élevé de portabilité puisque les développeurs consomment une API de niveau supérieur qui est implémentée en termes d'API d'infrastructure de cloud spécifiques.

Lorsque vos développeurs créent leurs applications en termes d'images de conteneurs et les déploient en termes d'API Kubernetes portables, le transfert de votre application entre des environnements, ou même son exécution dans des environnements hybrides, est une simple question d'envoi de configuration déclarative vers un nouveau cluster. Kubernetes a un certain nombre de plug-ins qui permettent de vous abstraire d'un cloud particulier. Par exemple, les services Kubernetes savent comment créer des équilibreurs de charge sur tous les principaux clouds publics ainsi que sur plusieurs infrastructures privées et physiques différentes. De la même manière, les objets Kubernetes `PersistentVolumes` et `PersistentVolumeClaims` peuvent être utilisés pour faire abstraction de vos applications en ignorant les implémentations de stockage spécifiques. Bien entendu, pour atteindre cette transférabilité, vous devez éviter les services gérés par le cloud (par exemple, DynamoDB d'Amazon, Cosmos DB d'Azure ou Cloud Spanner de Google), ce qui signifie que vous serez obligé de déployer et de gérer des solutions de stockage open source comme Cassandra, MySQL ou MongoDB.

En combinant tout cela et en créant par-dessus les abstractions orientées application de Kubernetes, vous vous assurez que l'effort que vous consentez dans la construction, le déploiement et la gestion de votre application la rend vraiment portable dans une grande variété d'environnements.

1.4 EFFICACITÉ

En plus des avantages en matière de gestion pour le développeur et le service informatique que procurent les conteneurs et Kubernetes, il y a aussi un bénéfice éco-

nomique concret pour l'abstraction. Comme les développeurs ne pensent plus en termes de machines, leurs applications peuvent être situées sur les mêmes machines sans impact sur les applications elles-mêmes. Cela signifie que les tâches de plusieurs utilisateurs peuvent être gérées sur moins de machines.

L'efficacité peut être mesurée par le rapport entre le travail utile effectué par une machine (ou un processus) et la quantité totale d'énergie dépensée. Lorsqu'il s'agit de déployer et de gérer des applications, de nombreux outils et processus disponibles (par exemple, des scripts bash, les mises à jour `apt` ou la gestion de configuration impérative) sont assez inefficaces. Quand on réfléchit au concept d'efficacité, il est souvent utile de penser à la fois au coût financier de l'exécution d'un serveur et au coût humain nécessaire à sa gestion.

L'exécution d'un serveur entraîne un coût basé sur l'utilisation d'énergie, les besoins de climatisation, l'espace du data center et la puissance brute de calcul. Une fois qu'un serveur est mis en rack et sous tension, le compteur commence à tourner. Tout temps mort du CPU est de l'argent gaspillé. Cela fait donc maintenant partie des responsabilités de l'administrateur système que de maintenir l'utilisation à des niveaux acceptables, ce qui nécessite une gestion permanente. C'est là que les conteneurs et le workflow Kubernetes ont un rôle à jouer. Kubernetes fournit des outils qui automatisent la distribution d'applications à travers un cluster de machines, ce qui assure des niveaux d'utilisation plus élevés qu'avec l'outillage traditionnel.

Une nouvelle augmentation de l'efficacité provient du fait que l'environnement de test d'un développeur peut être rapidement créé pour un coût réduit sous la forme d'un ensemble de conteneurs s'exécutant dans une vue personnelle d'un cluster Kubernetes partagé (en utilisant une fonctionnalité appelée *espaces de noms*). Par le passé, la mise en place d'un cluster de test pour un développeur pouvait impliquer l'activation de trois machines. Avec Kubernetes, tous les développeurs peuvent partager simplement un cluster de test unique, en agrégeant leur utilisation sur un ensemble de machines beaucoup plus petit. La réduction du nombre total de machines utilisées augmente à son tour l'efficacité de chaque système : comme plus de ressources (CPU, RAM, etc.) sur chaque machine sont utilisées, le coût global de chaque conteneur devient beaucoup plus faible.

La réduction du coût des instances de développement dans votre pile autorise des pratiques de développement dont le coût aurait été auparavant prohibitif. Par exemple, avec votre application déployée via Kubernetes, il devient concevable de déployer et de tester chaque modification de code de chaque développeur sur toute la pile.

Lorsque le coût de chaque déploiement est mesuré en fonction d'un petit nombre de conteneurs, plutôt qu'en multipliant les machines virtuelles complètes, le coût des tests est beaucoup plus faible. Si l'on revient à la valeur originale de Kubernetes, cette augmentation des tests accroît aussi la vitesse, puisque vous avez des signaux forts sur la fiabilité de votre code et sur la granularité des détails nécessaires à l'identification rapide de l'origine des problèmes.

Enfin, comme nous l'avons mentionné dans les sections précédentes, l'utilisation de la mise à l'échelle automatique pour ajouter des ressources lorsque cela est nécessaire, et les supprimer lorsqu'elles sont inutiles, peut également être utilisée pour stimuler l'efficacité globale de vos applications tout en maintenant leurs performances.

1.5 ÉCOSYSTÈME CLOUD NATIVE

Kubernetes a été conçu dès le départ pour être un environnement extensible doté d'une communauté accueillante. Ces objectifs de conception et son omniprésence dans de nombreux environnements informatiques ont engendré un écosystème vaste et dynamique d'outils et de services qui se sont développés autour de Kubernetes. Dans la lignée de Kubernetes (et de Docker, et auparavant de Linux), la plupart de ces projets sont également open source. Cela signifie qu'un développeur qui se lance n'a pas à partir de zéro. Dans les années qui ont suivi la sortie de Kubernetes, des outils pour presque toutes les tâches, de l'apprentissage automatique au développement continu et aux modèles de programmation sans serveur, ont été élaborés. En effet, dans de nombreuses situations, le défi ne consiste pas à trouver une solution, mais à décider laquelle parmi les nombreuses qui sont proposées est la mieux adaptée à la tâche. La richesse des outils de l'écosystème cloud native est elle-même devenue une motivation forte pour de nombreuses personnes d'adopter Kubernetes. Lorsque vous exploitez l'écosystème cloud native, vous pouvez utiliser des projets créés et pris en charge par la communauté pour presque chaque partie de votre système, ce qui vous permet de vous concentrer sur le développement de la logique métier et des services qui vous sont propres.

Comme pour tout écosystème open source, la principale difficulté est la variété des solutions possibles et le fait qu'il y a souvent un manque d'intégration de bout en bout. Pour atténuer cette complexité, vous pouvez bénéficier des conseils techniques de la Cloud Native Computing Foundation (CNCF ; https://www.cncf.io/). La CNCF agit en toute neutralité à l'égard des différentes entreprises du secteur et offre une expertise pour le code et la propriété intellectuelle des projets cloud native. Elle dispose de trois niveaux de maturité de projet pour vous guider dans l'adoption de projets cloud native. La majorité des projets de la CNCF sont au stade *sandbox* (bac à sable). Ce stade indique qu'un projet est encore en phase de développement précoce, et son adoption n'est pas recommandée, à moins que vous ne souhaitiez contribuer au développement du projet. L'étape suivante de la maturité est « l'incubation ». Les projets en incubation sont ceux qui ont prouvé leur utilité et leur stabilité par leur utilisation en production ; ils sont cependant encore en cours de développement et leurs communautés en phase de croissance. Alors qu'il existe des centaines de projets *sandbox*, il n'y a guère plus d'une vingtaine de projets en incubation. Le stade final des projets CNCF est *graduated*. Ces projets sont totalement matures et largement adoptés. Il n'existe que quelques projets de ce type, dont Kubernetes lui-même.

Vous pouvez également naviguer dans l'écosystème cloud native via l'intégration avec Kubernetes-as-a-Service. À ce stade, la plupart des offres KaaS proposent également des services supplémentaires via des projets open source de l'écosystème cloud native. Comme ces services sont intégrés dans des produits compatibles avec le cloud, vous pouvez être assuré que les projets sont matures et prêts pour la production.

1.6 RÉSUMÉ

Kubernetes a été construit pour changer radicalement la façon dont les applications sont créées et déployées dans le cloud. Fondamentalement, il a été conçu pour

donner aux développeurs plus de rapidité, d'efficacité et d'agilité. À l'heure actuelle, bon nombre de services et d'applications Internet que vous utilisez tous les jours fonctionnent au-dessus d'une couche Kubernetes. Vous êtes probablement déjà un utilisateur de Kubernetes qui s'ignore ! Nous espérons que ce chapitre vous aura incité à déployer vos applications à l'aide de Kubernetes. Maintenant que vous êtes convaincus, les chapitres suivants vont vous apprendre *la manière* de déployer vos applications.

Création et exécution de conteneurs

Kubernetes est une plateforme de création, de déploiement et de gestion d'applications distribuées. Ces applications se présentent sous de nombreuses formes et tailles différentes, mais en fin de compte, elles sont toutes composées d'un ou plusieurs programmes qui fonctionnent sur des machines individuelles. Ces programmes reçoivent des données, les manipulent, puis renvoient les résultats. Avant d'envisager de construire un système distribué, nous devons d'abord étudier la manière de créer les *images de conteneur d'application* qui contiennent ces programmes et constituent les pièces de notre système distribué.

Les programmes des applications sont généralement composés d'un runtime de langage, de bibliothèques et de votre code source. Dans de nombreux cas, votre application exploite des bibliothèques externes partagées comme `libc` et `libssl`. Ces bibliothèques externes sont en général fournies sous la forme de composants partagés dans le système d'exploitation que vous avez installé sur une machine spécifique.

Cette dépendance aux bibliothèques partagées pose des problèmes quand une application développée sur l'ordinateur portable d'un programmeur a une dépendance avec une bibliothèque partagée qui n'est pas disponible lorsque le programme est déployé sur le système d'exploitation de production. Même lorsque les environnements de développement et de production partagent exactement la même version du système d'exploitation, des problèmes peuvent survenir lorsque les développeurs oublient d'inclure des fichiers de ressources à l'intérieur d'un package qu'ils déploient sur la machine de production.

Les méthodes traditionnelles d'exécution de plusieurs programmes sur une seule machine requièrent que tous ces programmes utilisent conjointement les mêmes versions des bibliothèques partagées sur le système. Si les différents programmes sont développés par différentes équipes ou organisations, ces dépendances partagées ajoutent une complexité inutile et la nécessité d'une communication entre ces équipes.

Un programme ne peut s'exécuter correctement que s'il peut être déployé de manière fiable sur l'ordinateur où il doit s'exécuter. Trop souvent, les méthodes actuelles de déploiement impliquent l'exécution de scripts impératifs, qui produisent inévitablement des pannes difficiles à détecter. Le déploiement d'une nouvelle version de tout ou partie d'un système distribué est par conséquent une tâche laborieuse et difficile.

Dans le premier chapitre, nous avons souligné l'intérêt des images et des infrastructures immutables. Il s'avère que l'image d'un conteneur procure exactement les mêmes avantages. Comme nous le verrons, cette technique résout avec élégance tous les problèmes de gestion des dépendances et d'encapsulation que nous venons de décrire.

Lorsque vous travaillez avec des applications, il est souvent utile de les packager de manière à faciliter leur partage. Docker, le moteur d'exécution de conteneurs par défaut, facilite l'empaquetage d'une application qui sera ensuite envoyée dans un registre distant où elle pourra être récupérée ultérieurement par d'autres personnes. Au moment où nous écrivons ces lignes, les registres de conteneurs sont disponibles dans tous les principaux clouds publics, et un grand nombre d'entre eux propose des services permettant de créer des images. Vous pouvez également gérer votre propre registre en utilisant des systèmes open source ou commerciaux. Ces registres permettent aux utilisateurs de gérer et de déployer facilement des images privées, tandis que les services de construction d'images permettent une intégration facile avec les systèmes de livraison continue.

Dans ce chapitre, nous exploitons un exemple simple d'application que nous avons créée pour ce livre afin de vous montrer comment fonctionne ce workflow. Vous pouvez télécharger l'application sur GitHub (https://github.com/kubernetes-up-and-running/kuard).

Les images de conteneurs regroupent un programme et ses dépendances, sous un système de fichiers racine, en un seul artefact. Le format d'image de conteneur le plus populaire est le format d'image Docker, qui a été normalisé par l'Open Container Initiative pour devenir le format d'image OCI. Kubernetes prend en charge les images compatibles avec Docker et OCI via Docker et d'autres runtimes. Les images Docker incluent également des métadonnées supplémentaires qui sont utilisées par un runtime de conteneur pour démarrer l'exécution d'une instance d'application en fonction du contenu de l'image du conteneur.

Ce chapitre traite des sujets suivants :

- ✓ Comment packager une application avec le format d'image Docker.
- ✓ Comment démarrer une application à l'aide du runtime de conteneur Docker.

2. 1 IMAGES DE CONTENEURS

La plupart des gens découvrent la technologie de conteneur grâce à une image de conteneur. Une *image de conteneur* est un package binaire qui encapsule tous les fichiers nécessaires à l'exécution d'un programme à l'intérieur d'un conteneur de système d'exploitation. Vous pouvez créer une image de conteneur à partir de votre système de fichiers local ou bien télécharger une image préexistante à partir d'un *registre de conteneurs*. Dans les deux cas, une fois que l'image du conteneur est présente sur votre ordinateur, vous pouvez exécuter cette image afin de produire une application fonctionnelle à l'intérieur d'un conteneur de système d'exploitation.

Le format d'image de conteneur le plus populaire et le plus répandu est le format d'image Docker, qui a été développé par le projet open source Docker pour packager, distribuer, et exécuter des conteneurs à l'aide de la commande `docker`. Ensuite, la société Docker et d'autres entreprises ont entamé un travail pour standardiser le format d'image de conteneur au sein du projet OCI (Open Container Initiative). Bien que la norme OCI ait franchi l'étape de la version 1.0 à la mi-2017, l'adoption de ces normes se fait lentement. Le format d'image Docker, qui est toujours le standard de fait, est composé d'une série de couches de système de fichiers. Chaque couche ajoute, supprime ou modifie des fichiers de la couche précédente du système de fichiers. Il s'agit d'un exemple de système de fichiers par *overlays*. Ce système de superposition est utilisé à la fois lors de l'empaquetage de l'image et lorsque l'image est réellement utilisée. Pendant l'exécution, on compte une grande variété d'implémentations concrètes de ces systèmes de fichiers, notamment `aufs`, `overlay` et `overlay2`.

Stratification des conteneurs

Les expressions « format d'image Docker » et « images de conteneur » peuvent prêter à confusion. L'image n'est pas un fichier unique, mais plutôt une spécification pour un fichier de manifeste qui pointe vers d'autres fichiers. Le manifeste et les fichiers associés sont souvent traités par les utilisateurs comme une unité. Le niveau d'indirection permet un stockage et une transmission plus efficaces. Associée à ce format, une API permet de télécharger et d'envoyer des images vers un registre d'images.

Les images de conteneurs sont constituées d'une série de couches de systèmes de fichiers, où chaque couche hérite des couches précédentes qu'elle modifie. Pour vous aider à comprendre cela en détail, nous allons créer des conteneurs. Vous noterez que l'ordre des couches devrait aller du bas vers le haut, mais pour des raisons pédagogiques, nous présentons les choses dans le sens inverse :

```
.
└── conteneur A : il ne s'agit que d'un système d'exploitation de base, par exemple Debian
    └── conteneur B : construit à partir de #A, en ajoutant Ruby v2.1.10
    └── conteneur C : construit à partir de #A, en ajoutant Golang v1.6
```

À ce stade, nous avons trois conteneurs : A, B et C. Les conteneurs B et C *dérivent* de A et ne partagent rien en dehors des fichiers du conteneur de base. Si l'on poursuit, nous pouvons créer un conteneur à partir de B en ajoutant Ruby on Rails (version 4.2.6). Nous pouvons aussi souhaiter exploiter une ancienne application qui nécessite une version moins récente de Ruby on Rails (par exemple, la version 3.2.x). Nous pouvons alors construire une image de conteneur pour prendre en charge cette application basée sur B, avec le projet de migrer ultérieurement l'application en version 4 :

```
. (suite du schéma précédent)
└── conteneur B : construit à partir de #A, en ajoutant Ruby v2.1.10
    └── conteneur D : construit à partir de #B, en ajoutant Rails v4.2.6
    └── conteneur E : construit à partir de #B, en ajoutant Rails v3.2.x
```

Conceptuellement, chaque couche d'image de conteneur s'appuie sur la précédente. Chaque référence parente est un pointeur. Alors que l'exemple illustré ici est un ensemble simple de conteneurs, les véritables conteneurs peuvent faire partie d'un graphe orienté acyclique plus étendu.

Les images de conteneurs sont en général combinées avec un fichier de configuration de conteneur, qui fournit des instructions sur la façon de configurer l'environnement du conteneur et d'exécuter le point d'entrée de l'application. La configuration

du conteneur comprend aussi souvent des informations sur la façon de paramétrer la mise en réseau, l'isolation des espaces de noms, les contraintes de ressources (cgroups), et les restrictions `syscall` à appliquer sur une instance de conteneur en cours exécution. Le système de fichiers racine du conteneur et le fichier de configuration sont généralement regroupés en employant le format d'image Docker.

Les conteneurs se divisent en deux catégories principales :

- ✓ les conteneurs système
- ✓ les conteneurs d'application

Les conteneurs système cherchent à imiter les machines virtuelles et exécutent souvent un processus de démarrage complet. Ils incluent souvent un ensemble de services système que l'on trouve d'habitude dans une machine virtuelle, comme `ssh`, `cron` et `syslog`. Lorsque Docker était nouveau, ces types de conteneurs étaient beaucoup plus courants. Au fil du temps, ils ont fini par être considérés comme une mauvaise pratique et les conteneurs d'applications sont désormais privilégiés.

Les conteneurs d'application diffèrent des conteneurs système en ce qu'ils exécutent généralement un seul programme. Bien que l'exécution d'un seul programme par conteneur puisse paraître inutilement contraignante, cela offre un niveau parfait de granularité pour la composition évolutive d'applications, et il s'agit d'une philosophie de conception qui est fortement exploitée par les pods. Nous examinerons en détail le fonctionnement des pods au chapitre 5.

2. 2 CRÉATION D'IMAGES D'APPLICATION AVEC DOCKER

En général, les systèmes d'orchestration de conteneurs comme Kubernetes sont axés sur la création et le déploiement de systèmes distribués constitués de conteneurs d'applications. C'est la raison pour laquelle nous nous concentrerons sur les conteneurs d'application dans le reste de ce chapitre.

2.2.1 Dockerfiles

Un Dockerfile peut être utilisé pour automatiser la création d'une image de conteneur Docker.

Commençons par créer une image d'application pour un simple programme Node.js. Cet exemple serait très similaire pour de nombreux autres langages dynamiques, comme Python ou Ruby.

La plus simple des applications npm/Node/Express comporte deux fichiers : *package.json* (exemple 2.1) et *server.js* (exemple 2.2). Placez-les dans un répertoire, puis exécutez `npm install express --save` pour établir une dépendance sur Express et l'installer.

Exemple 2.1 – package.json

```
{
  "name": "simple-node",
  "version": "1.0.0",
```

```
  "description": "Exemple simple d'application pour Kubernetes Up & Running",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "author": ""
}
```

Exemple 2.2 – server.js

```
var express = require('express');

var app = express();
app.get('/', function (req, res) {
  res.send('Hello World!');
});
app.listen(3000, function () {
  console.log('Listening on port 3000!');
  console.log(' http://localhost:3000');
});
```

Pour empaqueter ceci en tant qu'image Docker, créez deux fichiers supplémentaires : *.dockerignore* (exemple 2.3) et le Dockerfile (exemple 2.4). Le Dockerfile est une recette pour la création de l'image du conteneur, tandis que .dockerignore définit l'ensemble des fichiers qui doivent être ignorés lors de la copie des fichiers dans l'image. Une description complète de la syntaxe du Dockerfile est disponible sur le site Web de Docker (https://docs.docker.com/engine/reference/builder/).

Exemple 2.3 –.dockerignore

```
node_modules
```

Exemple 2.4 – Dockerfile

```
# Démarre à partir d'une image Node.js 16 (LTS) ❶
FROM node:16

# Spécifie le répertoire à l'intérieur de l'image
# dans lequel toutes les commandes seront exécutées ❷
WORKDIR /usr/src/app

# Copie les fichiers du package et installe les dépendances ❸
COPY package*.json ./
RUN npm install
RUN npm install express

# Copie tous les fichiers de l'application dans l'image ❹
```

```
COPY . .

# Commande par défaut à exécuter au démarrage du conteneur ❺
CMD ["npm", "start" ]
```

1. Chaque Dockerfile s'appuie sur d'autres images de conteneurs. Cette ligne spécifie que nous commençons à partir de l'image `node:16` présente sur le Docker Hub. Il s'agit d'une image préconfigurée avec Node.js 16.
2. Cette ligne définit le répertoire de travail dans l'image du conteneur pour toutes les commandes suivantes.
3. Ces trois lignes initialisent les dépendances pour Node.js. D'abord, nous copions les fichiers du package dans l'image. Cela comprendra *package.json* et *package-lock.json*. La commande `RUN` exécute ensuite la commande correcte dans le conteneur pour installer les dépendances nécessaires.
4. À présent, nous copions le reste des fichiers du programme dans l'image. Cela inclura tout sauf *node_modules*, car il est exclu via le fichier *.dockerignore*.
5. Enfin, nous spécifions la commande qui doit être exécutée lorsque le conteneur est lancé.

Exécutez la commande suivante pour créer l'image Docker `simple-node` :

```
$ docker build -t simple-node .
```

Lorsque vous voulez exécuter cette image, vous pouvez le faire avec la commande suivante. Allez à http://localhost:3000 pour accéder au programme qui s'exécute dans le conteneur :

```
$ docker run --rm -p 3000:3000 simple-node
```

À ce stade, notre image `simple-node` réside dans le registre Docker local où l'image a été créée et où elle n'est accessible qu'à une seule machine. La véritable puissance de Docker provient de sa capacité à partager des images à travers des milliers de machines et la vaste communauté Docker.

2.2.2 Optimisation de la taille des images

Il y a plusieurs pièges dans lesquels peuvent tomber les gens qui commencent à travailler avec des images de conteneur, notamment le fait d'engendrer des images trop grandes. La première chose à retenir est que les fichiers qui sont supprimés par les couches successives dans le système sont en fait encore présents dans les images ; ils sont simplement inaccessibles. Examinez la situation suivante :

```
.
└── couche A : contient un gros fichier nommé 'GrosFichier'
    └── couche B : supprime 'GrosFichier'
        └── couche C : créé à partir de B, en ajoutant un binaire statique
```

Vous pensez sans doute que *GrosFichier* n'est plus présent dans cette image. Après tout, lorsque vous exécutez l'image, il n'est plus accessible. Mais en fait il est toujours présent dans la couche A, ce qui signifie que chaque fois que vous voulez envoyer ou extraire l'image, *GrosFichier* est toujours transmis par le réseau, même si vous ne pouvez plus y accéder.

Il y a un autre écueil qui concerne la mise en cache et la construction des images. Vous devez vous rappeler que chaque couche est différente et indépendante de la couche qui est en dessous. Chaque fois que vous modifiez une couche, cela change les couches qui viennent après. Si vous modifiez les couches précédentes, cela signifie qu'il faut les reconstruire, les renvoyer et les extraire pour déployer votre image vers l'environnement de développement.

Pour mieux comprendre cela, comparons ces deux images :

```
.
└── couche A : contient un système d'exploitation de base
    └── couche B : ajoute le code source server.js
        └── couche C : installe le package 'node'
et :
.
└── couche A : contient un système d'exploitation de base
    └── couche B : installe le package 'node'
        └── couche C : ajoute le code source server.js
```

Il semble évident que ces deux images se comportent de manière identique, et c'est effectivement le cas la première fois où elles sont extraites. Cependant, réfléchissez à ce qui se passe quand *server.js* est modifié. Dans le second cas, il s'agit seulement d'une modification qui doit être envoyée ou extraite, mais dans le premier cas, à la fois *server.js* et la couche fournissant le package `node` doivent être extraits et envoyés, puisque la couche `node` est dépendante de la couche *server.js*. En général, il est souhaitable d'organiser ses couches en partant de celle qui est la moins susceptible d'être modifiée jusqu'à celle qui a le plus de risque de changer afin d'optimiser la taille de l'image à envoyer ou à extraire. C'est la raison pour laquelle, dans l'exemple 2.4, nous copions les fichiers *package*.json* et installons les dépendances avant de copier le reste des fichiers du programme. Un développeur va mettre à jour et modifier les fichiers du programme beaucoup plus souvent que les dépendances.

2.2.3 Sécurité des images

En matière de sécurité, il n'y a pas de demi-mesures. Lorsque vous créez des images qui s'exécuteront à la fin dans un cluster de production Kubernetes, assurez-vous de suivre les meilleures pratiques pour le packaging et la distribution des applications. Par exemple, ne créez pas de conteneurs avec des mots de passe recyclés, cette précaution s'appliquant non seulement à la dernière couche, mais aussi à toutes les couches de l'image. Cela peut sembler paradoxal, mais un des problèmes introduits par les couches des conteneurs réside dans le fait que la suppression d'un fichier dans une

couche ne supprime pas ce fichier dans les couches précédentes. Il occupe encore de l'espace et on peut y accéder en employant les bons outils (un hacker motivé peut simplement créer une image qui se compose uniquement des couches contenant le mot de passe).

Il ne faut *jamais* mélanger les secrets et les images. Si vous faites cela, vous serez piraté, et la honte rejaillira sur votre entreprise ou votre service. Nous voulons tous passer un jour à la télévision, mais il y a sans doute de meilleures façons d'y parvenir.

En outre, comme les images de conteneur se concentrent sur l'exécution d'applications individuelles, une meilleure pratique consiste à minimiser le nombre de fichiers dans l'image de conteneur. Chaque bibliothèque supplémentaire dans une image fournit un vecteur potentiel de vulnérabilités dans votre application. En fonction du langage employé, vous pouvez réaliser des images très petites avec un ensemble très restreint de dépendances. Cet ensemble réduit garantit que votre image n'est pas exposée à des vulnérabilités présentes dans des bibliothèques qu'elle n'utilisera jamais.

2. 3 GÉNÉRATION D'IMAGES EN PLUSIEURS ÉTAPES

L'une des façons les plus courantes de générer accidentellement des images de grande taille consiste à réaliser la compilation du programme dans le cadre de la construction de l'image de conteneur de l'application. Compiler le code pendant la construction de l'image semble naturel, et c'est le moyen le plus simple de créer une image de conteneur à partir de votre programme. Le problème est que vous laissez traîner dans votre image tous les outils de développement inutiles (en plus ils sont généralement assez volumineux), ce qui va ralentir vos déploiements.

Pour résoudre ce problème, Docker a introduit les générations (*builds*) en plusieurs étapes, où au lieu de produire une seule image, un fichier Docker peut en fait en créer plusieurs. Chaque image est considérée comme une étape. Les artefacts peuvent être copiés à partir des étapes précédentes vers l'étape actuelle.

Pour illustrer cela concrètement, nous allons voir comment générer notre application d'exemple, `kuard`. Il s'agit d'une application quelque peu compliquée qui implique un frontend React.js (avec son propre processus de génération) qui est ensuite intégré dans un programme Go. Le programme Go exécute un serveur API backend avec lequel le frontend React.js interagit.

Voici à quoi peut ressembler un simple Dockerfile :

```
FROM golang:1.17-alpine

# Installe Node et NPM
RUN apk update && apk upgrade && apk add --no-cache git nodejs bash npm

# Récupère les dépendances de Go
RUN go get -u github.com/jteeuwen/go-bindata/...
RUN go get github.com/tools/godep
RUN go get github.com/kubernetes-up-and-running/kuard
```

```
WORKDIR /go/src/github.com/kubernetes-up-and-running/kuard

# Copie toutes les sources
COPY . .

# Ensemble de variables attendues par le script de génération
ENV VERBOSE=0
ENV PKG=github.com/kubernetes-up-and-running/kuard
ENV ARCH=amd64
ENV VERSION=test

# Réalise la génération. Ce script faire partie des sources.
RUN build/build.sh

CMD [ "/go/bin/kuard" ]
```

Ce Dockerfile produit une image de conteneur contenant un exécutable statique, mais il contient également tous les outils de développement Go ainsi que les outils pour générer le frontend React.js et le code source de l'application, aucun de ces éléments n'étant nécessaire pour l'application finale. L'image, si l'on compte toutes les couches, totalise plus de 500 Mo.

Examinez le Dockerfile suivant qui réalise la génération en plusieurs étapes :

```
# Étape 1 : Génération
FROM golang:1.17-alpine AS build

# Installe Node et NPM
RUN apk update && apk upgrade && apk add --no-cache git nodejs bash npm

# Récupère les dépendances de Go
RUN go get -u github.com/jteeuwen/go-bindata/...
RUN go get github.com/tools/godep

WORKDIR /go/src/github.com/kubernetes-up-and-running/kuard

# Copie toutes les sources
COPY . .

# Ensemble de variables attendues par le script de génération
ENV VERBOSE=0
ENV PKG=github.com/kubernetes-up-and-running/kuard
ENV ARCH=amd64
ENV VERSION=test

# Réalise la génération. Ce script faire partie des sources.
RUN build/build.sh
```

```
# Étape 2 : Déploiement
FROM alpine

USER nobody:nobody
COPY --from=build /go/bin/kuard /kuard

CMD [ "/kuard" ]
```

Ce Dockerfile produit deux images. La première est l'image de la génération, qui contient le compilateur Go, les outils React.js et le code source du programme. La seconde image est celle du déploiement, qui contient simplement le binaire compilé. La génération d'une image de conteneur en plusieurs étapes peut réduire la taille de votre image de conteneur finale de plusieurs centaines de mégaoctets et ainsi accélérer considérablement vos déploiements, puisque généralement, la latence de déploiement est conditionnée par les performances du réseau. L'image finale produite à partir de ce Dockerfile fait environ 20 Mo.

Ces scripts sont présents dans le dépôt `kuard` sur GitHub (https://github.com/kubernetes-up-and-running/kuard) et vous pouvez générer et exécuter cette image avec les commandes suivantes :

Note : si vous travaillez sous Windows, vous devrez peut-être corriger les fins de ligne en utilisant :

```
# Note : si vous travaillez sous Windows,vous devrez
# peut-être corriger les fins de ligne en utilisant :
# --config core.autocrlf=input
$ git clone https://github.com/kubernetes-up-and-running/kuard
$ cd kuard
$ docker build -t kuard .
$ docker run --rm -p 8080:8080 kuard
```

2.4 STOCKAGE D'IMAGES DANS UN REGISTRE DISTANT

À quoi sert une image de conteneur si elle n'est disponible que sur une seule machine ?

Kubernetes repose sur le fait que les images décrites dans un manifeste de pod sont disponibles sur chaque machine du cluster. Pour récupérer cette image sur toutes les machines du cluster, on pourrait exporter l'image `kuard` et l'importer sur chacune d'elles. Gérer des images Docker de cette façon-là serait cependant extrêmement fastidieux. Le processus manuel d'importation et d'exportation d'images Docker comporte trop de risques d'erreur humaine et il vaut mieux ne pas y penser.

La norme au sein de la communauté Docker consiste à stocker des images dans un registre distant. Il existe de très nombreuses options en matière de registre Docker, et vos choix seront guidés par vos exigences de sécurité et vos besoins de fonctionnalités collaboratives.

En règle générale, le premier choix que vous devez faire est de savoir si vous voulez utiliser un registre privé ou un registre public. Les registres publics permettent à toute personne de télécharger les images stockées dans le registre, tandis que les registres privés nécessitent une authentification pour le téléchargement. Votre choix dépendra de l'utilisation que vous voulez faire de votre conteneur.

Les registres publics sont parfaits pour partager des images dans le monde entier, parce qu'ils permettent une utilisation facile et non authentifiée des images de conteneur. Vous pouvez aisément distribuer votre logiciel sous la forme d'une image de conteneur et avoir la certitude que les utilisateurs bénéficieront partout de la même expérience.

En revanche, un registre privé est préférable pour stocker vos applications qui ne sont destinées qu'à votre service et que vous ne voulez pas partager avec tout le monde. En outre, les registres privés offrent souvent de meilleures garanties de disponibilité et de sécurité, car ils sont spécifiques à vos images plutôt que de servir à tout le monde.

Quoi qu'il en soit, pour envoyer une image, vous devez vous authentifier sur le registre. Vous pouvez généralement faire cela avec la commande `docker login`, bien qu'il y ait quelques différences avec certains registres. Dans les exemples de cet ouvrage, nous envoyons les images sur le registre Google Cloud Platform, appelé GCR (Google Container Registry) ; d'autres clouds, notamment Azure et Amazon Web Services (AWS), disposent également de registres de conteneurs hébergés. Pour les nouveaux utilisateurs hébergeant des images lisibles publiquement, le Docker Hub (https://hub.docker.com) est un excellent endroit pour débuter.

Une fois connecté, vous pouvez attribuer une balise à l'image `kuard` en ajoutant le registre Docker cible. Vous pouvez également ajouter un identifiant qui est habituellement employé pour désigner la version ou la variante de cette image, séparé par le caractère deux-points (`:`) :

```
$ docker tag kuard gcr.io/kuar-demo/kuard-amd64:blue
```

Ensuite, vous pouvez envoyer l'image `kuard` :

```
$ docker push gcr.io/kuar-demo/kuard-amd64:blue
```

Maintenant que l'image `kuard` est disponible sur un registre distant, il est temps de la déployer à l'aide de Docker. Lorsque nous avons envoyé l'image vers GCR, elle a été marquée comme étant publique, si bien qu'elle sera disponible partout sans authentification.

2.5 L'INTERFACE DU RUNTIME DE CONTENEUR

Kubernetes fournit une API pour décrire un déploiement d'application, mais il repose sur un runtime de conteneur pour configurer un conteneur d'application en utilisant des API spécifiques au conteneur qui sont natives du système d'exploitation cible. Sur un système Linux, cela signifie qu'il faut configurer cgroups et les espaces de noms. L'interface avec ce runtime de conteneur est définie par la norme CRI (Container Runtime Interface). L'API CRI est implémentée par un certain nombre de programmes

différents, notamment le programme `containerd-cri` créé par Docker et l'implémentation `cri-o` fournie par Red Hat. Lorsque vous installez les outils Docker, le runtime `containerd` est également installé et employé par le démon Docker.

À partir de la version 1.25 de Kubernetes, seuls les runtimes de conteneurs qui prennent en charge le CRI fonctionneront avec Kubernetes. Heureusement, les fournisseurs de la version gérée de Kubernetes ont rendu cette transition presque automatique pour leurs utilisateurs.

2.5.1 Exécution de conteneurs avec Docker

Dans Kubernetes, les conteneurs sont généralement lancés sur chaque nœud par un démon appelé *kubelet*. Toutefois, il est plus facile de commencer à utiliser les conteneurs à l'aide de l'outil de ligne de commande Docker. L'outil CLI Docker peut être utilisé pour déployer des conteneurs. Pour déployer un conteneur à partir de l'image `gcr.io/kuar-demo/kuard-amd64:blue`, exécutez la commande suivante :

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:blue
```

Cette commande démarre le conteneur `kuard` et mappe les ports 8080 de votre ordinateur local sur les ports 8080 du conteneur. L'option `--publish` peut être raccourcie en `-p`. Cette redirection est nécessaire car chaque conteneur obtient sa propre adresse IP, l'écoute sur *localhost* à l'intérieur du conteneur ne permettant pas l'écoute sur votre machine. Sans le transfert de port, les connexions seront inaccessibles sur votre machine. L'option `-d` spécifie que cela doit s'exécuter en arrière-plan (démon), tandis que `--name kuard` donne au conteneur un nom convivial.

2.5.2 Découverte de l'application kuard

`kuard` possède une interface Web simple, que vous pouvez charger en pointant votre navigateur à http://localhost:8080 ou via la ligne de commande :

```
$ curl http://localhost:8080
```

`kuard` propose aussi un certain nombre de fonctions intéressantes que nous découvrirons plus tard dans ce livre.

2.5.3 Limitation de l'utilisation des ressources

Docker permet aux applications d'employer moins de ressources en exposant la technologie cgroup sous-jacente fournie par le noyau Linux. Cette possibilité est également utilisée par Kubernetes pour limiter les ressources dont se sert chaque pod.

◆ *Limitation des ressources de mémoire*

L'un des principaux avantages de l'exécution d'applications dans un conteneur est la possibilité de restreindre l'utilisation des ressources. Ceci permet à plusieurs

applications de coexister sur le même matériel tout en garantissant une utilisation équitable des ressources.

Pour limiter `kuard` à 200 Mo de mémoire et 1 Go d'espace de swap, utilisez les options `--memory` et `--memory-swap` avec la commande `docker run`.

Pour arrêter et supprimer le conteneur `kuard` en cours d'utilisation :

```
$ docker stop kuard
$ docker rm kuard
```

Démarrez ensuite un autre conteneur `kuard` en utilisant les options appropriées pour limiter l'utilisation de la mémoire :

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  gcr.io/kuar-demo/kuard-amd64:blue
```

Si le programme dans le conteneur utilise trop de mémoire, il sera interrompu.

◆ *Limitation des ressources CPU*

Le CPU est aussi une ressource critique sur une machine. Vous pouvez restreindre l'utilisation du CPU à l'aide de l'option `--cpu-shares` avec la commande `docker run` :

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  gcr.io/kuar-demo/kuard-amd64:blue
```

2.6 NETTOYAGE

Une fois que vous en avez terminé avec une image, vous pouvez la supprimer avec la commande `docker rmi` :

```
docker rmi <nom de balise>
```

ou

```
docker rmi <identificateur d'image>
```

Les images peuvent être supprimées en utilisant leur nom de balise (par exemple, gcr.io/kuar-demo/kuardamd64:blue) ou leur identificateur d'image. Comme pour toutes les valeurs d'identificateur de l'outil `docker`, l'identificateur d'image peut être raccourci tant qu'il reste unique. En général, les trois ou quatre premiers caractères de l'identificateur suffisent.

Il est important de noter que, sauf si vous supprimez explicitement une image, elle demeure à jamais sur votre système, *même* si vous créez une nouvelle image avec un nom identique. La création de cette nouvelle image déplace simplement la balise sur la nouvelle image ; cela ne supprime pas ou ne remplace pas l'ancienne image.

Cela a pour conséquence qu'au fur et à mesure que vous créez de nouvelles images, vous créez souvent de nombreuses images différentes qui occupent inutilement de l'espace sur votre ordinateur. Pour voir les images présentes sur votre machine, vous pouvez utiliser la commande `docker images`. Vous pouvez ensuite supprimer les balises que vous n'utilisez plus.

Docker fournit un outil appelé `docker system prune` pour effectuer un nettoyage général. Cela supprimera tous les conteneurs arrêtés, toutes les images non étiquetées et toutes les couches d'images inutilisées mises en cache dans le cadre du processus de génération. Bien entendu, il faut employer cet outil avec beaucoup de prudence.

Une approche un peu plus sophistiquée consiste à mettre en place un job `cron` pour exécuter un garbage collector d'images. Par exemple, vous pouvez facilement exécuter `docker system prune` sous la forme d'un job `cron` récurrent, une fois par jour ou bien une fois par heure, selon le nombre d'images que vous créez.

2.7 RÉSUMÉ

Les conteneurs d'application fournissent une bonne abstraction pour les applications et, lorsqu'ils sont packagés dans des images au format Docker, les applications deviennent faciles à créer, à déployer et à distribuer. Les conteneurs offrent aussi une isolation entre les applications fonctionnant sur le même ordinateur, ce qui permet d'éviter les conflits de dépendance.

Dans les chapitres suivants, nous allons voir comment la possibilité de monter des répertoires externes permet non seulement d'exécuter des applications *stateless* dans un conteneur, mais aussi des applications comme MySQL ou bien d'autres applications qui génèrent beaucoup de données.