UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Cloud Native JAVA QUARKUS

Programme

1. Introduction à Cloud Native

- Définition du Cloud Native
- Pourquoi adopter le Cloud Native ?
- Principes et avantages du Cloud Native
- Philosophie DevSecOps

Programme

2. Runtimes éphémères

- Comprendre les concepts de runtime éphémère
- Utilisation des runtimes éphémères pour des applications évolutives et hautement disponibles
- Sécurité des runtimes éphémères

Programme

3. Modèles d'architectures Cloud Native

- Concept API Rest et Microservices
- Conteneurs
- Orchestration de conteneurs avec Kubernetes
- Serverless

Programme

4. Déploiement et The Twelve-Factor App

- Processus de déploiement Cloud Native
- Concepts clés du The Twelve-Factor Apppour le Cloud Native
- Stratégies de déploiement avancées
  - Rollbacks automatiques
  - Déploiements progressifs
  - Déploiements en rolling update
  - Déploiements en blue-green

Programme

5. Gérer ses données

- Bases de données Cloud Native
- Stockage Cloud Native
- La sécurité liée à la gestion des données Cloud Native

Programme

6. Observabilité et troubleshooting

- Principes de l'observabilité Cloud Native
- Techniques de monitoring et d'analyse des logs
- Stratégies de résolution des problèmes

Programme

7. Élasticité et résilience

- Introduction à l'élasticité et la résilience
- Gestion de la charge

Programme

8. Introduction à la conteneurisation Quarkus et GraalVM

- Comprendre les différentes étapes du cycle de vie de l'application Quarkus
- Configurer une application Quarkus pour une exécution conteneurisée

Programme

9. Déploiement d'une application Quarkus conteneurisée

- Créer une image Docker de l'application Quarkus
- Déployer l'application Quarkus sur une plateforme de conteneurisation telle que Kubernetes

Programme

10. Observation d'une application Java conteneurisée

- Comprendre les principaux défis d'observation d'une application Java conteneurisée
- Utiliser les outils d'observation pour surveiller une application Quarkus conteneurisée
- Configurer des alertes pour être averti en cas de problèmes

Programme

11. Utilisation des outils de cloud pour tracer les performances et résoudre des problèmes

- Comprendre les outils de surveillance des performances disponibles dans le cloud (AWS CloudWatch, Google Cloud Monitoring et Azure Monitor…)
- Utiliser ces outils pour surveiller les performances d'une application Quarkus conteneurisée
- Identifier les problèmes de performance et les résoudre en utilisant les informations fournies par ces outils

Programme

12. Feature flipping

- Comprendre les avantages du feature flipping pour les applications conteneurisées
- Utiliser une bibliothèque de feature flipping pour gérer les fonctionnalités d'une application Quarkus conteneurisée
- Mettre en œuvre des stratégies de déploiement pour les nouvelles fonctionnalités
- Identifier les fonctionnalités à activer ou désactiver en utilisant des "clés" :
  - Configurer les fonctionnalités pour le feature flipping
  - Déployer la nouvelle fonctionnalité graduellement : Activer la nouvelle fonctionnalité pour un petit groupe d'utilisateurs et surveiller les performances et les retours d'expérience.
  - Gérer les erreurs

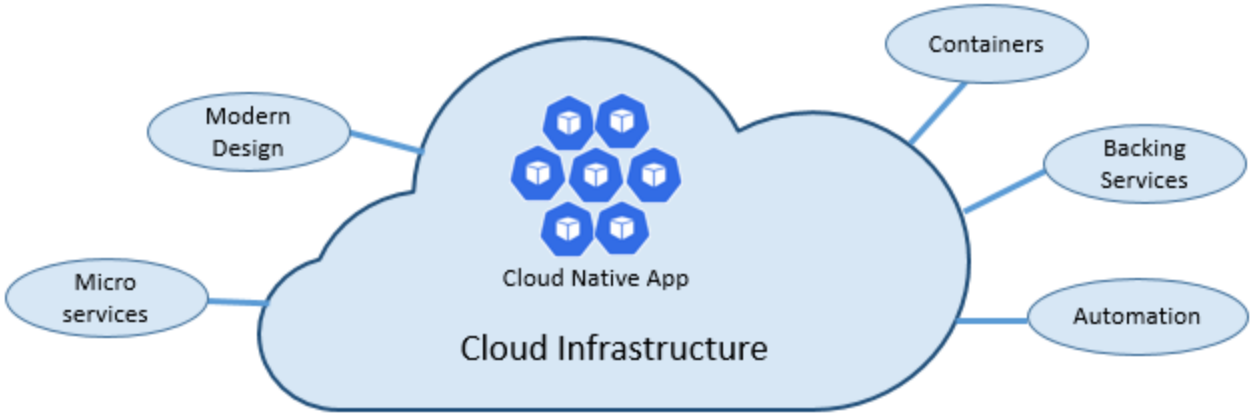
Introduction à cloud Native

Définition du cloud Native

- Cloud Native est une approche de la conception et de la construction d'applications qui exploite les avantages de l'infrastructure cloud.
- Elle est associée à des technologies comme les microservices, les conteneurs, les orchestrateurs comme Kubernetes, et les infrastructures cloud continues et automatiquement mises à jour.

Introduction à cloud Native

Définition du cloud Native



Introduction à cloud Native

Pourquoi adopter le Cloud Native ?

- Le cloud native permet aux entreprises de créer et de déployer des applications de manière plus rapide et efficace.
- Le cloud native offre une évolutivité, une résilience et une portabilité optimales, tout en permettant une innovation constante et une gestion plus efficace des coûts d'infrastructure.

Introduction à cloud Native

Principes et avantages du Cloud Native

1. Évolutivité et agilité : Les applications cloud native sont conçues pour tirer parti de la flexibilité et de l'évolutivité du cloud. Cela signifie que votre application peut s'adapter rapidement aux demandes changeantes, ce qui peut être particulièrement utile pour gérer les pics de demande.
2. Rapidité de mise sur le marché : Grâce à des pratiques comme l'intégration continue/déploiement continu (CI/CD), les entreprises peuvent livrer des fonctionnalités plus rapidement et plus fréquemment.
3. Résilience : Les applications cloud native sont conçues pour être résilientes face aux pannes. Si une partie de votre application tombe en panne, le reste de l'application peut continuer à fonctionner sans interruption.

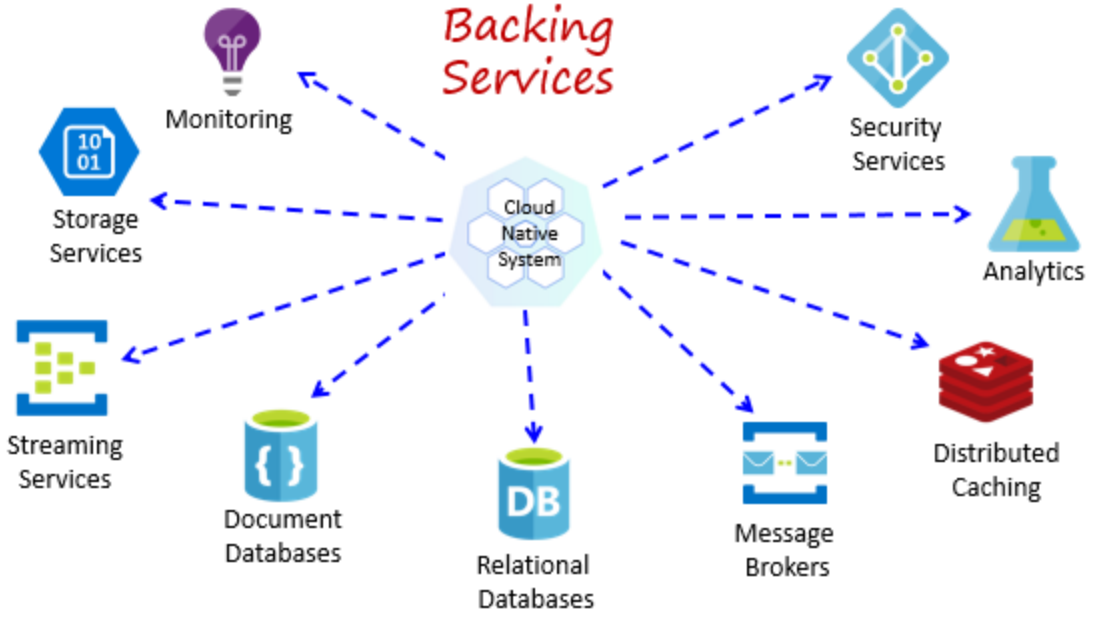
Introduction à cloud Native

Principes et avantages du Cloud Native

4. Optimisation des coûts : Le cloud computing permet d'optimiser les coûts car vous ne payez que pour les ressources que vous utilisez. De plus, la réduction de l'infrastructure physique peut entraîner une réduction des coûts d'exploitation.
5. Innovation : Le cloud native favorise l'innovation en facilitant l'expérimentation et en réduisant le coût de l'échec. Les équipes peuvent tester rapidement de nouvelles idées et fonctionnalités, et si elles ne fonctionnent pas, elles peuvent les supprimer sans avoir à faire face à des coûts importants.
6. Portabilité et indépendance du fournisseur : Les applications cloud native sont souvent construites en utilisant des technologies ouvertes, ce qui signifie qu'elles peuvent être facilement déplacées d'un environnement à un autre, ou d'un fournisseur de cloud à un autre.

Introduction à cloud Native

Principes et avantages du Cloud Native



Introduction à cloud Native

Philosophie DevSecOps

- DevSecOps est une philosophie de développement qui intègre la sécurité dans toutes les phases du cycle de développement de logiciel. C'est un prolongement de l'approche DevOps, qui combine les équipes de développement (Dev) et d'exploitation (Ops) pour favoriser une collaboration plus étroite et une livraison plus rapide des logiciels.
- En ajoutant la "Sec" (pour "Sécurité") à DevOps, l'idée est de rendre la sécurité partie intégrante du cycle de vie du développement de logiciels, plutôt que de la traiter comme une considération après coup. Cela signifie que la sécurité est prise en compte dès le début du développement d'une application, et tout au long de son cycle de vie, y compris la conception, le développement, les tests, le déploiement et la maintenance.

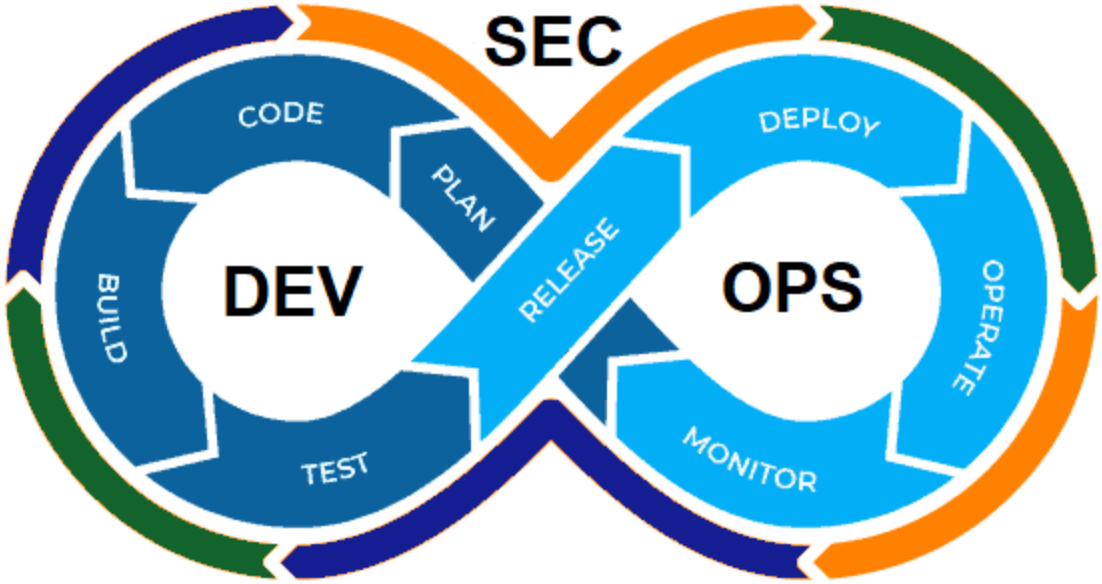
Introduction à cloud Native

Philosophie DevSecOps

- Cela a plusieurs avantages.
  - Conduire à une détection plus précoce des vulnérabilités et des problèmes de sécurité, ce qui peut réduire les coûts et les risques associés.
  - Améliorer l'efficacité du développement de logiciels en réduisant le besoin de retravailler ou de corriger les problèmes de sécurité après le déploiement.
  - Intégrer la sécurité dans toutes les phases du développement, cela encourage une culture de la sécurité au sein de l'organisation, où tous les membres de l'équipe comprennent l'importance de la sécurité et sont responsables de la maintenir.

Introduction à cloud Native

Philosophie DevSecOps



Runtimes éphémères

Comprendre les concepts de runtime éphémère

- Dans l'architecture Cloud Native, l'idée d'un "runtime éphémère" est un concept important, en particulier lorsqu'il est associé à des environnements comme les conteneurs et les serveurs sans serveur (serverless).
- Un "runtime éphémère" se réfère à une instance d'une application qui est créée dynamiquement pour répondre à une demande spécifique et qui est détruite dès que la demande a été traitée. C'est l'opposé d'un "runtime persistant", où une instance d'une application tourne en continu et traite une série de demandes au fil du temps.

Runtimes éphémères

Comprendre les concepts de runtime éphémère

1. Éphémère : L'éphémérité se réfère à la nature temporaire de l'instance de l'application. Dans un environnement éphémère, les instances sont créées et détruites à la volée, souvent en réponse à des demandes spécifiques. Cela diffère des environnements persistants, où les instances continuent à exister même lorsqu'elles ne sont pas utilisées.
2. Scalabilité : Un des principaux avantages des runtimes éphémères est leur capacité à monter en charge rapidement. Comme chaque demande est traitée par une nouvelle instance de l'application, il est possible de traiter un grand nombre de demandes simultanément en lançant simplement plus d'instances. De plus, comme les instances sont détruites lorsqu'elles ne sont plus nécessaires, cela permet également de réduire les ressources inutilisées, ce qui peut être plus rentable.
3. Isolation : Chaque instance d'un runtime éphémère est isolée des autres. Cela signifie que si une instance rencontre une erreur ou un problème de sécurité, cela n'affecte pas les autres instances. Cela contribue à la résilience et à la sécurité de l'application.

Runtimes éphémères

Comprendre les concepts de runtime éphémère

4. Statelessness (Sans état) : Les runtimes éphémères sont généralement sans état. Cela signifie qu'ils ne conservent aucune information d'une demande à l'autre. Toute information qui doit être persistée doit être stockée dans un service externe, comme une base de données ou un système de fichiers.
5. Event-driven (Piloté par les événements) : Les runtimes éphémères sont souvent utilisés dans des architectures pilotées par les événements. Dans ce type d'architecture, une nouvelle instance de l'application est créée en réponse à un événement spécifique, comme une demande HTTP, un message dans une file d'attente, ou un événement de base de données.

Runtimes éphémères

Sécurité des runtimes éphémères

1. Isolation : Dans le contexte des conteneurs et des environnements sans serveur, l'isolation est un facteur de sécurité crucial. Chaque instance de fonction ou de conteneur doit être isolée des autres pour empêcher les attaques de type "cross-container" ou "cross-function". Les plateformes modernes de conteneurs et de fonctions sans serveur prennent en charge cette isolation à différents niveaux, y compris le réseau, le système de fichiers et le processus.
2. Réduction de la surface d'attaque : Les environnements d'exécution éphémères ont généralement une surface d'attaque plus petite car ils sont créés pour exécuter une seule tâche ou fonction, et ils disparaissent une fois cette tâche accomplie. Cependant, il est important de s'assurer que le conteneur ou la fonction ne contient que les dépendances nécessaires pour accomplir cette tâche, et rien de plus. Tout code ou dépendance supplémentaire pourrait augmenter la surface d'attaque.
3. Gestion des secrets : Les environnements d'exécution éphémères peuvent avoir besoin d'accéder à des secrets, tels que des clés d'API ou des mots de passe. Ces secrets doivent être gérés de manière sécurisée. Ils ne doivent jamais être inclus dans l'image du conteneur ou du code de la fonction, mais plutôt fournis via un mécanisme sécurisé comme les variables d'environnement ou, encore mieux, un service de gestion des secrets.

Runtimes éphémères

Sécurité des runtimes éphémères

4. Vérifications de sécurité : Il est important d'effectuer des vérifications de sécurité sur le code qui s'exécute dans les environnements d'exécution éphémères. Cela peut inclure des analyses de sécurité statiques du code, l'utilisation de conteneurs de confiance et l'application de mises à jour de sécurité pour les dépendances du conteneur ou du runtime.
5. Limites de ressources : Les limites de ressources peuvent être appliquées pour éviter les attaques par déni de service qui pourraient essayer de consommer toutes les ressources du système en créant un grand nombre d'instances de conteneurs ou de fonctions.
6. Traçabilité et surveillance : Enfin, il est important de surveiller les environnements d'exécution éphémères pour détecter les activités suspectes ou malveillantes. Cela peut être réalisé en recueillant et en analysant les logs et les métriques d'exécution, ainsi que par la mise en place de systèmes de détection des intrusions.

Modèles d'architectures Cloud Native

Concept API Rest et Microservices

1. API REST

- REST (Representational State Transfer) est un style d'architecture qui définit un ensemble de contraintes pour créer des services web. Les API REST utilisent les méthodes standard du protocole HTTP (comme GET, POST, PUT, DELETE) pour créer, lire, mettre à jour et supprimer les ressources.
- Les API REST sont sans état, ce qui signifie que chaque requête doit contenir toutes les informations nécessaires pour comprendre et traiter la requête. Cela rend les API REST bien adaptées au cloud, car elles peuvent être facilement échelonnées en ajoutant simplement plus de serveurs pour traiter les requêtes.
- Les API REST sont basées sur des ressources, où chaque ressource est identifiée par une URL unique. Les clients interagissent avec ces ressources en utilisant les méthodes HTTP. Cela rend les API REST simples à utiliser et à comprendre, ce qui peut faciliter l'intégration avec d'autres services et applications.

Modèles d'architectures Cloud Native

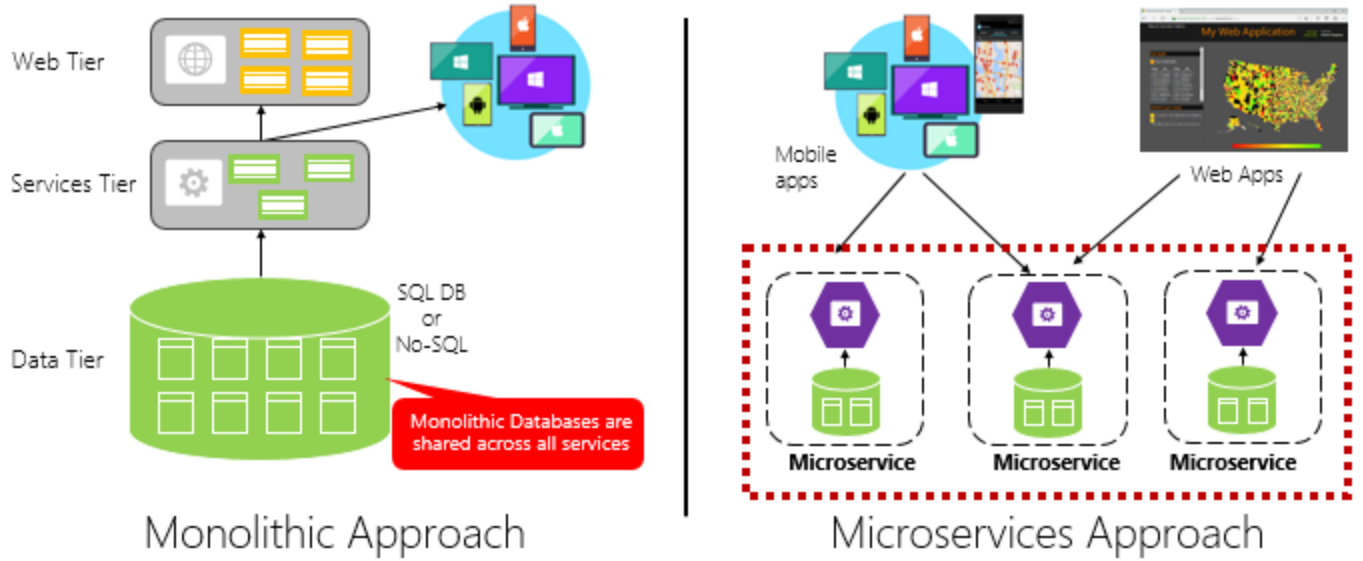
Concept API Rest et Microservices

2. Microservices

- Le concept de microservices est une approche de développement de logiciels qui structure une application comme un ensemble de services faiblement couplés et indépendants. Chaque microservice est une petite application qui fonctionne de manière autonome et communique avec les autres microservices via des API, généralement RESTful.
- Dans le cadre du cloud natif, les microservices peuvent être déployés, mis à l'échelle et mis à jour indépendamment les uns des autres. Cela offre une flexibilité considérable et permet une livraison et une mise à jour continues des différentes parties d'une application. De plus, si un microservice tombe en panne, cela n'affecte pas directement les autres microservices.
- Cette architecture peut également faciliter le travail des équipes de développement, car chaque équipe peut se concentrer sur un seul microservice à la fois, en utilisant les technologies et les langages de programmation qui conviennent le mieux à ce service particulier.

Modèles d'architectures Cloud Native

Concept API Rest et Microservices



Modèles d'architectures Cloud Native

Conteneurs

- Les conteneurs sont une technologie clé dans le paysage du cloud natif.
- Ils sont utilisés pour emballer et isoler les applications avec leurs dépendances entières, y compris le système d'exploitation, les bibliothèques système, les scripts, etc.
- Cela permet d'assurer que l'application fonctionne de manière cohérente et fiable dans n'importe quel environnement, que ce soit en développement, en test ou en production.
- Un conteneur est plus léger qu'une machine virtuelle traditionnelle car il partage le système d'exploitation de l'hôte et n'a pas besoin de son propre système d'exploitation.
- Cela rend les conteneurs très efficaces en termes d'utilisation des ressources système et de temps de démarrage.

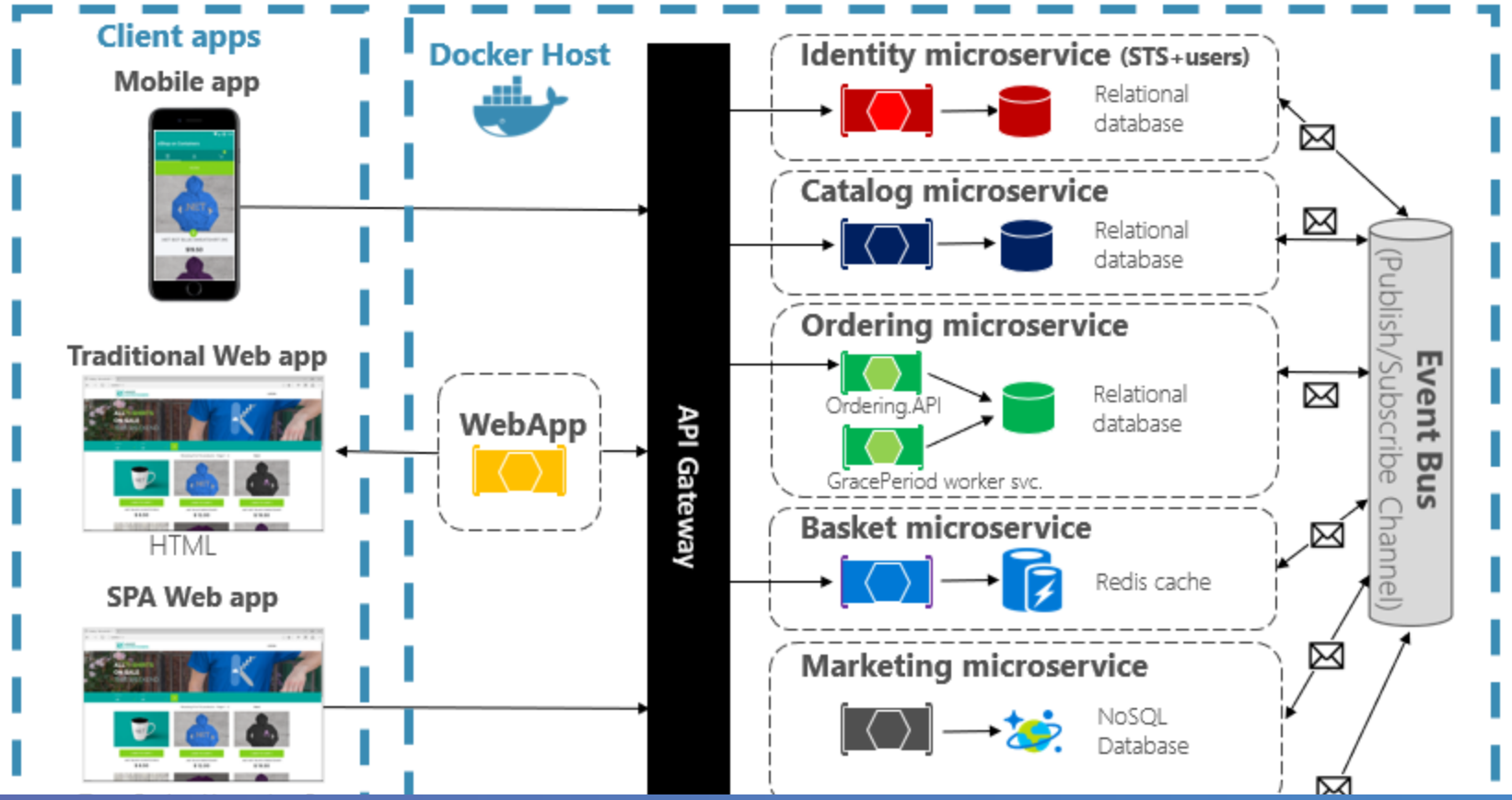
Modèles d'architectures Cloud Native

Conteneurs

1. Évolutivité : Les conteneurs peuvent être démarrés et arrêtés rapidement, ce qui facilite leur mise à l'échelle en fonction de la demande. Si la demande pour une application augmente, plus de conteneurs peuvent être lancés pour gérer cette demande.
2. Déploiement continu et livraison continue (CI/CD) : Les conteneurs sont parfaits pour les pipelines CI/CD car ils permettent de déployer rapidement de nouvelles versions d'une application.
3. Isolation : Chaque conteneur fonctionne de manière isolée. Cela signifie qu'un conteneur n'a pas d'effet sur les autres conteneurs et peut avoir ses propres configurations système et logicielles.
4. Portabilité : Les conteneurs garantissent que l'application fonctionne de manière identique dans tous les environnements. Cela permet de déplacer facilement les applications d'un système à un autre, ou d'un cloud à un autre.

Modèles d'architectures Cloud Native

Conteneurs



Modèles d'architectures Cloud Native

Orchestration de conteneurs avec Kubernetes

- Dans le contexte du cloud natif, l'orchestration de conteneurs est essentielle pour gérer les opérations de grands nombres de conteneurs et de services. Kubernetes, souvent appelé K8s, est l'un des systèmes d'orchestration de conteneurs les plus populaires et les plus puissants.
- Kubernetes facilite le déploiement, la mise à l'échelle et la gestion de groupes de conteneurs.

1. Pods : Dans Kubernetes, le pod est la plus petite unité déployable qui peut être créée et gérée. Un pod représente un ou plusieurs conteneurs qui sont déployés ensemble sur le même hôte et qui partagent le même réseau et le même espace de stockage.
2. Services : Un service est une abstraction qui définit un ensemble logique de pods et une politique d'accès à ceux-ci. Les services permettent de découpler les dépendances réseau entre les consommateurs de services et les pods qui les fournissent.

Modèles d'architectures Cloud Native

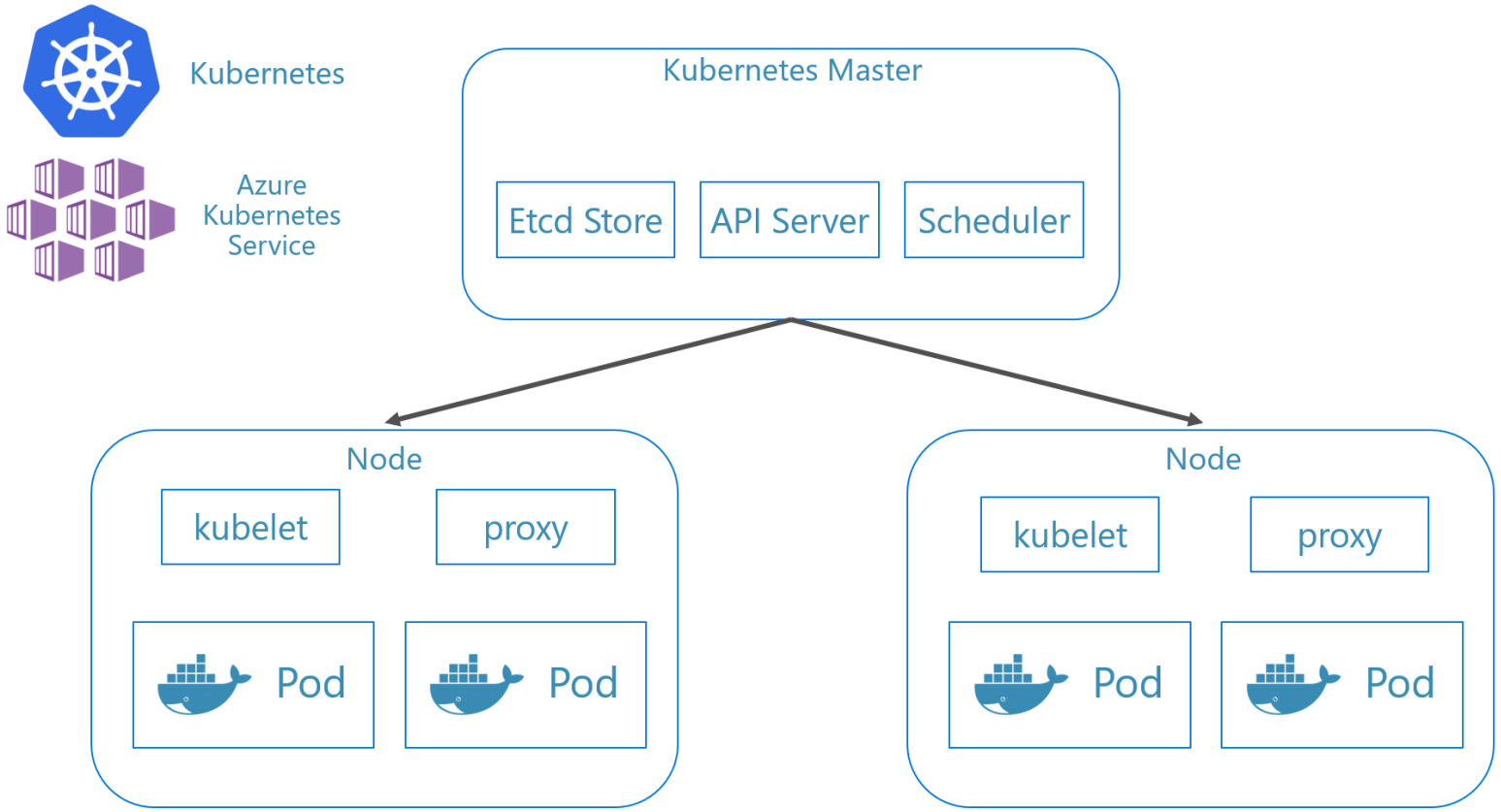
Orchestration de conteneurs avec Kubernetes

3. Déploiements et ReplicaSets : Les déploiements et les ReplicaSets garantissent la disponibilité et la mise à l'échelle des pods. Ils permettent de définir combien de réplicas d'un pod doivent être en cours d'exécution à un moment donné et gèrent le processus de mise à jour des pods.
4. Configuration et gestion des secrets : Kubernetes permet de gérer les configurations et les secrets (comme les mots de passe, les clés API, les jetons, etc.) de manière sécurisée et flexible.

Kubernetes peut fonctionner sur n'importe quel environnement cloud public (comme Google Cloud, AWS, Azure), dans des clouds privés, dans des environnements hybrides et même sur des ordinateurs portables. Cela permet une portabilité et une flexibilité considérables pour le déploiement d'applications.

Modèles d'architectures Cloud Native

Orchestration de conteneurs avec Kubernetes



Modèles d'architectures Cloud Native

Serverless

Le serverless (ou sans serveur) est un modèle de calcul dans lequel le fournisseur de cloud gère automatiquement l'infrastructure sous-jacente, permettant aux développeurs de se concentrer uniquement sur l'écriture du code pour leurs applications.

1. Évolutivité automatique : Dans un environnement serverless, les ressources sont automatiquement mises à l'échelle en fonction de la demande. Si votre application a besoin de plus de ressources pour traiter une augmentation du trafic, le fournisseur de cloud ajoutera automatiquement ces ressources. De même, si le trafic diminue, le fournisseur réduira les ressources utilisées, ce qui peut entraîner une réduction des coûts.
2. Facturation à l'utilisation : Dans un modèle serverless, vous ne payez que pour le temps d'exécution de votre code. Si votre code n'est pas en cours d'exécution, vous n'avez pas à payer pour des ressources inutilisées. Cela peut être plus rentable que de payer pour une infrastructure qui est constamment en marche, même lorsqu'elle n'est pas utilisée.

Modèles d'architectures Cloud Native

Serverless

3. Absence de gestion de l'infrastructure : Avec le serverless, les développeurs n'ont pas à se soucier de la gestion de l'infrastructure. Le fournisseur de cloud s'occupe de toutes les tâches d'infrastructure, comme le provisionnement de serveurs, la mise à jour du système d'exploitation, le patching de sécurité, etc.

Les fonctions en tant que service (FaaS) sont un aspect courant du serverless, où les développeurs peuvent déployer des morceaux individuels de code (appelés fonctions) qui s'exécutent en réponse à des événements spécifiques. Les services comme AWS Lambda, Google Cloud Functions, et Azure Functions sont des exemples de FaaS.

Déploiement et The Twelve-Factor App

Processus de déploiement Cloud Native

- Le déploiement d'applications dans le cloud natif implique généralement une combinaison de techniques et de technologies qui maximisent l'agilité et minimisent les temps d'arrêt.

1. Développement de l'application : Les applications sont souvent construites en utilisant une architecture de microservices, où chaque service est développé et déployé de manière indépendante. Les développeurs peuvent choisir le langage de programmation et le framework qui conviennent le mieux à chaque service.
2. Conteneurisation : Une fois le code de l'application écrit et testé, il est emballé dans un conteneur à l'aide d'outils comme Docker. Les conteneurs incluent le code de l'application ainsi que toutes les dépendances nécessaires pour exécuter le code. Cela garantit que l'application fonctionnera de manière identique dans tous les environnements.
3. Orchestration de conteneurs : Les conteneurs sont déployés sur un orchestrateur de conteneurs comme Kubernetes. L'orchestrateur gère l'infrastructure sous-jacente, détermine où et quand exécuter chaque conteneur, équilibre la charge de travail entre les conteneurs, surveille l'état de santé des conteneurs, et redémarre les conteneurs qui échouent.

Déploiement et The Twelve-Factor App

Processus de déploiement Cloud Native

4. Intégration continue et livraison continue (CI/CD) : Les pipelines CI/CD sont utilisés pour automatiser le processus de déploiement. Lorsqu'un développeur apporte des modifications au code de l'application, le pipeline CI/CD automatise le processus de test du code, de construction du conteneur, et de déploiement du conteneur sur l'orchestrateur.
5. Monitoring et Logging : Les outils de surveillance et de journalisation sont utilisés pour collecter et analyser les performances et les journaux d'activité de l'application. Ces informations peuvent aider à identifier et résoudre les problèmes, à comprendre comment l'application est utilisée, et à optimiser les performances et la fiabilité.
6. Gestion des configurations et des secrets : Les informations de configuration et les secrets (comme les clés d'API et les mots de passe) sont gérés de manière sécurisée et flexible, souvent en utilisant des outils ou des services fournis par l'orchestrateur de conteneurs.

Déploiement et The Twelve-Factor App

Concepts clés du The Twelve-Factor App pour le Cloud Native

"The Twelve-Factor App" est une méthodologie développée par les ingénieurs d'Heroku pour construire des applications logicielles-as-a-service qui sont évolutives, portables et faciles à gérer. Bien que ce ne soit pas spécifique au cloud natif, il est fortement lié aux meilleures pratiques du cloud natif. Voici les douze facteurs :

1. Codebase (Base de code) : Il doit y avoir exactement une base de code par application avec le code source suivi dans un système de contrôle de version, comme Git. Plusieurs déploiements peuvent être issus de la même base de code.
2. Dependencies (Dépendances) : Les dépendances d'application doivent être déclarées explicitement et isolées. En d'autres termes, ne vous appuyez jamais sur l'existence présumée de packages système.
3. Config (Configuration) : Stockez la configuration dans l'environnement. Cela signifie que toutes les informations spécifiques à l'environnement ne doivent pas être stockées dans le code, mais fournies par l'environnement d'exécution.

Déploiement et The Twelve-Factor App

Concepts clés du The Twelve-Factor App pour le Cloud Native

4. Backing Services (Services auxiliaires) : Traitez les services auxiliaires comme des ressources attachées. Cela signifie que toutes les ressources de service, comme les bases de données, les files d'attente, le stockage, etc., sont des ressources attachées qui peuvent être remplacées sans modification du code.
5. Build, release, run (Construire, livrer, exécuter) : Séparez strictement les étapes de build et d'exécution. Cela signifie que le même code doit être exécuté dans tous les environnements, de la phase de test à la production.
6. Processes (Processus) : Exécutez l'application comme un ou plusieurs processus sans état. Toute donnée qui doit survivre à un redémarrage de processus doit être stockée dans un service auxiliaire à état.
7. Port Binding (Liaison de port) : L'application web doit être autonome, c'est-à-dire capable de se lier à un port pour servir les requêtes, sans dépendre d'un serveur web séparé.

Déploiement et The Twelve-Factor App

Concepts clés du The Twelve-Factor App pour le Cloud Native

8. Concurrency (Concurrence) : Échelonnez les applications par le processus. C'est-à-dire que chaque processus est une première entité de citoyen qui peut être démarré ou arrêté à tout moment.
9. Disposability (Jetabilité) : Maximisez la robustesse avec des démarrages rapides et des arrêts gracieux. Les processus doivent pouvoir être démarrés ou arrêtés à tout moment sans affecter les autres processus ou services.
10. Dev/prod parity (Parité dev/prod) : Gardez le développement, le staging et la production aussi similaires que possible. Cela signifie que tous les environnements doivent être configurés de manière similaire pour éviter les bugs spécifiques à l'environnement.

Déploiement et The Twelve-Factor App

Concepts clés du The Twelve-Factor App pour le Cloud Native

11. Logs (Journaux) : Traitez les journaux comme des flux d'événements. Les applications doivent produire des journaux pour aider à surveiller leur comportement en temps réel, pour l'analyse historique ou pour la détection des problèmes.
12. Admin processes (Processus d'administration) : Exécutez les tâches d'administration ou de maintenance comme des processus ponctuels. Ces tâches sont exécutées dans un environnement identique à celui de l'application régulière, mais sont souvent initiées manuellement.

Déploiement et The Twelve-Factor App

Stratégies de déploiement avancées

- Dans le contexte du cloud natif, les stratégies de déploiement avancées sont utilisées pour minimiser les temps d'arrêt pendant les mises à jour de l'application et pour limiter l'impact des problèmes qui peuvent survenir lors du déploiement de nouvelles versions

1. Rollbacks automatiques : Cette stratégie est utilisée pour revenir automatiquement à la version précédente de l'application en cas de détection d'une erreur lors d'un déploiement. Les erreurs peuvent être détectées par une variété de signaux, comme les échecs de tests automatisés, les alertes de surveillance, ou l'augmentation du taux d'erreurs de l'application. Les rollbacks automatiques peuvent aider à minimiser l'impact des problèmes de déploiement sur les utilisateurs finaux.
2. Déploiements progressifs : Cette stratégie, également connue sous le nom de déploiement en canari, implique de déployer la nouvelle version de l'application à un sous-ensemble limité d'utilisateurs avant de la déployer à tous les utilisateurs. Cela permet de tester la nouvelle version en production avec un impact limité. Si des problèmes sont détectés, le déploiement peut être arrêté ou annulé avant qu'il n'affecte tous les utilisateurs.

Déploiement et The Twelve-Factor App

Stratégies de déploiement avancées

3. Déploiements en rolling update : Cette stratégie consiste à déployer progressivement la nouvelle version de l'application, en remplaçant un conteneur, un pod ou un serveur à la fois. Cela permet d'éviter les temps d'arrêt, mais peut également prolonger le temps nécessaire pour le déploiement complet. Si des problèmes sont détectés pendant le déploiement, il peut être arrêté, limitant ainsi l'impact à une petite partie de l'application.
4. Déploiements en blue-green : Cette stratégie implique d'avoir deux environnements de production parallèles, appelés "bleu" et "vert". À tout moment, l'un de ces environnements est actif, servant du trafic en production. Lors d'un déploiement, la nouvelle version de l'application est déployée sur l'environnement inactif. Une fois le déploiement terminé et les tests réussis, le trafic est basculé de l'environnement actif vers l'environnement inactif.

Gérer ses données

Bases de données Cloud Native

- Les bases de données cloud native sont conçues pour exploiter pleinement le potentiel du cloud computing. Elles sont conçues pour la scalabilité, l'élasticité, et l'automatisation, ce qui les rend parfaitement adaptées aux architectures de microservices utilisées en développement cloud natif.

1. Scalabilité : Les bases de données cloud native sont conçues pour supporter les volumes de données massifs générés par les applications modernes. Elles permettent une montée en charge horizontale, ce qui signifie que vous pouvez ajouter plus de ressources à votre base de données (c'est-à-dire des nœuds de serveur) pour augmenter les performances au fur et à mesure de l'augmentation du volume de données.
2. Élasticité : Ces bases de données peuvent s'étendre et se rétracter en fonction des besoins de l'application. Elles peuvent donc gérer efficacement les pics de charge et les périodes de faible utilisation, ce qui se traduit par une utilisation plus efficace des ressources et des économies de coûts.
3. Résilience : Les bases de données cloud native sont conçues pour être résilientes face aux défaillances. Elles sont souvent répliquées sur plusieurs régions, zones de disponibilité ou nœuds pour garantir la disponibilité et la durabilité des données, même en cas de défaillance d'une partie du système.

Gérer ses données

Bases de données Cloud Native

4. Multi-tenancy : Elles sont conçues pour supporter le multi-tenant, ce qui signifie que plusieurs clients ou utilisateurs peuvent partager la même infrastructure de base de données tout en conservant la séparation et l'isolation de leurs données.
5. Automatisation : Les bases de données cloud native sont conçues pour fonctionner avec un minimum de gestion manuelle. Elles peuvent souvent être gérées, mises à jour et monitorées par des API, ce qui permet une automatisation facile et une intégration avec d'autres outils cloud natifs.

- Des exemples de bases de données cloud native comprennent Google Cloud Spanner, Amazon DynamoDB, Microsoft Azure Cosmos DB, et CockroachDB. Ces bases de données offrent des modèles de données variés (relationnel, clé-valeur, document, colonnes larges, graphes) et prennent en charge divers types de transactions et de requêtes pour répondre aux besoins spécifiques des applications cloud native.

Gérer ses données

Stockage Cloud Native

- Le stockage cloud native est un concept qui désigne les méthodes de stockage de données conçues pour travailler de manière optimale avec les applications et les services basés sur le cloud.
- Les solutions de stockage cloud native sont conçues pour s'intégrer parfaitement aux environnements cloud, offrant une élasticité, une résilience et une facilité de gestion qui correspondent aux exigences des applications cloud native.

1. Élasticité : La capacité du stockage peut être augmentée ou diminuée à la demande pour répondre aux besoins changeants de l'application. Cela permet d'éviter le surdimensionnement ou le sous-dimensionnement, et assure que vous payez uniquement pour ce que vous utilisez.
2. Résilience : Les solutions de stockage cloud native sont généralement conçues pour être hautement disponibles et résistantes aux pannes. Elles peuvent utiliser la réplication des données sur plusieurs zones de disponibilité ou régions pour assurer la sécurité des données en cas de panne d'un composant ou d'une zone entière.
3. Interopérabilité : Le stockage cloud native est souvent conçu pour être compatible avec des API standard, comme l'interface S3 pour le stockage d'objets, ce qui facilite l'intégration avec diverses applications et services.

Gérer ses données

Stockage Cloud Native

4. Automatisation : Le stockage cloud native est conçu pour être facilement automatisé, ce qui permet une gestion efficace des ressources de stockage par le biais d'infrastructures codifiées et d'automatisation des opérations de routine.

- Il existe plusieurs types de stockage cloud native, chacun adapté à différents types de données et d'exigences :

1. Stockage d'objets : Il est idéal pour le stockage de grandes quantités de données non structurées, comme les images, les vidéos ou les sauvegardes. Les données sont stockées comme des objets dans un espace de noms plat et sont accessibles via des API HTTP. Exemples : Amazon S3, Google Cloud Storage.

Gérer ses données

Stockage Cloud Native

2. Stockage en bloc : Il fonctionne en allouant un certain nombre de blocs de stockage à une machine virtuelle ou à un conteneur, qui peut alors les utiliser comme un disque dur local. C'est un bon choix pour les bases de données et autres applications qui nécessitent un accès à faible latence et à haut débit. Exemples : Amazon EBS, Google Persistent Disk.
3. Stockage de fichiers : Il fournit un système de fichiers réseau qui peut être monté par plusieurs machines virtuelles ou conteneurs. C'est utile pour partager des fichiers entre plusieurs instances ou pour des applications qui nécessitent un système de fichiers POSIX. Exemples : Amazon EFS, Google Cloud Filestore.

Gérer ses données

La sécurité liée à la gestion des données Cloud Native

- La sécurité des données est un élément clé de toute stratégie de cloud natif. Avec les architectures cloud natives, les données sont souvent distribuées sur de nombreux services et emplacements, ce qui peut présenter des défis en termes de gestion et de sécurisation des données.

1. Chiffrement : Le chiffrement est essentiel pour protéger les données en transit et au repos. Les données sensibles doivent être chiffrées lorsqu'elles sont stockées et lorsqu'elles sont transmises entre les services. De nombreux fournisseurs de cloud offrent des services de gestion des clés de chiffrement pour faciliter le chiffrement et la rotation des clés.
2. Gestion des accès : L'accès aux données doit être soigneusement contrôlé. Cela comprend la gestion des permissions pour les utilisateurs et les services, ainsi que l'utilisation de politiques d'accès basées sur les rôles pour limiter l'accès aux données en fonction du besoin réel. Il est également important de suivre le principe du moindre privilège, en accordant aux utilisateurs et aux services uniquement les permissions nécessaires pour accomplir leurs tâches.

Gérer ses données

La sécurité liée à la gestion des données Cloud Native

3. Surveillance et audit : Les activités liées aux données doivent être surveillées et enregistrées pour détecter et réagir rapidement aux incidents de sécurité. Cela peut inclure le suivi des accès aux données, la détection des comportements anormaux et l'audit des actions effectuées sur les données.
4. Résilience des données : Les données doivent être protégées contre les pertes et les défaillances. Cela peut impliquer l'utilisation de la réplication pour stocker les données à plusieurs endroits, la sauvegarde régulière des données et la mise en place de plans de récupération après sinistre.

Observabilité et troubleshooting

Principes de l'observabilité Cloud Native

- L'observabilité est un aspect essentiel des applications Cloud Native. Elle désigne notre capacité à comprendre l'état interne d'un système en examinant ses sorties. Dans un contexte Cloud Native, cela implique généralement trois types de données : les métriques, les traces et les journaux.

1. Métriques : Les métriques sont des mesures quantitatives qui fournissent un aperçu de haut niveau du fonctionnement de l'application. Par exemple, la latence, le débit, l'erreur et l'utilisation des ressources sont des métriques couramment utilisées.
2. Traces : Les traces fournissent un enregistrement détaillé des interactions d'une transaction ou d'une demande avec les différents composants d'un système. Elles sont essentielles pour comprendre les performances et les problèmes d'un système distribué.
3. Journaux : Les journaux fournissent un enregistrement des événements qui se sont produits dans un système. Ils sont essentiels pour le débogage et l'audit.

Observabilité et troubleshooting

Principes de l'observabilité Cloud Native

- Quelques principes de base guident l'observabilité dans le cloud natif :

1. Automatisation : Dans le Cloud Native, l'observabilité doit être automatisée. Cela signifie que la collecte, l'agrégation et l'analyse des données d'observabilité doivent être intégrées dans le cycle de vie de l'application.
2. Contexte : Pour être utile, l'observabilité doit fournir un contexte. Cela signifie qu'il doit être possible de corréler les données d'observabilité avec les événements de l'application et les actions de l'utilisateur.
3. Evolutivité : L'observabilité doit être évolutive pour accompagner la croissance de l'application. Cela signifie que les outils et les processus d'observabilité doivent pouvoir gérer des volumes croissants de données et des systèmes de plus en plus complexes.
4. Accessibilité : Les données d'observabilité doivent être accessibles aux développeurs, aux opérateurs et aux autres parties prenantes. Cela signifie que les outils d'observabilité doivent être faciles à utiliser et fournir des informations claires et exploitables.

Observabilité et troubleshooting

Techniques de monitoring et d'analyse des logs

- Le monitoring et l'analyse des logs sont deux aspects clés de l'observabilité dans le contexte du Cloud Native.

1. Monitoring :

- Le monitoring est la pratique de collecter, analyser et utiliser les métriques pour comprendre les performances d'une application ou d'un système.

    1. Monitoring de l'infrastructure : Il s'agit de surveiller les performances et l'état de santé de l'infrastructure sur laquelle votre application s'exécute. Cela comprend le suivi des ressources comme le CPU, la mémoire, le réseau, le disque, etc. Des outils comme Prometheus, Grafana, ou les outils de surveillance intégrés des fournisseurs de cloud comme AWS CloudWatch, peuvent être utilisés pour cela.
    2. Monitoring de l'application : Il s'agit de surveiller les métriques spécifiques à l'application, comme le taux d'erreur, la latence, le débit, etc. Ces métriques sont souvent collectées à l'aide d'agents de surveillance intégrés à l'application ou à l'aide de bibliothèques de code.
    3. Monitoring distribué : Dans un système de microservices, il est important de pouvoir suivre une transaction à travers plusieurs services.

Observabilité et troubleshooting

Techniques de monitoring et d'analyse des logs

2. Analyse des logs :

- L'analyse des logs implique la collecte, le stockage, l'agrégation et l'analyse des journaux générés par les applications et l'infrastructure. Dans le contexte du Cloud Native, cela peut être réalisé de plusieurs façons :
    1. Agrégation des logs : Les environnements Cloud Native génèrent souvent des journaux à partir de nombreuses sources différentes. Des outils comme Fluentd ou Logstash peuvent être utilisés pour collecter ces journaux et les centraliser pour l'analyse.
    2. Stockage des logs : Une fois collectés, les journaux doivent être stockés de manière à faciliter l'analyse. Des outils comme Elasticsearch peuvent être utilisés pour stocker les journaux et permettre des recherches complexes.
    3. Analyse des logs : Une fois les journaux collectés et stockés, ils doivent être analysés pour en extraire des informations utiles. Des outils comme Kibana, Grafana, ou des services cloud comme AWS CloudWatch Logs Insights, peuvent être utilisés pour visualiser et analyser les journaux.

Observabilité et troubleshooting

Stratégies de résolution des problèmes

- Dans le contexte du Cloud Native, les stratégies de résolution de problèmes se concentrent sur l'identification rapide des problèmes, leur isolation, et la mise en œuvre de correctifs ou de solutions de contournement efficaces.

1. Observabilité : Comme mentionné précédemment, l'observabilité est cruciale pour la résolution des problèmes. Les métriques, les journaux et les traces fournissent des informations précieuses pour déterminer où se situe un problème, quels composants sont affectés, et comment les performances ou le comportement du système ont changé avec le temps.
2. Déploiement progressif et tests : La mise en œuvre de techniques de déploiement progressif, comme les déploiements en canary ou blue-green, permet de limiter l'impact des problèmes sur l'ensemble des utilisateurs. De plus, l'automatisation des tests à tous les niveaux (unitaires, d'intégration, de système, etc.) permet d'identifier rapidement les régressions et les autres problèmes.
3. Gestion des incidents : Les méthodes systématiques de gestion des incidents, qui peuvent inclure des éléments comme la désignation d'un responsable de la gestion des incidents, la communication régulière des mises à jour, et l'analyse post-incident, sont essentielles pour gérer efficacement les problèmes.

Observabilité et troubleshooting

Stratégies de résolution des problèmes

4. Conception pour la tolérance aux pannes : Dans le Cloud Native, l'objectif n'est pas d'éviter toutes les pannes (ce qui est pratiquement impossible à grande échelle), mais de minimiser l'impact des pannes lorsqu'elles se produisent. Cela peut être réalisé grâce à des techniques comme la réplication, le partitionnement, le redémarrage automatique des composants défaillants, et la mise en place de limites sur les ressources pour éviter les effets en cascade.
5. Post-mortem et apprentissage : Après la résolution d'un incident, il est important de mener une analyse post-mortem pour comprendre ce qui s'est mal passé, pourquoi, et comment éviter que de tels problèmes ne se reproduisent à l'avenir. Cela peut impliquer l'amélioration des tests, l'ajout de nouvelles métriques ou alertes, ou la modification de l'architecture ou des processus.

Élasticité et résilience

Introduction à l'élasticité et la résilience

L'élasticité et la résilience sont deux attributs clés des applications Cloud Native. Elles permettent à ces applications de s'adapter rapidement aux fluctuations de la demande et de récupérer efficacement des problèmes.

1. Élasticité :

L'élasticité est la capacité d'un système à s'adapter de manière flexible et efficace aux variations de charge. Dans le contexte du Cloud Native, l'élasticité implique généralement la possibilité d'ajouter ou de retirer rapidement des ressources (comme des conteneurs ou des nœuds de cluster) en fonction de la demande.
Pour atteindre l'élasticité, vous pouvez utiliser des services de mise à l'échelle automatique, qui surveillent les métriques de performance (comme l'utilisation du CPU ou de la mémoire) et ajoutent ou suppriment des ressources en fonction des seuils que vous définissez. Cela permet d'assurer que votre application dispose toujours des ressources nécessaires pour gérer la charge actuelle, tout en minimisant les coûts en évitant les ressources inutilisées.

Élasticité et résilience

Introduction à l'élasticité et la résilience

2. Résilience :

La résilience est la capacité d'un système à récupérer rapidement des erreurs et à continuer à fonctionner de manière fiable même en présence de problèmes. Dans le contexte du Cloud Native, la résilience implique souvent des techniques comme la redondance, la tolérance aux pannes, la dégradation gracieuse et le confinement des erreurs.
La redondance implique d'avoir des instances multiples d'un service pour pouvoir encaisser la défaillance d'une ou plusieurs d'entre elles. La tolérance aux pannes signifie concevoir votre système de manière à ce qu'il puisse continuer à fonctionner même si certains composants échouent. La dégradation gracieuse implique de permettre à votre application de continuer à fournir un service utile (même si c'est un service réduit) en cas de problème. Le confinement des erreurs (ou isolation des erreurs) implique d'empêcher les problèmes de se propager à d'autres parties du système.
En combinant l'élasticité et la résilience, vous pouvez créer des applications Cloud Native qui peuvent gérer efficacement les variations de la demande et minimiser l'impact des problèmes lorsqu'ils surviennent.

Élasticité et résilience

Gestion de la charge

- La gestion de la charge est un aspect essentiel du développement Cloud Native, car elle permet à vos applications de répondre efficacement à la demande variable des utilisateurs. Elle repose sur plusieurs techniques et principes, parmi lesquels :

1. Mise à l'échelle horizontale : La mise à l'échelle horizontale, ou scaling out, est une méthode qui consiste à ajouter plus de machines ou de nœuds à un système pour gérer une charge accrue. Par exemple, si une application est hébergée sur un cluster Kubernetes, vous pouvez augmenter le nombre de pods pour cette application en réponse à une augmentation de la demande.
2. Mise à l'échelle verticale : La mise à l'échelle verticale, ou scaling up, est une méthode qui consiste à ajouter plus de ressources (comme de la CPU ou de la mémoire) à une machine existante. Cette approche peut être utile pour les charges de travail qui ne peuvent pas être facilement réparties sur plusieurs nœuds, mais elle a des limites basées sur la capacité maximale du matériel.
3. Équilibrage de charge : L'équilibrage de charge répartit le trafic entrant entre plusieurs instances d'une application pour éviter de surcharger une seule instance. Cela peut être réalisé à l'aide de services d'équilibrage de charge, qui peuvent être fournis par votre plateforme Cloud ou être mis en œuvre à l'aide d'outils open source comme NGINX ou HAProxy.

Élasticité et résilience

Gestion de la charge

4. Mise en cache : La mise en cache est une technique qui consiste à stocker temporairement des données fréquemment demandées dans un emplacement à accès rapide pour réduire la charge sur votre système. Cela peut être particulièrement utile pour les données qui sont coûteuses à calculer ou à récupérer, et qui ne changent pas fréquemment.
5. Throttling : Le throttling, ou étranglement, est une technique qui consiste à limiter le taux auquel une application accepte les requêtes. Cela peut être utile pour empêcher une application d'être submergée par une charge trop importante.

Introduction à la conteneurisation Quarkus et GraalVM

GraalVM

- GraalVM est une machine virtuelle universelle de haute performance destinée à exécuter des applications écrites en JavaScript, Python, Ruby, R, JVM-based languages comme Java, Scala, Groovy, Kotlin, Clojure, et LLVM-based languages tels que C et C++. C'est un projet open source développé par Oracle Labs
- **Compilateur JIT Haute Performance** : GraalVM contient un compilateur "just-in-time" (JIT) de nouvelle génération appelé Graal, qui peut améliorer considérablement les performances des applications Java et JVM-based.
- **Compilateur AOT** : GraalVM est capable de compiler du code Java en code machine natif via "ahead-of-time" (AOT) compilation. Cela donne naissance à des programmes qui ne nécessitent pas de JVM pour s'exécuter, démarrant plus rapidement et consommant moins de mémoire. Cette fonctionnalité est souvent utilisée pour créer des microservices et des fonctions serverless.
- **Polyglotte** : GraalVM a la capacité d'exécuter des applications écrites dans plusieurs langages sur une même VM. Plus important encore, il permet à ces langages de s'interfacer les uns avec les autres de manière transparente. Par exemple, un code JavaScript peut appeler une fonction écrite en Python, et vice versa, sans frais d'interopérabilité importants.

Introduction à la conteneurisation Quarkus et GraalVM

GraalVM

- **Truffle Framework** : Truffle est un cadre pour écrire des interpréteurs pour n'importe quel langage de programmation. Ces interpréteurs, lorsqu'ils sont exécutés sur GraalVM, permettent à ce langage de bénéficier automatiquement des optimisations JIT de Graal. C'est grâce à Truffle que GraalVM supporte une multitude de langages.
- **Support des langages natifs** : Avec le support de LLVM, GraalVM peut exécuter du code écrit en langages qui compilent vers le bytecode LLVM, comme C et C++.
- **Intégration avec des outils et écosystèmes modernes** : GraalVM peut être intégré avec des outils tels que Docker et Kubernetes, et des frameworks tels que Micronaut, Quarkus, Spring, et Helidon pour créer des applications cloud-native.
- **GraalVM Native Image** : L'une des fonctionnalités les plus populaires de GraalVM est la capacité de produire une image native d'une application. Une "native image" est un exécutable autonome qui peut être exécuté sans JVM. Cela est particulièrement utile pour les applications qui nécessitent un démarrage rapide, comme les fonctions serverless.
- **Librairie de gestion de la mémoire** : GraalVM propose également une librairie de gestion de la mémoire, appelée SubstrateVM, qui remplace le garbage collector classique de la JVM pour les images natives.

Introduction à la conteneurisation Quarkus et GraalVM

Comprendre les différentes étapes du cycle de vie de l'application Quarkus

1. Développement:

La première étape du cycle de vie d'une application Quarkus est le développement. Cela inclut la création de l'application, l'écriture du code, la configuration de l'application et le test de son comportement. Un des principaux avantages de Quarkus est le mode développement qui permet un "live coding" - c'est-à-dire, les modifications du code sont rechargées à la volée, ce qui accélère grandement le cycle de développement.

2. Construction:

La phase de construction est là où votre application est préparée pour être déployée. En utilisant Maven ou Gradle, l'application est compilée en un artefact que vous pouvez déployer. Dans le cas de Quarkus, cet artefact peut être soit une application JVM classique, soit un exécutable natif grâce à GraalVM.

Introduction à la conteneurisation Quarkus et GraalVM

Comprendre les différentes étapes du cycle de vie de l'application Quarkus

3. Test:

Les tests sont un élément crucial du cycle de vie de l'application. Quarkus prend en charge les tests unitaires avec JUnit et a également une intégration avec Testcontainers pour les tests d'intégration. De plus, grâce à son démarrage rapide, Quarkus est particulièrement adapté aux tests, car il permet d'exécuter les tests beaucoup plus rapidement que les applications Java classiques.

4. Déploiement:

Une fois votre application testée, elle est prête à être déployée. Vous pouvez déployer une application Quarkus comme n'importe quelle autre application Java. Vous pouvez la déployer sur un serveur d'applications, dans un conteneur Docker, sur une plateforme Kubernetes, ou dans un environnement serverless. Si vous avez compilé votre application en un exécutable natif, elle sera encore plus efficace en termes d'utilisation des ressources et de temps de démarrage.

Introduction à la conteneurisation Quarkus et GraalVM

Comprendre les différentes étapes du cycle de vie de l'application Quarkus

5. Monitoring:

Une fois que votre application est déployée, vous voulez la surveiller pour vous assurer qu'elle fonctionne correctement. Quarkus a une intégration avec MicroProfile Metrics, ce qui vous permet de collecter et d'exposer des métriques d'application qui peuvent être récupérées et analysées par des outils de surveillance.

6. Maintenance:

La dernière étape du cycle de vie de l'application est la maintenance. Cela peut inclure la mise à jour des dépendances, la résolution des bugs, l'ajout de nouvelles fonctionnalités, ou même le scale up ou down en fonction de la charge de travail. Quarkus facilite la maintenance grâce à son modèle de programmation familier basé sur des standards populaires tels que CDI, JAX-RS, etc.

Introduction à la conteneurisation Quarkus et GraalVM

Démo Développement

- Utilisation de la configuration
- Construction des ressources avec JAX-RS
- Utilistion de CDI

Introduction à la conteneurisation Quarkus et GraalVM

Développement Exercice

- Création d'une API REST pour gérer un carnet d'adresses
- Contexte : Vous travaillez pour une entreprise qui a besoin d'une API pour gérer un carnet d'adresses en ligne. Chaque entrée du carnet d'adresses doit avoir les propriétés suivantes:
  - Nom
  - Adresse email
  - Numéro de téléphone
- Votre tâche est de créer une API REST avec les fonctionnalités suivantes:

1. Créer une nouvelle entrée du carnet d'adresses : Les clients doivent être en mesure de créer une nouvelle entrée en envoyant une requête POST à /contacts. Le corps de la requête doit être un objet JSON contenant le nom, l'adresse email et le numéro de téléphone.
2. Récupérer toutes les entrées du carnet d'adresses : Les clients doivent être en mesure de récupérer toutes les entrées en envoyant une requête GET à /contacts.
3. Mettre à jour une entrée du carnet d'adresses : Les clients doivent être en mesure de mettre à jour une entrée en envoyant une requête PUT à /contacts/{id}. Le corps de la requête doit être un objet JSON contenant le nom, l'adresse email et le numéro de téléphone.
4. Supprimer une entrée du carnet d'adresses : Les clients doivent être en mesure de supprimer une entrée en envoyant une requête DELETE à /contacts/{id}.

Pour réaliser cet exercice, vous devrez créer une nouvelle application Quarkus, définir une classe Contact avec les propriétés requises, et créer une classe ContactResource qui expose les méthodes requises via des endpoints REST.

Introduction à la conteneurisation Quarkus et GraalVM

Démo Construction

- Construction pour la JVM
- Construction native pour GraalVM.
- Construction des images de conteneurs

Introduction à la conteneurisation Quarkus et GraalVM

Construction Exercice

On souhaite créer une image de conteneur pour notre application créée par l'exercice développement:

- Ajouter un dockerfile pour créer l'image.

Introduction à la conteneurisation Quarkus et GraalVM

Déploiement Exercice

On souhaite créer déployer notre exercice annuaire dans un cluster kubernetes

1. Health Checks: Implémentez des contrôleurs de santé pour l'API:

- Readiness Check: Indiquez si l'API est prête à répondre aux requêtes. Par exemple, vérifiez si toutes les initialisations nécessaires ont été complétées et si la base de données est accessible.
- Liveness Check: Indiquez si l'API est en cours d'exécution et en bonne santé.

2. Déploiement Kubernetes: Préparez votre API pour le déploiement dans un cluster Kubernetes :

- Rédigez un manifeste Kubernetes pour déployer l'API.
- Utilisez le health check précédemment créé pour configurer les sondes de liveness et readiness de Kubernetes.

Quarkus

Fault Tolerance

- Quarkus s'intègre à l'extension SmallRye Fault Tolerance pour offrir des fonctionnalités de tolérance aux pannes. Voici quelques-unes des principales fonctionnalités fournies par cette extension:

1. **Retry**

- Permet de retenter automatiquement une méthode en cas d'échec. Vous pouvez configurer le nombre maximum de tentatives, le délai entre elles, etc.

2. **Timeout**

- Définit une durée maximale pour l'exécution d'une méthode. Si la méthode ne se termine pas dans le délai imparti, une exception est levée.

Quarkus

Fault Tolerance

3. **Circuit Breaker**

- Agit comme un disjoncteur électrique. Si un nombre spécifié d'échecs est atteint, le circuit s'ouvre et empêche l'exécution de la méthode pendant un certain temps. Cela permet d'éviter de surcharger un service déjà en difficulté.

4. **Fallback**

- Fournit une méthode alternative à exécuter en cas d'échec de la méthode principale. Cela garantit que votre application peut toujours répondre d'une manière ou d'une autre, même si ce n'est pas l'option idéale.

Quarkus

Fault Tolerance Exercice

Contexte :

- Vous travaillez pour une entreprise qui souhaite développer une application permettant de récupérer la météo d'une ville donnée. Pour cela, elle utilise un service externe de météo. Cependant, ce service est connu pour être parfois instable. Vous devez donc construire une application qui peut gérer ces instabilités.

Service de météo:

1. Définissez un client REST pour le service de météo externe.
   Supposons que le service externe ait une API accessible via GET /weather?city={cityName}.

Circuit Breaker:

2. Si le service de météo échoue plus de 3 fois consécutivement, ouvrez un circuit breaker pendant 2 minutes. Pendant cette période, toutes les demandes doivent être automatiquement redirigées vers une méthode de repli.

Quarkus

Fault Tolerance Exercice

Méthode de repli (Fallback):

3. Si le service de météo n'est pas accessible ou si le circuit breaker est ouvert, retournez une réponse générique, par exemple : "Les informations météorologiques ne sont pas disponibles pour le moment. Veuillez réessayer plus tard."

Endpoint:

4. Exposez un endpoint GET /api/weather/{cityName}.

Lorsqu'un utilisateur accède à cet endpoint, il doit obtenir les informations météorologiques pour la ville donnée, soit depuis le service externe, soit depuis la méthode de repli si le circuit breaker est ouvert.

Observation d'une application Java conteneurisée

- Observer une application Quarkus native conteneurisée présente des défis spécifiques, car Quarkus optimise les applications Java pour la compilation native via GraalVM. Lorsqu'une application Java est compilée en exécutable natif, plusieurs aspects de son comportement changent par rapport à l'exécution sur une JVM traditionnelle.
- **Réduction des indicateurs de la JVM** :
- Les applications natives ne fonctionnent pas sur une JVM, donc beaucoup d'indicateurs JVM couramment surveillés (comme l'utilisation de la mémoire heap, le comptage des threads, les statistiques GC, etc.) ne sont pas disponibles ou ne sont pas pertinents.
- **Utilisation de la mémoire :**
- Les applications natives ont tendance à avoir une utilisation de la mémoire au démarrage beaucoup plus faible que les applications JVM traditionnelles. Cependant, étant donné que la gestion de la mémoire est différente, il peut être nécessaire de surveiller de manière différente la croissance potentielle de l'utilisation de la mémoire.
- **Démarrage rapide :**
  Les applications Quarkus natives démarrent très rapidement, ce qui est idéal pour les environnements éphémères comme Kubernetes. Cependant, cela signifie également que les problèmes peuvent survenir et se résoudre très rapidement, rendant les défaillances temporaires plus difficiles à détecter.

Observation d'une application Java conteneurisée

- **Dépendance des bibliothèques natives :**
- Les applications natives dépendent des bibliothèques du système sous-jacent. Toute incompatibilité ou problème avec ces bibliothèques peut affecter l'application, il est donc essentiel de surveiller également le conteneur et l'hôte sous-jacent.
- **Introspection limitée :**
- Les outils qui dépendent de la JVM pour l'introspection (par exemple, certains profilers ou débogueurs) ne fonctionneront pas avec les applications natives. Les équipes peuvent avoir besoin de nouveaux outils ou méthodes pour observer et déboguer des problèmes.
- **Gestion des logs et des traces :**
- Bien que ce ne soit pas spécifique à Quarkus ou aux applications natives, dans un environnement conteneurisé, les logs doivent être traités différemment. Plutôt que d'écrire localement, ils doivent être envoyés à un collecteur ou à un service centralisé.
- **Métriques et surveillance :**
- Les mécanismes traditionnels de collecte de métriques peuvent ne pas fonctionner ou nécessiter une adaptation. Heureusement, Quarkus offre des extensions pour faciliter la collecte de métriques, mais leur mise en place et leur compréhension nécessitent un effort.

Observation d'une application Java conteneurisée

Interactions avec d'autres services :

- Dans un environnement microservices, une application conteneurisée peut interagir avec de nombreux autres services. Suivre les interactions, notamment les appels réseau, est essentiel pour comprendre les performances et les erreurs.
- **Changements environnementaux :**
  Les conteneurs peuvent être déplacés, redémarrés ou supprimés en fonction des besoins de l'orchestrateur. Il est donc crucial de surveiller non seulement l'application elle-même, mais aussi l'environnement dans lequel elle s'exécute.

Monitoring d'une application Quarkus

- Le monitoring d'une application Quarkus peut se faire à l'aide de plusieurs outils et techniques
- **Extensions Quarkus pour le Monitoring:**
- Quarkus fournit plusieurs extensions qui facilitent l'intégration avec des outils de monitoring populaires.
  - Micrometer : C'est une bibliothèque de métriques pour JVM qui peut intégrer avec plusieurs systèmes de monitoring comme Prometheus, Grafana, etc.
  - SmallRye Health : Fournit une API pour vérifier la santé de votre application.
  - OpenTracing : Pour la traçabilité distribuée.

Feature Flipping

- **Déploiement Progressif**: Le feature flipping permet de déployer une fonctionnalité sans l'activer immédiatement pour tous les utilisateurs. Cela offre la possibilité de tester la fonctionnalité en production mais dans des conditions contrôlées.
- **Réduction des Risques:** Si une nouvelle fonctionnalité pose problème, elle peut être désactivée instantanément, minimisant ainsi les impacts négatifs.
- **Flexibilité de Déploiement:** Dans les environnements conteneurisés comme Kubernetes, il est possible de déployer une nouvelle image contenant plusieurs nouvelles fonctionnalités et de les activer/désactiver selon les besoins.
- Quarkus ne fournit pas directement une extension pour le feature flipping, mais vous pouvez utiliser des bibliothèques Java comme **Togglz** ou **Unleash**. Ces bibliothèques peuvent être intégrées dans une application Quarkus comme toute autre dépendance Maven.

Feature Flipping

- **Déploiement à Basculer:** Activez une fonctionnalité pour un pourcentage spécifique d'utilisateurs ou sur des critères spécifiques (par exemple, géographie, type d'utilisateur).
- **Tests A/B:** Utilisez le feature flipping pour implémenter des tests A/B, où différentes variations d'une fonctionnalité sont présentées à différents groupes d'utilisateurs.
- **Configurer les fonctionnalités pour le feature flipping:** À l'aide de la bibliothèque choisie, définissez des "toggle points" dans le code qui peuvent être activés ou désactivés à l'aide de clés. Par exemple, avec Togglz, vous pouvez définir des fonctionnalités et les contrôler via un tableau de bord ou une configuration.
- **Déployer la nouvelle fonctionnalité graduellement:** Après avoir déployé la nouvelle version de votre application conteneurisée, activez la nouvelle fonctionnalité pour un petit groupe d'utilisateurs. Surveillez les performances, les erreurs et recueillez les retours d'expérience.
- **Gérer les erreurs:** Si des problèmes surviennent après avoir activé une nouvelle fonctionnalité, utilisez le système de feature flipping pour désactiver rapidement cette fonctionnalité. Cela évite d'avoir à effectuer un déploiement en urgence pour corriger le problème.

Etude de cas

Service de Citations et d'Auteurs avec Quarkus

Dans cette étude de cas, vous allez développer deux microservices: l'un pour les citations (QuoteService) et l'autre pour les auteurs (AuthorService). QuoteService consommera AuthorService pour associer une citation à un auteur.

1. **Développement de QuoteService**

- Initialisez un projet Quarkus.
- Mettez en place un service REST.
- Créez un endpoint qui renvoie une citation aléatoire.
- Assurez-vous que chaque citation est associée à un ID d'auteur.

2. **Développement de AuthorService**

- Initialisez un autre projet Quarkus.
- Mettez en place un service REST pour ce projet.
- Créez un endpoint qui renvoie les détails d'un auteur en fonction de l'ID fourni.
- Préparez quelques entrées d'auteurs pour être interrogées.

Etude de cas

Service de Citations et d'Auteurs avec Quarkus

3. **Intégration et Communication**

- Intégrez les outils nécessaires à QuoteService pour lui permettre de consommer AuthorService.
- Faites en sorte que l'endpoint des citations interroge également le service d'auteurs pour fournir les détails de l'auteur associé à la citation.

4. **Ajout de Résilience**

- Intégrez une fonctionnalité qui simule des échecs lors de l'appel d'un service à l'autre.
- Implémentez les mécanismes de résilience pour gérer ces échecs.

5. **Déploiement Natif avec Kubernetes**

- Compilez votre application pour un déploiement natif.
- Préparez les conteneurs pour les deux services.
- Déployez ces services sur un cluster Kubernetes.

Etude de cas

Service de Citations et d'Auteurs avec Quarkus

6. **Monitoring**

- Intégrez des métriques pour les deux services.
- Préparez l'infrastructure pour surveiller ces métriques.
- Visualisez ces métriques en temps réel.