Microsoft

Azure

Guide d'architecture d'applications cloud

PUBLIÉ PAR
Microsoft Press
Une division de Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Les livres Microsoft Press sont disponibles auprès des libraires et distributeurs du monde entier. Si vous avez besoin d'aide au sujet de ce livre, contactez le support Microsoft Press à l'adresse mspinput@microsoft.com. Veuillez nous faire part de votre avis sur ce livre à l'adresse http://aka.ms/tellpress.

Rédacteur responsable des commandes :
Christopher Bennage

Rédacteurs responsables du développement :
Mike Wasson, Masashi Narumoto et l'équipe Microsoft Patterns and Practices

Production éditoriale :
Phil Evans

Responsable révision :
Jamie Letain

Table des matières

Guide d'architecture d'applications cloud

Ce guide offre une approche structurée de la conception d'applications cloud évolutives, résilientes et hautement disponibles. Les conseils de ce livre blanc sont destinés à aider vos décisions architecturales, quelle que soit la plateforme cloud que vous utilisez. Nous allons cependant utiliser Azure pour partager les meilleures pratiques que nous avons apprises au cours de nombreuses années d'engagement des clients.

Dans les chapitres suivants, nous vous guiderons à travers une sélection de ressources et de considérations importantes pour contribuer à déterminer la meilleure approche pour votre application cloud :

1. Sélection de l'architecture correspondant à votre application en fonction du type de solution que vous concevez.
2. Choix des technologies de banque de données et de calcul les plus appropriées.
3. Intégration des dix principes de conception essentiels afin d'assurer l'évolutivité, la résilience et la facilité de gestion de votre application.
4. Utilisation des cinq piliers de la qualité logicielle pour bâtir un succès.
5. Application de patrons de conception spécifiques au problème que vous essayez de résoudre.

Introduction

Le cloud révolutionne la conception des applications. Plutôt que d'être constituées d'un seul bloc, les applications se décomposent en petits services décentralisés. Ces services communiquent via des API ou des événements/ une messagerie asynchrones. Les applications évoluent de façon horizontale par l'ajout de nouvelles instances en fonction de la demande.

Ces tendances s'accompagnent de nouveaux défis. Les applications sont distribuées. Les opérations s'effectuent en parallèle et de manière asynchrone. Le système dans son ensemble doit être résilient en cas de défaillance. Les déploiements doivent être automatisés et prévisibles. La surveillance et la télémétrie sont des éléments essentiels pour disposer d'informations sur le système. Le guide d'architecture des applications Azure est conçu pour vous aider à gérer ces changements.

Gestion classique sur site

- Monolithique, centralisée
- Conception pour une évolutivité prévisible
- Base de données relationnelle
- Forte cohérence
- Traitement en série et synchronisé
- Conception pour éviter les défaillances (MTBF)
- Mises à jour majeures occasionnelles
- Gestion manuelle
- Serveurs Snowflake

Cloud moderne

- Décomposé, décentralisé
- Conception pour une montée en charge élastique
- Persistance polyglotte (mélange de technologies de stockage)
- Cohérence à terme
- Design for failure, « Conçu pour supporter la défaillance » (MTTR)
- Mises à jour mineures fréquentes
- Autogestion automatisée
- Infrastructure immuable

Le cloud révolutionne la conception des applications. Plutôt que d'être constituées d'un seul bloc, les applications se décomposent en petits services décentralisés. Ces services communiquent via des API ou des événements/une messagerie asynchrones. Les applications évoluent de façon horizontale par l'ajout de nouvelles instances en fonction de la demande.

Ces tendances s'accompagnent de nouveaux défis. Les applications sont distribuées. Les opérations s'effectuent en parallèle et de manière asynchrone. Le système dans son ensemble doit être résilient en cas de défaillance. Les déploiements doivent être automatisés et prévisibles. La surveillance et la télémétrie sont des éléments essentiels pour disposer d'informations sur le système. Le guide d'architecture des applications cloud est conçu pour vous aider à gérer ces changements.

Structure du guide

Le guide d'architecture des applications cloud se présente comme une série d'étapes allant de l'architecture et la conception à la mise en œuvre. Pour chaque étape, des conseils vous aideront à concevoir l'architecture de votre application.

Styles d'architecture. Le premier élément de décision est l'élément le plus fondamental. Quel type d'architecture allez-vous concevoir ? Il peut s'agir d'une architecture de microservices, d'une application n-tier plus classique ou d'une solution de Big Data. Nous avons identifié sept styles d'architecture distincts. Chacun de ces styles présente des avantages et des défis propres.

- Les architectures de référence Azure présentent des déploiements recommandés dans Azure, ainsi que des éléments de réflexion sur l'évolutivité, la disponibilité, la facilité de gestion et la sécurité. La plupart de ces déploiements incluent également des modèles Resource Manager pouvant être déployés.

Large choix de technologies. Deux décisions en matière de technologie doivent être prises dès le départ, parce qu'elles auront un impact sur l'intégralité de l'architecture. Il s'agit du choix des technologies de calcul et de stockage. Le terme calcul désigne le modèle d'hébergement des ressources informatiques sur lesquelles s'exécutent vos applications. Le stockage se rapporte aux base de données, mais aussi au stockage des files d'attente de messages, des caches, des données IoT, des données de journal non structurées, ainsi qu'à tout ce qu'une application est susceptible de stocker en permanence.

- Les options de calcul et les options de stockage offrent des critères de comparaison détaillés pour sélectionner les services de calcul et de stockage.

Principes de conception. Tout au long du processus de conception, gardez à l'esprit ces dix principes de conception essentiels.

- Pour découvrir les articles sur les meilleures pratiques qui fournissent des indications précises sur la mise à l'échelle automatique, la mise en cache, le partitionnement de données, la conception d'API et autres, rendez-vous sur la page https://docs.microsoft.com/en-us/azure/architecturebest-practices/index.

Piliers. Une application cloud réussie mettra l'accent sur ces cinq piliers de la qualité logicielle : évolutivité, disponibilité, résilience, gestion et sécurité.

- Utilisez nos listes de revue de conception pour évaluer votre conception à l'aune de ces piliers de la qualité.

Patrons de conception cloud. Ces patrons de conception sont utiles pour concevoir des applications fiables, évolutives et sécurisées sur Azure. Chaque patron décrit un problème, un patron qui résout le problème et un exemple basé sur Azure.

- Découvrez l'intégralité du catalogue des patrons de conception cloud.

Choisir un style d'architecture

Lorsque vous concevez une application cloud, la première décision que vous devez prendre concerne l'architecture. Il s'agit de choisir la meilleure architecture pour l'application que vous concevez, en fonction de sa complexité, du type de domaine, de son modèle (application IaaS ou PaaS) et des fonctionnalités de l'application. Réfléchissez également aux compétences des équipes DevOps et de développement, ainsi qu'à l'existence préalable ou non d'une architecture existante pour cette application.

Un style d'architecture place des contraintes sur la conception, qui vont guider la « forme » d'un style d'architecture en limitant les choix possibles. Ces contraintes offrent à la fois des avantages et des défis en matière de conception. Exploitez les renseignements présents dans cette section pour comprendre les concessions qui vont accompagner l'adoption de chacun de ces styles.

Cette section décrit dix principes de conception à garder à l'esprit lors de la génération d'application. Ils vous aideront à générer une application plus évolutive, plus résiliente et plus facile à gérer.

Nous avons identifié un ensemble de styles d'architecture que l'on retrouve généralement dans des applications cloud. Pour chaque style, l'article comprend les éléments suivants :

- Une description et un diagramme logique du style.
- Des recommandations pour déterminer les situations dans lesquelles choisir ce style.
- Des avantages, des problématiques et des meilleures pratiques.
- Un déploiement recommandé en utilisant les services Azure pertinents.

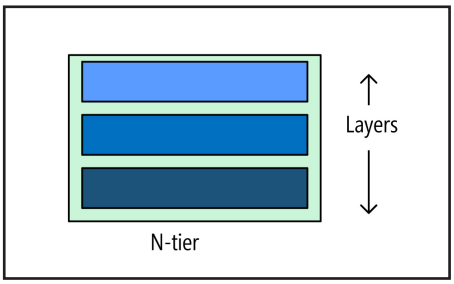
Présentation rapide des différents styles

Cette section donne un aperçu des styles d'architecture que nous avons identifiés, ainsi que quelques considérations importantes en vue de leur utilisation. Parcourez les rubriques liées pour obtenir des informations plus détaillées.

N-tier

N-tier est une architecture classique pour les applications d'entreprise. La gestion des dépendances se fait en divisant l'application en plusieurs couches qui remplissent des fonctions logiques telles que la présentation, la logique métier et l'accès aux données. Une couche peut uniquement appeler les couches situées en dessous. Toutefois, cette stratification horizontale peut s'avérer être un handicap. En effet, il peut être difficile d'introduire des changements dans une partie de l'application sans toucher au reste de l'application. La réalisation de mises à jour fréquentes devient ainsi un véritable défi, limitant le rythme auquel il est possible d'ajouter de nouvelles fonctionnalités.

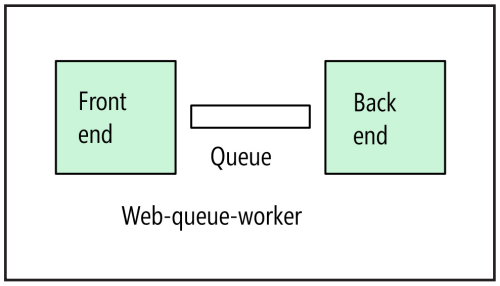
N-tier est un choix logique pour la migration d'applications existantes qui utilisent déjà une architecture en couches. C'est la raison pour laquelle le n-tier est plus fréquemment observé dans les solutions IaaS (Infrastructure as a Service) ou dans les applications utilisant à la fois des services Iaas et des services gérés.



Web-File d'attente-Agent de travail

Pour une solution strictement PaaS, pensez à une architecture Web-File d'attente-Agent de travail. Dans ce style, l'application possède un serveur frontal Web qui traite les requêtes HTTP et un agent de travail back-end qui effectue les tâches très consommatrices de ressources du processeur ou les opérations nécessitant un délai d'exécution long. Le serveur frontal communique avec l'agent de travail via une file d'attente de messages asynchrones.

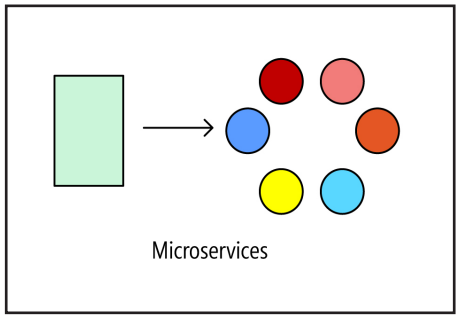
Web-File d'attente-Agent de travail convient à des domaines relativement simples avec quelques tâches gourmandes en ressources. Comme N-tier, l'architecture est facile à comprendre. L'utilisation de services gérés simplifie le déploiement et l'exploitation. Mais avec des domaines complexes, il peut être difficile de gérer les dépendances. Le serveur frontal et l'agent de travail peuvent facilement devenir des composants monolithiques volumineux, difficiles à gérer et à mettre à jour. Comme avec N-tier, cela peut réduire la fréquence des mises à jour et limiter l'innovation.



Microservices

Si votre application possède un domaine plus complexe, vous pouvez envisager de migrer vers une architecture de microservices. Une application de microservices est composée de nombreux petits services indépendants. Chaque service met en œuvre une fonctionnalité d'entreprise unique. Les services sont associés librement et communiquent via des contrats API.

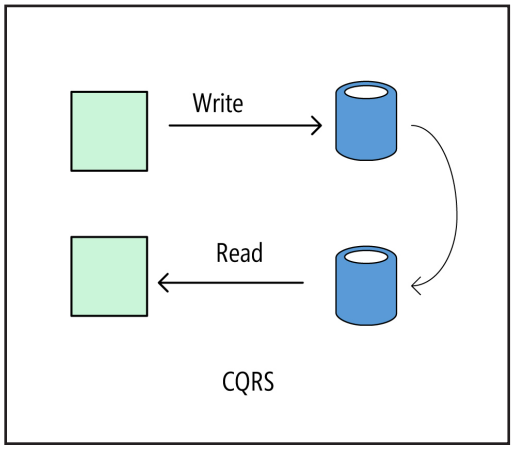
Chaque service peut être construit par une petite équipe de développement dédiée. Les services individuels peuvent être déployés sans nécessiter de coordination importante entre les équipes, ce qui encourage la réalisation de mises à jour fréquentes. Une architecture de microservices est plus complexe à construire et à gérer qu'une architecture N-tier ou Web-File d'attente-Agent de travail. Elle nécessite un développement mature et une culture DevOps. Lorsque la conception est réussie, ce style peut donner les résultats suivants : vitesse de publication élevée, innovation accélérée et architecture plus résiliente.



CQRS

Le style CQRS (Command and Query Responsibility Segregation) sépare les opérations de lecture des opérations d'écriture en modèles distincts. Cela isole les parties du système chargées de la mise à jour des données des parties chargées de la lecture des données. En outre, les lectures peuvent être exécutées sur une vue matérialisée qui est physiquement séparée de la base de données d'écriture. Cela permet une mise à l'échelle indépendante des scénarios d'usage en lecture et en écriture, ainsi qu'une optimisation de la vue matérialisée pour les requêtes.

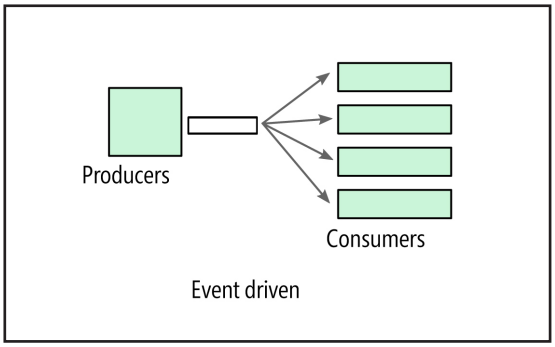
Le style CQRS est parfaitement justifié lorsqu'il est appliqué à un sous-système d'une architecture plus grande. En règle générale, il ne faut pas l'imposer sur l'ensemble de l'application, car cela engendrera une complexité inutile. Envisagez-le pour les domaines de collaboration où de nombreux utilisateurs accèdent aux mêmes données.



Architecture orientée événements

Les architectures orientées événements utilisent un modèle de publication/abonnement (pub/sub) dans lequel des producteurs publient des événements auxquels les consommateurs s'abonnent. Les producteurs sont indépendants des consommateurs et les consommateurs sont indépendants les uns des autres.

Vous pouvez envisager une architecture orientée événements pour les applications qui ingèrent et traitent un volume important de données avec une latence très faible, telles que les solutions IoT. Ce style est également utile lorsque différents sous-systèmes doivent effectuer différents types de traitement sur les mêmes données d'événement.



Big Data, Big Compute

Big Data et Big Compute sont les styles d'architectures spécialisées pour les scénarios d'usage correspondant à certains profils spécifiques. Big Data divise un très grand ensemble de données en blocs et effectue en parallèle un traitement sur l'ensemble à des fins d'analyse et de génération de rapports. Big Compute, également appelé calcul hautes performances (HPC – High Performance Computing), effectue des calculs parallèles sur un grand nombre (plusieurs milliers) de cœurs. Domaines concernés : les simulations, la modélisation et le rendu 3D.

Styles d'architecture sous forme de contraintes

Un style d'architecture place des contraintes sur la conception, y compris l'ensemble d'éléments qui peuvent apparaître et les relations autorisées entre ces éléments. Ces contraintes vont guider la « forme » d'une architecture en limitant le panorama de choix possibles. Lorsqu'une architecture est conforme aux contraintes d'un style particulier, certaines propriétés souhaitables émergent.

Par exemple, les contraintes d'une architecture de microservices comprennent les points suivants :

- Un service représente une seule responsabilité.
- Chaque service est indépendant des autres.
- Les données sont accessibles uniquement au service auquel elles appartiennent. Les services ne partagent pas les données.

En adhérant à ces contraintes, le système qui émerge est un système dans lequel il est possible de déployer des services de façon indépendante, d'isoler des défaillances, de réaliser des mises à jour fréquentes, et dans lequel il est facile d'intégrer de nouvelles technologies dans l'application.

Avant de choisir un style d'architecture, assurez-vous que vous comprenez les principes sous-jacents et les contraintes de ce style. Sinon, vous risquez de vous retrouver avec une conception conforme au style à un niveau superficiel, mais qui n'atteint pas le plein potentiel de ce style. Il est également important de faire preuve de pragmatisme. Il est parfois préférable d'assouplir une contrainte plutôt que d'insister sur la pureté architecturale.

Le tableau qui suit résume la façon dont chaque style gère les dépendances et les types de domaine les mieux adaptés à chaque style.

Style d'architecture	Gestion des dépendances	Type de domaine
N-tier	Niveaux horizontaux divisés par sous-réseau.	Secteur d'activité classique. La fréquence des mises à jour est faible.
Web-File d'attente-Agent de travail	Tâches en front et back-end découplées par messagerie asynchrone.	Domaine relativement simple avec quelques tâches intensives en ressources.
Microservices	Services décomposés verticalement (fonctionnellement) qui s'appellent mutuellement via des API.	Domaine complexe. Mises à jour fréquentes.
CQRS	Ségrégation lecture/écriture. Le schéma et l'échelle sont optimisés séparément.	Domaine de collaboration où beaucoup d'utilisateurs accèdent aux mêmes données.
Architecture orientée événements	Producteur/consommateur. Vue indépendant par sous-système.	IoT et systèmes en temps réel.
Le Big Data	Division d'un très grand ensemble de données en petits blocs. Traitement parallèle sur des ensembles de données locaux.	Analyse des données en temps réel et en lots. Analyse prédictive à l'aide de ML.
Le Big Compute	Ventilation des données sur des milliers de cœurs.	Domaines de calculs intensifs tels que de la simulation.

Envisager les défis et les avantages

Les contraintes génèrent également des défis, c'est pourquoi il est important de comprendre les concessions qui accompagnent l'adoption de chacun de ces styles. Les avantages de l'architecture l'emportent-ils sur les défis pour ce sous-domaine et dans ce contexte délimité ?

Voici quelques-unes des problématiques à prendre en compte lors du choix d'un style d'architecture :

- **Complexité.** La complexité de l'architecture est-elle justifiée pour votre domaine ? À l'inverse, le style est-il trop simpliste pour votre domaine ? Dans ce cas, vous risquez de vous retrouver avec une « boule de boue », parce que l'architecture ne vous aide pas à gérer correctement les dépendances.

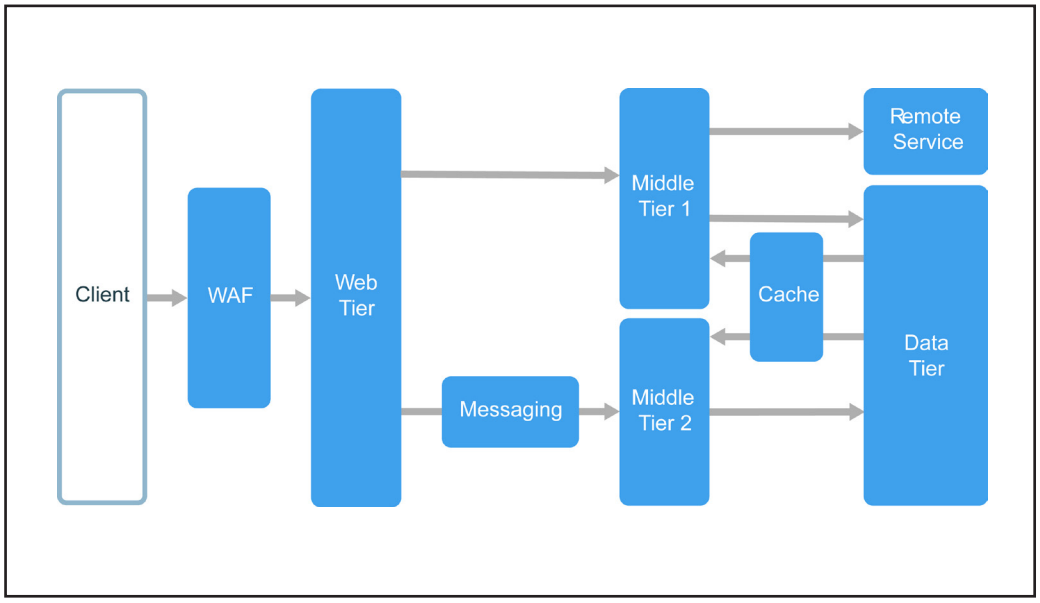
- **Messagerie asynchrone et cohérence à terme.** La messagerie asynchrone peut être utilisée pour dissocier les services et accroître la fiabilité (parce que les messages peuvent être retentés) et l'évolutivité. Toutefois, cela crée également des défis tels que la sémantique et la cohérence à terme.

- **Communication entre services.** Lorsque vous décomposez une application en services distincts, il existe un risque que la communication entre services provoque une latence inacceptable ou génère une congestion du réseau (par exemple, dans une architecture de microservices).

- **Simplicité de gestion.** Est-ce qu'il est difficile de gérer l'application, de surveiller, de déployer des mises à jour et ainsi de suite ?

Style d'architecture N-tier

Une architecture N-tier divise une application en couches logiques et niveaux physiques.



Les couches sont un moyen de séparer les responsabilités et de gérer les dépendances. Chaque couche a une responsabilité spécifique. Une couche supérieure peut utiliser des services dans une couche inférieure, mais pas l'inverse.

Les niveaux sont physiquement séparés et s'exécutent sur des machines distinctes. Un niveau peut appeler directement un autre niveau ou utiliser la messagerie asynchrone (file d'attente de messages). Bien que chaque couche puisse être hébergée dans son propre niveau, cela n'est pas obligatoire. Plusieurs couches peuvent être hébergées sur le même niveau. La séparation physique des couches améliore l'évolutivité et la résilience, mais ajoute également de la latence issue de la communication du réseau supplémentaire.
Une application classique à trois niveaux possède une couche de présentation, une couche intermédiaire et une couche de base de données. La couche intermédiaire est facultative. Des applications plus complexes peuvent avoir plus de trois niveaux. Le diagramme ci-dessus montre une application avec deux niveaux intermédiaires encapsulant différents domaines de fonctionnalités.

Une application N-tier peut avoir une ***architecture de couches fermées*** ou une ***architecture de couches ouvertes*:**

- Dans une architecture de couches fermées, une couche peut uniquement appeler la couche située immédiatement en aval
- Dans une architecture en couches ouvertes, une couche peut appeler n'importe quelle couche en aval.

Une architecture en couches fermées limite les dépendances entre les couches. Toutefois, elle risque de créer un trafic réseau inutile si une couche ne fait que transmettre les requêtes à la couche suivante.

Quand utiliser cette architecture

Les architectures N-tier sont généralement implémentées comme des applications Infrastructure-as-a-Service (IaaS), chaque niveau s'exécutant sur un ensemble distinct de machines virtuelles. Cependant, il n'est pas obligatoire qu'une application N-tier soit intégralement IaaS. Souvent, il est avantageux d'utiliser des services gérés pour certaines parties de l'architecture, en particulier la mise en cache, la messagerie et le stockage de données.

Envisagez une architecture N-tier dans les cas suivants :

- Applications web simples.
- Migration d'une application sur site vers Azure avec un minimum de refactorisation.
- Développement unifié des applications cloud et sur site.

Les architectures N-tier sont très fréquentes dans les applications classiques sur site, c'est donc un choix qui prend tout son sens pour la migration des scénarios d'usage existants sur Azure.

Avantages

- Portabilité entre le cloud et le site, ainsi qu'entre les plateformes cloud.
- Courbe d'apprentissage moindre pour la plupart des développeurs.
- Évolution naturelle du modèle traditionnel d'application.
- Environnement ouvert à hétérogène (Windows/Linux)

Problématique

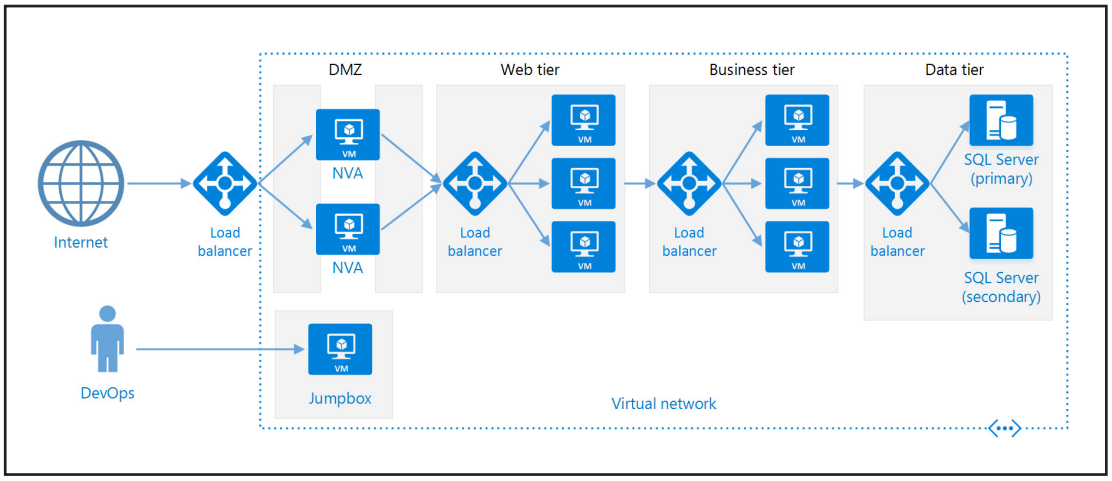
- Il est facile de se retrouver avec un niveau intermédiaire qui se contente d'effectuer des opérations de création, de lecture, de mise à jour et de suppression (Create, Read, Update et Delete ou CRUD) sur la base de données, ajoutant ainsi une latence supplémentaire sans réaliser aucun travail utile.
- La conception monolithique empêche le déploiement indépendant des fonctions.
- La gestion d'une application IaaS implique plus de travail que lorsqu'il s'agit d'une application qui utilise uniquement des services gérés.
- Il peut être difficile de gérer la sécurité du réseau dans un grand système.

Meilleures pratiques

- Utiliser la mise à l'échelle automatique pour gérer les changements de charge. Voir Meilleures pratiques de mise à l'échelle automatique.
- Utiliser la messagerie asynchrone pour découpler les niveaux.
- Mettre en cache les données semi-statiques. Voir Meilleures pratiques de mise en cache.
- Configurer un niveau de base de données pour la haute disponibilité en utilisant une solution telle que les groupes de disponibilité Always On SQL Server.
- Placer un pare-feu d'application web (WAF) entre le serveur frontal et Internet.
- Placer chaque niveau dans son propre sous-réseau et utiliser des sous-réseaux comme limite de sécurité.
- Restreindre l'accès à la couche de données en autorisant des requêtes uniquement à partir du ou des niveaux intermédiaires.

Architecture N-tier sur des machines virtuelles

Cette section décrit une architecture N-tier recommandée s'exécutant sur des machines virtuelles.



Cette section décrit une architecture N-tier recommandée s'exécutant sur des machines virtuelles. Chaque niveau est constitué de deux ou plusieurs machines virtuelles placées dans un ensemble de disponibilités ou un groupe de machines virtuelles. Plusieurs machines virtuelles assurent une résilience en cas de défaillance de l'une des machines virtuelles. Les équilibreurs de charge sont utilisés pour répartir les requêtes entre les machines virtuelles au sein d'un même niveau. Une couche peut être mise à niveau horizontalement par l'ajout de davantage de machines virtuelles au pool.

Chaque niveau est également placé à l'intérieur de son propre sous-réseau, ce qui signifie que ses adresses IP internes relèvent de la même plage d'adresses. Cela simplifie l'application des règles du groupe de sécurité réseau (network security group ou NSG) et des tables d'itinéraire à différents niveaux.

Les niveaux Web et Business sont autonomes. Toute machine virtuelle peut traiter n'importe quelle demande pour ce niveau. La couche de données doit être composée d'une base de données répliquée. Pour Windows, nous vous recommandons SQL Server en utilisant les groupes de disponibilité Always On pour une haute disponibilité. Pour Linux, choisissez une base de données qui prend en charge la réplication, telle que la base Apache Cassandra.

Les groupes de sécurité réseau (NSG) restreignent l'accès à chaque niveau. Par exemple, seul le niveau Business est autorisé à accéder au niveau de la base de données.

Pour obtenir plus de détails et un modèle Resource Manager déployable, reportez-vous aux architectures de référence suivantes :

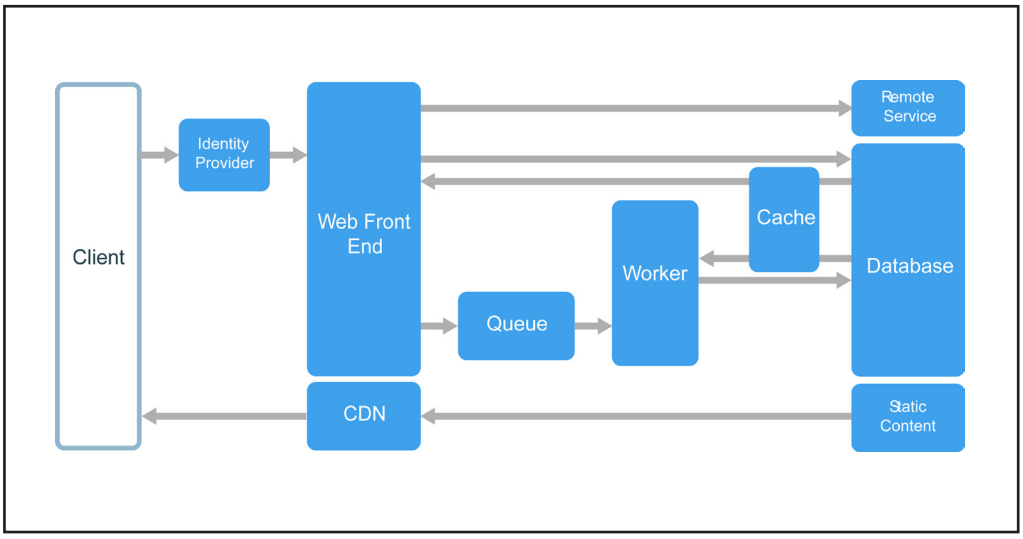
- Exécuter des machines virtuelles Windows pour une application N-tier
- Exécuter des machines virtuelles Linux pour une application N-tier

Considérations supplémentaires

- Les architectures N-tier ne se limitent pas à trois niveaux. Pour des applications plus complexes, il est fréquent de disposer de plus de niveaux. Dans ce cas, envisagez d'utiliser le routage de couche 7 pour router les requêtes vers un niveau particulier.
- Les niveaux sont la limite de l'évolutivité, de la fiabilité et de la sécurité. Pensez à disposer de niveaux distincts pour les services avec différentes exigences dans ces domaines.
- Utilisez des groupes de machines virtuelles pour la mise à l'échelle automatique.
- Recherchez des emplacements dans l'architecture où vous pouvez utiliser un service géré sans refactorisation significative. Pensez notamment à la mise en cache, à la messagerie, au stockage et aux bases de données.
- Pour une plus grande sécurité, placez un réseau DMZ en face de l'application. La zone démilitarisée comprend les applications réseau virtuelles (Network Virtual Applicance ou NVA) qui implémentent des fonctionnalités de sécurité telles que des pare-feu et l'inspection approfondie des paquets. Pour de plus amples informations, consultez Architecture de référence réseau DMZ.
- Pour la haute disponibilité, placez deux ou plusieurs appliances réseau virtuelles dans un ensemble de disponibilité avec un équilibreur de charge externe pour répartir les requêtes Internet entre les instances. Pour de plus amples informations, consultez Déployer des appliances réseau virtuelles hautement disponibles.
- N'autorisez pas un accès RDP ou SSH direct aux machines virtuelles qui exécutent du code d'application. Les opérateurs doivent plutôt se connecter à un jumpbox, également appelé bastion. Il s'agit d'une machine virtuelle sur le réseau que les administrateurs utilisent pour se connecter aux autres machines virtuelles. Le jumpbox a un groupe de sécurité réseau qui autorise RDP ou SSH uniquement à partir des adresses IP publiques approuvées.
- Vous pouvez étendre le réseau virtuel Azure à votre réseau local à l'aide d'un réseau privé virtuel (VPN) Site-to-Site ou Azure ExpressRoute. Pour de plus amples informations, consultez Architecture de référence pour le réseau hybride.
- Si votre organisation utilise Active Directory pour gérer les identités, vous souhaitez peut-être étendre votre environnement Active Directory au réseau virtuel Azure. Pour de plus amples informations, consultez Architecture de référence pour la gestion des identités.
- Si vous avez besoin d'une plus grande disponibilité que celle fournie par le SLA Azure pour les machines virtuelles, répliquez l'application entre les deux régions et utilisez Azure Traffic Manager pour le basculement. Pour de plus amples informations, consultez Exécuter des machines virtuelles Windows dans plusieurs régions ou Exécuter des machines virtuelles Linux dans plusieurs régions.

Style d'architecture Web-File d'attente-Agent de travail

Les composants principaux de cette architecture sont un serveur web frontal qui traite les demandes des clients et un agent de travail qui exécute des tâches gourmandes en ressources, des workflows de longue durée ou des tâches par lots. Le serveur web frontal communique avec l'agent de travail via une file d'attente de messages.



Parmi les autres composants couramment incorporés à cette architecture, on trouve :

- Une ou plusieurs bases de données.
- Un cache pour stocker des valeurs de la base de données pour des lectures rapides.
- Un CDN pour le contenu statique.

Le serveur web et l'agent de travail sont tous deux autonomes. L'état de la session peut être stocké dans un cache distribué. Toutes les tâches de longue durée sont réalisées de façon asynchrone par l'agent de travail. L'agent de travail peut être déclenché par des messages dans la file d'attente ou s'exécuter selon un calendrier pour le traitement par lots. L'agent de travail est un composant facultatif. S'il n'y a aucune opération de longue durée, l'agent de travail peut être omis.

Le serveur frontal peut être constitué d'une API Web. Côté client, l'API Web peut être consommée par une application monopage qui effectue des appels AJAX ou par une application cliente native.

Quand utiliser cette architecture

L'architecture Web-File d'attente-Agent de travail est généralement implémentée à l'aide de services de calcul gérés (Azure App Service ou Azure Cloud Services).

Pensez à ce style d'architecture pour les cas suivants :

- Applications avec un domaine relativement simple.
- Applications avec des workflows de longue durée ou des opérations par lots.
- Lorsque vous souhaitez utiliser des services gérés plutôt que l'infrastructure en tant que service (Infrastructure as a service ou IaaS).

Avantages

- Architecture relativement simple et facile à comprendre.
- Simplicité de déploiement et de gestion.
- Séparation claire des préoccupations.
- Le serveur frontal est découplé de l'agent de travail à l'aide de la messagerie asynchrone.
- Le serveur frontal et l'agent de travail sont évolutifs de façon indépendante.

Problématique

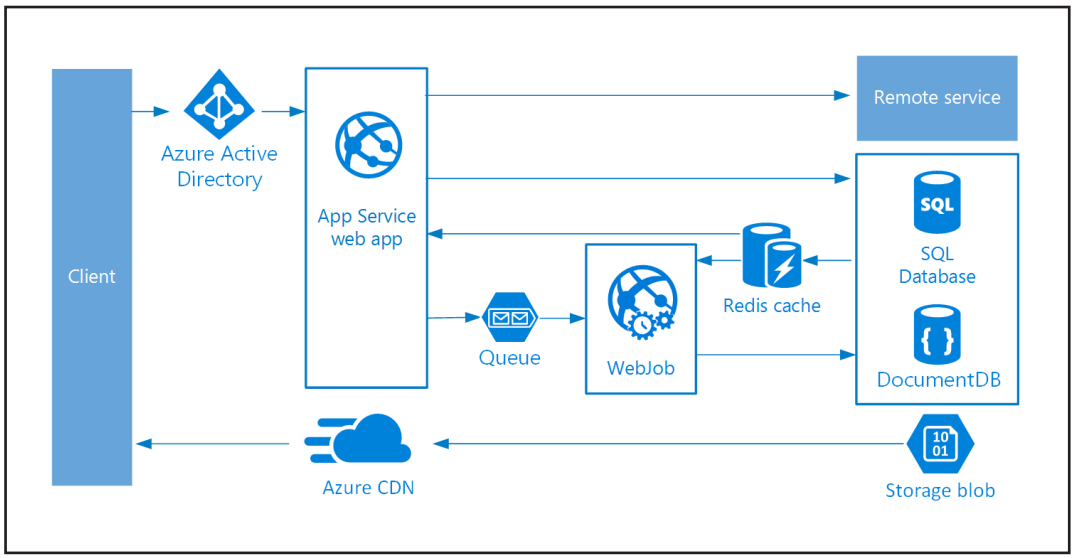
- Sans une conception soignée, le serveur frontal et l'agent de travail risquent de devenir de grands composants monolithiques difficiles à gérer et à mettre à jour.
- Il peut y avoir des dépendances cachées si le serveur frontal et l'agent de travail partagent des schémas de données ou les modules de code.

Meilleures pratiques

- Utilisez la persistance polyglotte lorsque cela est approprié. Reportez-vous à l'utilisation du meilleur magasin de données pour la tâche.
- Pour découvrir les articles sur les meilleures pratiques qui fournissent des indications précises sur la mise à l'échelle automatique, la mise en cache, le partitionnement de données, la conception d'API et autres, rendez-vous sur la page https://docs.microsoft.com/en-us/azure/architecture/best-practices/index.

Web-File d'attente-Agent de travail sur Azure App Service

Cette section décrit une architecture Web-File d'attente-Agent de travail recommandée qui utilise Azure App Service.



Le serveur frontal est implémenté comme une application Azure App Service, et l'agent de travail est implémenté comme un WebJob. L'application web et le WebJob sont tous deux associés à un plan App Service qui fournit les instances des machines virtuelles.

Vous pouvez utiliser les files d'attente Azure Service Bus ou Azure Storage pour la file d'attente de messages. (Le diagramme montre une file d'attente Azure Storage.)

Azur Redis Cache stocke l'état de la session et d'autres données nécessitant un accès à faible latence.

Azure CDN est utilisé pour mettre en cache les contenus statiques tels que les images, CSS et HTML.

Pour le stockage, choisissez les technologies de stockage qui conviennent le mieux aux besoins de l'application. Vous pouvez utiliser plusieurs technologies de stockage (persistance polyglotte). Pour illustrer cette idée, le diagramme montre Azure SQL Database et Azure Cosmos DB.

Pour plus d'informations, consultez l'architecture de référence pour les applications web gérées.

Considérations supplémentaires

- Toutes les transactions ne passent pas par la file d'attente et l'agent de travail en vue du stockage. Le serveur web frontal peut effectuer directement des opérations simples de lecture/écriture. Les agents de travail sont conçus pour des tâches exigeantes en ressources ou pour des workflows de longue durée. Dans certains cas, vous n'avez pas besoin d'agent de travail du tout.

- Utilisez la fonction de mise à l'échelle automatique intégrée d'App Service pour faire évoluer le nombre d'instances de machines virtuelles. Si la charge sur l'application suit des schémas prévisibles, utilisez la mise à l'échelle automatique basée sur la planification. Si la charge est imprévisible, utilisez des règles de mise à l'échelle automatique basées sur des métriques.

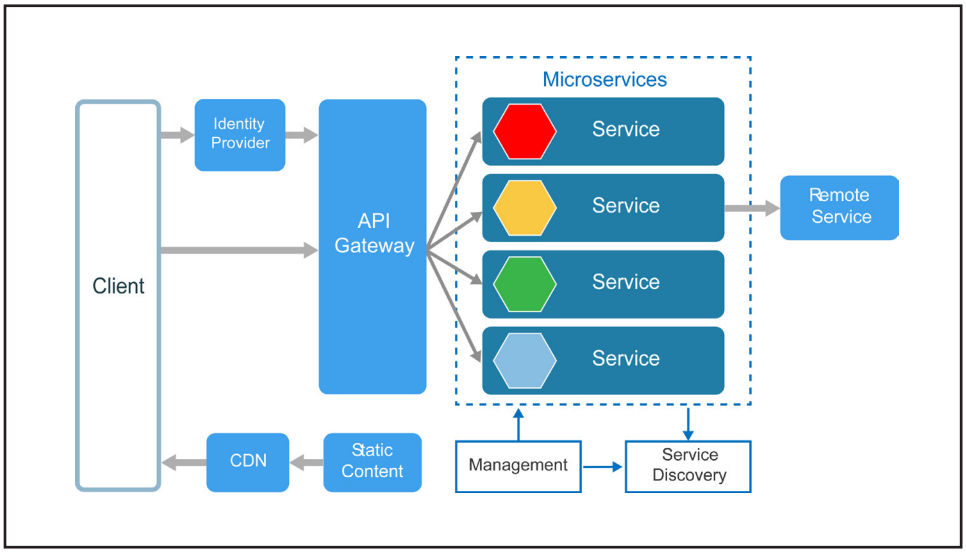
- Envisagez de placer l'application web et le WebJob dans des plans App Service distincts. De cette façon, ils sont hébergés sur des instances distinctes de machines virtuelles et peuvent être mis à l'échelle indépendamment.

- Utilisez des plans App Service distincts pour les phases de production et de test. Autrement, si vous utilisez le même plan pour la production et les tests, cela signifie que vos tests s'exécutent sur vos machines virtuelles de production.

- Utilisez des emplacements de déploiement pour gérer les déploiements. Cela permet de déployer une version mise à jour vers un emplacement de test, puis de basculer sur la nouvelle version. Cela vous permet également de rebasculer sur la version précédente en cas de problème au niveau de la mise à jour.

Architecture de microservices

Une architecture de microservices se compose d'une collection de petits services autonomes. Chaque service est autonome et doit mettre en œuvre une fonctionnalité d'entreprise unique.



À certains égards, les microservices représentent l'évolution naturelle des architectures orientées services (SOA), mais il y a des différences entre les microservices et la SOA. Voici quelques caractéristiques qui permettent de définir un microservice :

- Dans une architecture de microservices, les services sont de petite taille, indépendants et faiblement couplés entre eux.
- Chaque service constitue une base de code distincte, qui peut être gérée par une équipe de développement restreinte.
- Les services peuvent être déployés de manière indépendante. Une équipe peut mettre à jour un service existant sans avoir à recréer et à redéployer toute l'application.
- Les services sont eux-mêmes chargés de la persistance de leurs données ou de leur état externe, à la différence du modèle classique, où la persistance des données est gérée par une couche de données distincte.

- Les services communiquent entre eux à l'aide d'API bien définies. Les détails de la mise en œuvre interne de chaque service ne sont pas visibles par les autres services.
- Les services n'ont pas besoin de présenter une pile technologique, des bibliothèques ou des infrastructures identiques.

Outre les services eux-mêmes, une architecture de microservices types présente un certain nombre d'autres composants :

Gestion. Le composant de gestion est chargé notamment de la mise en place des services sur les nœuds, de l'identification des défaillances et du rééquilibrage des services entre les nœuds.

Découverte des services. Assure la conservation d'une liste des services et des nœuds où ils se trouvent. Permet la consultation des services pour trouver le point de terminaison d'un service.

Passerelle API. La passerelle API constitue le point d'entrée pour les clients. Les clients n'appellent pas les services directement. Au lieu de cela, ils appellent la passerelle API, qui transfère l'appel aux services appropriés sur le back end. La passerelle API peut agréger les réponses de plusieurs services et renvoyer la réponse agrégée.

L'utilisation d'une passerelle API offre notamment les avantages suivants :

- Elle dissocie les clients des services. Les services peuvent être versionnés ou réusinés sans nécessiter la mise à jour de tous les clients.
- Les services peuvent utiliser des protocoles de messagerie qui ne sont pas compatibles avec le web, comme AMQP.
- La passerelle API peut exécuter d'autres fonctions transversales, telles que l'authentification, la journalisation, la terminaison SSL et l'équilibrage de charge.

Quand utiliser cette architecture

Pensez à ce style d'architecture pour les cas suivants :

- Les applications de grande taille qui nécessitent une vitesse de publication élevée.
- Les applications complexes qui doivent être hautement évolutives.
- Les applications qui présentent des domaines riches ou de nombreux sous-domaines.
- Une organisation composée d'équipes de développement restreintes.

Avantages

- **Déploiements indépendants.** Vous pouvez mettre à jour un service sans redéployer toute l'application, et annuler ou restaurer par progression une mise à jour en cas de problème. Les résolutions de bogues et les publications de fonctionnalités sont plus faciles à gérer et moins risquées.
- **Développement indépendant.** Une équipe de développement unique peut créer, tester et déployer un service. Cela se traduit par une innovation continue et un rythme de publication plus élevé.

- **Équipes restreintes et appliquées.** Les équipes peuvent se concentrer sur un seul service. Du fait de l'étendue réduite de chaque service, la base de code est plus simple à comprendre et les nouveaux membres d'équipe peuvent maîtriser plus facilement les choses.

- **Isolation des pannes.** Si un service connaît une défaillance, cela n'entraîne pas la mise hors ligne complète de l'application. Toutefois, vous ne bénéficiez pas pour autant automatiquement de la résilience de l'application. Vous devez toujours suivre les meilleures pratiques et les patrons de conception en matière de résilience. Pour plus d'informations, consultez Conception d'applications résilientes pour Azure.

- **Piles technologiques mixtes.** Les équipes peuvent choisir la technologie la mieux adaptée à leur service.

- **Mise à l'échelle granulaire.** Les services peuvent être mis à l'échelle indépendamment. Par ailleurs, la plus forte densité de services par machine virtuelle signifie que les ressources de machines virtuelles sont pleinement utilisées. À l'aide de contraintes de positionnement, il est possible d'associer un service à un profil de machine virtuelle (processeur rapide, mémoire élevée, etc.).

Problématiques

- **Complexité.** Une application basée sur des microservices présente plus d'éléments mobiles que l'application monolithique équivalente. Chaque service est plus simple, mais le système dans son ensemble est plus complexe.

- **Développement et test.** Le développement en fonction des dépendances de services exige une approche différente. Les outils existants ne sont pas forcément conçus pour fonctionner avec les dépendances de services. La refactorisation au-delà des limites des services peut s'avérer complexe. Il est également difficile de tester les dépendances de services, surtout quand l'application évolue rapidement.

- **Manque de gouvernance.** L'approche décentralisée associée à la création de microservices présente des avantages, mais elle peut aussi entraîner des problèmes. Vous pouvez vous retrouver avec un tel nombre de langues et d'infrastructures différentes que l'application devient difficile à mettre à jour. Il peut être utile de mettre en place des normes à l'échelle du projet, sans restreindre outre mesure la flexibilité des équipes. Cela s'applique tout particulièrement aux fonctionnalités transversales, telles que la journalisation.

- **Surcharge et latence du réseau.** L'utilisation de nombreux services de petite taille et granulaires peut donner lieu à une communication interservices accrue. En outre, si la chaîne de dépendances de services devient trop longue (le service A appelle le service B, qui appelle le service C, etc.), la latence supplémentaire peut devenir problématique. Vous devrez concevoir les API avec soin. Évitez les API trop « bavardes », prenez en considération les formats de sérialisation et cherchez des endroits où utiliser des patrons de communication asynchrone.

- **Intégrité des données.** Chaque microservice est lui-même chargé de la persistance de ses données. Par conséquent, la cohérence des données peut s'avérer difficile. Quand cela est possible, adoptez un patron de cohérence éventuelle.

- **Gestion.** La réussite de la mise en œuvre de microservices passe par une culture DevOps mature. La journalisation corrélée pour l'ensemble des services peut être complexe. En général, la journalisation doit mettre en corrélation plusieurs appels de service pour une même opération utilisateur.

- **Gestion des versions.** Les mises à jour d'un service ne doivent pas interrompre les services qui en dépendent. Comme plusieurs services peuvent être mis à jour à tout moment, sans une conception soignée, vous pourriez avoir des problèmes de compatibilité descendante ou ascendante.

- **Compétences.** Les microservices sont des systèmes hautement distribués. Évaluez soigneusement si l'équipe possède les compétences et l'expérience requises pour réussir.

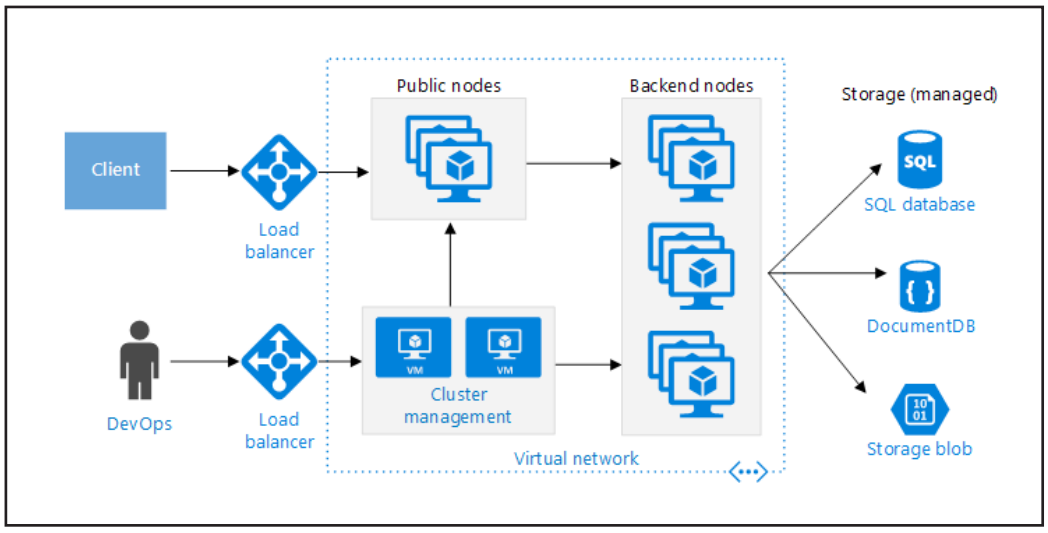
Meilleures pratiques

- Modélisez les services autour du domaine de l'entreprise.
- Décentralisez tout. Des équipes individuelles sont chargées de concevoir et de créer les services. Évitez le partage de code ou de schémas de données.
- Le stockage de données doit être accessible uniquement au service auquel les données appartiennent. Utilisez le meilleur stockage pour chaque type de service et de données.
- Les services communiquent via des API bien conçues. Évitez les fuites des détails de la mise en œuvre. Les API doivent modéliser le domaine, et non la mise en œuvre interne du service.
- Évitez le couplage entre les services. Le couplage peut notamment être dû à des schémas de base de données partagés et à des protocoles de communication rigides.
- Confiez les responsabilités transversales, telles que l'authentification et la terminaison SSL, à la passerelle.
- Gardez les connaissances du domaine hors de la passerelle. La passerelle doit gérer et acheminer les requêtes de clients sans aucune connaissance des règles métier ou de la logique de domaine. Dans le cas contraire, la passerelle devient une dépendance et peut causer un couplage entre les services.
- Les services doivent présenter un couplage faible et une forte cohésion fonctionnelle. Les fonctions qui sont susceptibles de changer en même temps doivent être empaquetées et déployées ensemble. S'ils résident dans des services distincts, ces services se retrouvent fortement couplés, car un changement de l'un d'eux exigera la mise à jour de l'autre. Une communication trop importante entre deux services peut être un signe de couplage fort et de faible cohésion.
- Isolez les défaillances. Utilisez des stratégies de résilience afin d'empêcher que les défaillances au sein d'un service n'entraînent une réaction en chaîne. Pour plus d'informations, consultez Conception d'applications résilientes pour Azure.

Pour obtenir une liste et une synthèse des patrons de résilience disponibles dans Azure, accédez à https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency.

Microservices avec Azure Container Service

Vous pouvez utiliser Azure Container Service pour configurer et approvisionner un cluster Docker. Azure Container Service prend en charge plusieurs orchestrateurs de conteneurs populaires, dont Kubernetes, DC/OS et Docker Swarm.



Nœuds publics. Ces nœuds sont accessibles via un équilibreur de charge public. La passerelle API est hébergée sur ces nœuds.

Nœuds principaux. Ces nœuds exécutent les services auxquels les clients accèdent via la passerelle API. Ces nœuds ne reçoivent pas directement de trafic Internet. Les nœuds principaux peuvent inclure plusieurs pools de machines virtuelles, chacun avec un profil matériel différent. Par exemple, vous pouvez créer des pools distincts pour les scénarios d'usage de calcul général, les scénarios d'usage impliquant une utilisation intensive du processeur et les scénarios d'usage impliquant une utilisation élevée de la mémoire.

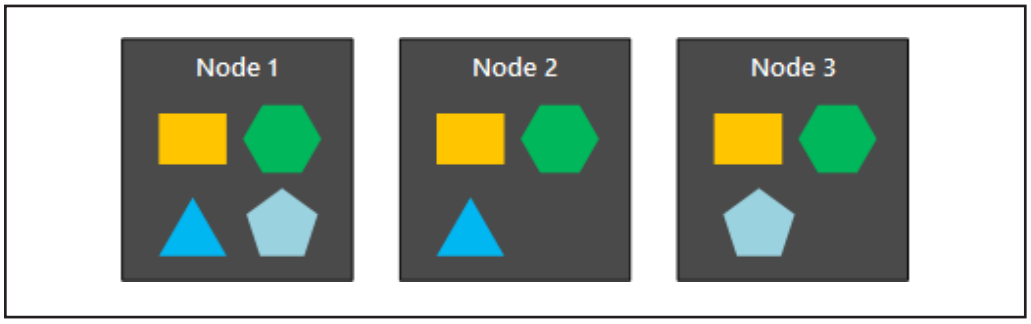
Machines virtuelles de gestion. Ces machines virtuelles exécutent les nœuds maîtres pour l'orchestrateur de conteneurs.

Mise en réseau. Les nœuds publics, les nœuds principaux et les machines virtuelles de gestion sont placés dans des sous-réseaux distincts au sein du même réseau virtuel.

Équilibreurs de charge. Un équilibreur de charge externe se trouve devant les nœuds publics. Il distribue les requêtes Internet aux nœuds publics. Un autre équilibreur de charge est placé devant les machines virtuelles de gestion pour permettre le trafic Secure Shell (SSH) vers ces dernières à l'aide de règles NAT.

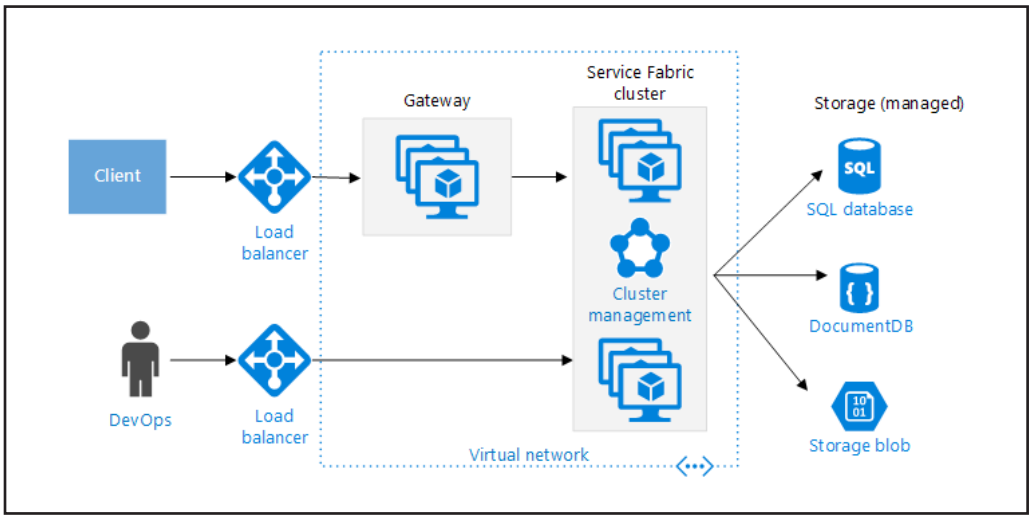
À des fins de fiabilité et d'évolutivité, chaque service est répliqué sur plusieurs machines virtuelles. Cependant, étant donné que les services sont relativement légers (par rapport à une application monolithique), plusieurs services sont généralement placés dans une même machine virtuelle. La densité plus élevée assure une meilleure utilisation des ressources. Si un service spécifique n'utilise pas beaucoup de ressources, vous n'avez pas besoin de consacrer une machine virtuelle complète à l'exécution de ce service.

Le diagramme suivant illustre trois nœuds exécutant quatre services différents (représentés par des formes différentes). Vous remarquerez que chaque service présente au moins deux instances.



Microservices avec Azure Service Fabric

Le diagramme suivant illustre une architecture de microservices avec Azure Service Fabric.



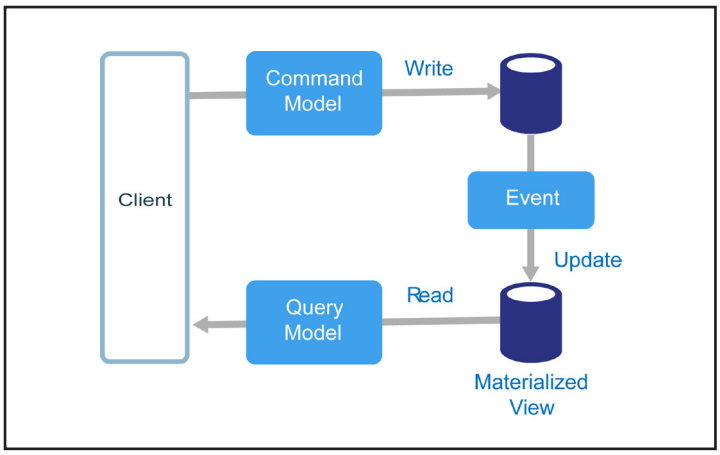
Le cluster Service Fabric est déployé dans un ou plusieurs groupes de machines virtuelles identiques. Vous pouvez inclure plus d'un groupe de machines virtuelles identiques dans le cluster afin de disposer de différents types de machines virtuelles. Une passerelle API est placée devant le cluster Service Fabric, avec un équilibreur de charge externe pour recevoir les requêtes de clients.

Le runtime Service Fabric accomplit les tâches de gestion du cluster, y compris le positionnement des services, le basculement entre les nœuds et la vérification du fonctionnement. Le runtime est déployé sur les nœuds du cluster eux-mêmes. Il n'y a pas d'ensemble distinct de machines virtuelles pour la gestion du cluster.

Les services communiquent entre eux à l'aide du proxy inverse intégré dans Service Fabric. Service Fabric offre un service de découverte capable de résoudre le point de terminaison pour un service nommé.

Architecture CQRS

CQRS (Command and Query Responsibility Segregation) est un style d'architecture qui sépare les opérations de lecture des opérations d'écriture.



Dans les architectures traditionnelles, le même modèle de données est utilisé pour interroger et mettre à jour une base de données. C'est simple et cela fonctionne bien pour les opérations de création, de lecture, de mise à jour et de suppression de base. Dans les applications plus complexes, cependant, cette approche peut devenir difficile à maîtriser. Par exemple, côté lecture, l'application peut exécuter de nombreuses requêtes différentes, renvoyant des objets de transfert de données (DTO) avec des formes différentes. Le mappage d'objets peut devenir compliqué. Côté écriture, le modèle peut mettre en œuvre une validation et une logique métier complexes. Par conséquent, vous pouvez vous retrouver avec un modèle excessivement complexe qui effectue de trop nombreuses opérations.

Un autre problème potentiel est que la lecture et l'écriture des scénarios d'usage sont souvent asymétriques, avec des exigences de performance et d'échelle très différentes.

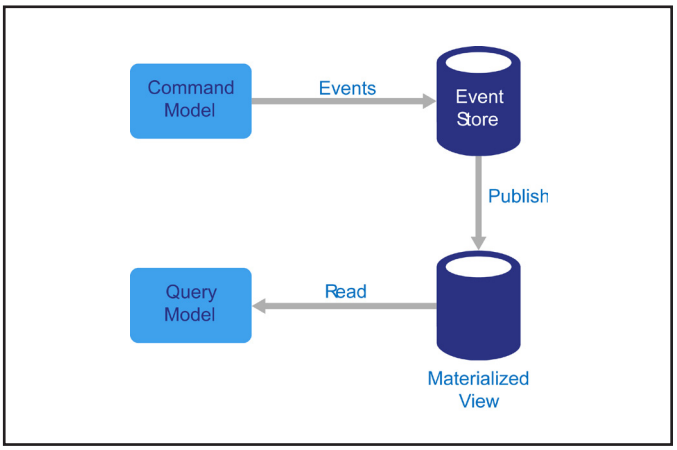
L'architecture CQRS résout ces problèmes en séparant les lectures et les écritures dans des modèles distincts, et en utilisant des commandes pour mettre à jour les données et des requêtes pour lire ces dernières.

- Les commandes doivent être basées sur les tâches plutôt que centrées sur les données (par exemple, « réserver chambre d'hôtel », et non pas « définir ÉtatRéservation sur Réservé »). Les commandes peuvent être placées dans une file d'attente à des fins de traitement asynchrone au lieu d'être traitées de façon synchrone.
- Les requêtes ne modifient jamais la base de données. Une requête renvoie un objet DTO qui n'encapsule aucune connaissance du domaine.

Pour accroître l'isolation, vous pouvez séparer physiquement les données en lecture des données en écriture. Dans ce cas, la base de données en lecture peut utiliser son propre schéma de données, qui est optimisé pour les requêtes. Par exemple, elle peut stocker une vue matérialisée des données afin d'éviter des jointures ou des mappages objet/relationnel complexes. Elle peut même utiliser un autre type de banque de données. Par exemple, la base de données en écriture peut être relationnelle, alors que la base de données en lecture est une base de données de documents.

Si des bases de données en lecture et en écriture distinctes sont utilisées, elles doivent rester synchronisées. Cela est généralement accompli grâce à la publication d'un événement par le modèle d'écriture chaque fois qu'il met à jour la base de données. La mise à jour de la base de données et la publication de l'événement doivent se faire en une seule transaction.

Certaines implémentations de l'architecture CQRS utilisent le patron Event Sourcing (Matérialisation d'événements). Avec ce patron, l'état de l'application est stocké sous forme de séquence d'événements. Chaque événement représente un ensemble de modifications apportées aux données. L'état actuel est construit à l'aide de la relecture des événements. Dans un contexte d'architecture CQRS, l'un des avantages du patron Event Sourcing est que les mêmes événements peuvent être utilisés pour notifier les autres composants, et plus particulièrement le modèle de lecture. Le modèle de lecture se sert des événements pour créer un instantané de l'état actuel, ce qui s'avère plus efficace pour les requêtes. Cependant, le patron Event Sourcing augmente la complexité de la conception.



Quand utiliser cette architecture

Envisagez d'utiliser l'architecture CQRS pour les domaines collaboratifs où de nombreux utilisateurs accèdent aux mêmes données, en particulier quand les scénarios d'usage en lecture et en écriture sont asymétriques.

CQRS n'est pas une architecture de premier niveau qui s'applique à l'ensemble d'un système. Appliquez l'architecture CQRS uniquement aux sous-systèmes où la séparation des lectures et des écritures présente un intérêt évident. Sinon, vous engendrez inutilement une complexité supplémentaire.

Avantages

- **Mise à l'échelle indépendante.** L'architecture CQRS permet la mise à l'échelle indépendante des scénarios d'usage en lecture et en écriture, et peut réduire les contentions de verrouillage.
- **Schémas de données optimisés.** Le côté lecture peut utiliser un schéma optimisé pour les requêtes et le côté écriture un schéma optimisé pour les mises à jour.
- **Sécurité.** Il est plus facile de s'assurer que seules les entités de domaine effectuent des écritures sur les données.

- **Séparation des préoccupations.** La séparation des côtés lecture et écriture peut offrir des modèles plus faciles à gérer et plus souples. La plus grande partie de la logique métier complexe est incluse dans le modèle d'écriture. Le modèle de lecture peut être relativement simple.
- **Simplification des requêtes.** En stockant une vue matérialisée dans la base de données en lecture, l'application peut éviter des jointures complexes lors de l'interrogation.

Problématiques

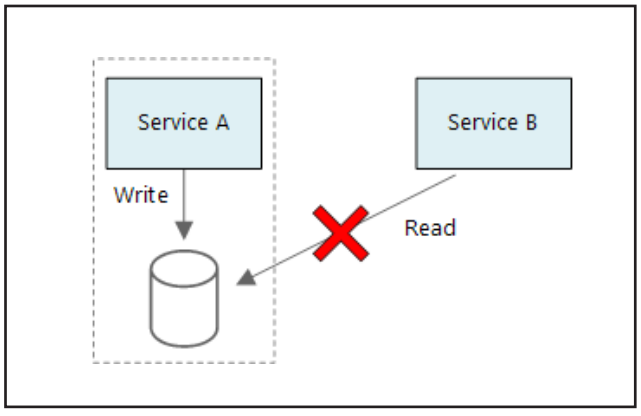
- **Complexité.** L'idée de base de l'architecture CQRS est simple. Cependant, cette architecture peut donner à une conception d'application plus complexe, en particulier si le patron Event Sourcing (Matérialisation d'événements) est inclus.
- **Messagerie.** Bien que l'architecture CQRS ne nécessite pas de messagerie, celle-ci est souvent utilisée pour traiter les commandes publier les événements de mise à jour. Dans ce cas, l'application doit gérer les échecs et les doublons de messages.
- **Cohérence à terme.** Si vous séparez les bases de données en lecture et en écriture, les données en lecture peuvent être obsolètes.

Meilleures pratiques

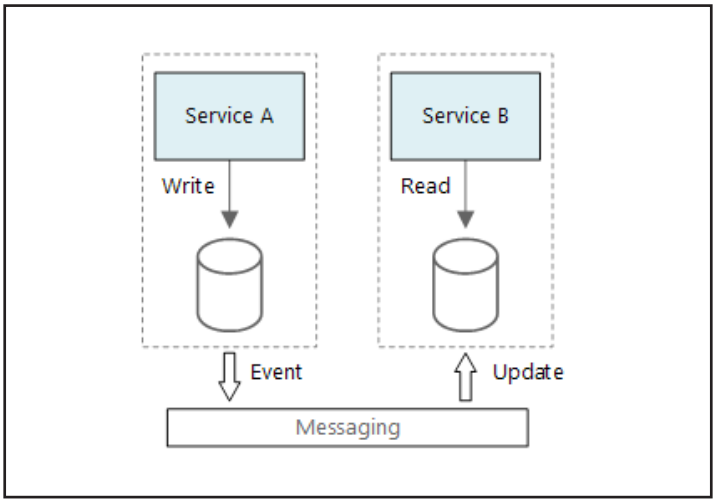
- Pour plus d'informations sur la mise en œuvre de l'architecture CQRS, accédez à https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs.
- Pour plus d'informations sur l'utilisation du patron Event Sourcing (Matérialisation d'événements) afin d'éviter les conflits de mise à jour, accédez à https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.
- Pour plus d'informations sur l'utilisation du patron Materialized View (Vue matérialisée) pour le modèle de lecture afin d'optimiser le schéma pour les requêtes, accédez à https://docs.microsoft.com/en-us/azure/architecture/patterns/materialized-view.

Architecture CQRS dans le cadre de microservices

L'architecture CQRS peut s'avérer particulièrement utile dans une architecture de microservices. L'un des principes des microservices est qu'un service ne peut pas accéder directement à la banque de données d'un autre service.

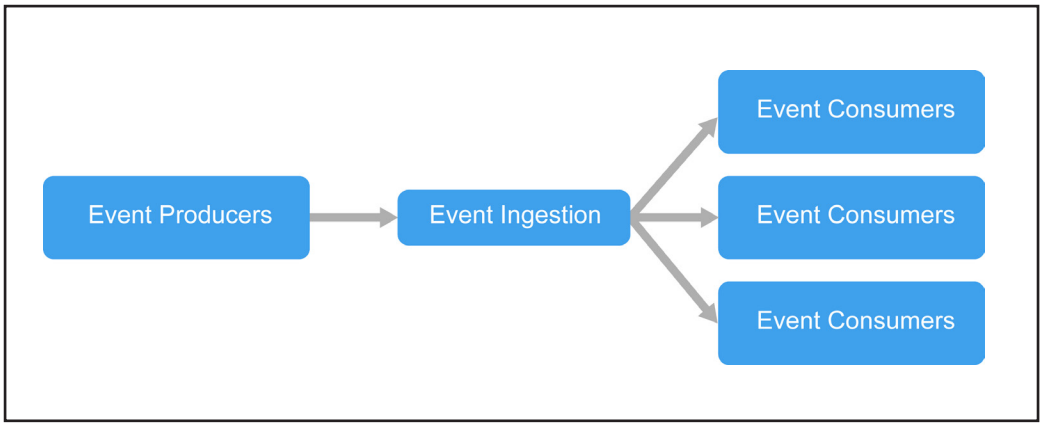


Dans le diagramme suivant, le service A écrit dans une banque de données et le service B conserve une vue matérialisée des données. Le service A publie un événement chaque fois qu'il écrit dans la banque de données. Le service B s'abonne à l'événement.



Architecture orientée événements

Une architecture orientée événements se compose de producteurs d'événements qui génèrent un flux d'événements et de consommateurs d'événements qui écoutent les événements.



Les événements sont transmis en quasi temps réel, ce qui permet aux consommateurs de répondre aux événements immédiatement après leur survenue. Les producteurs sont découplés des consommateurs : ils ne savent pas quels consommateurs écoutent les événements.
Les consommateurs sont également découplés les uns des autres, et chaque consommateur voit l'ensemble des événements. Cela diffère d'un Patron Competing Consumers (Consommateurs concurrents), où les consommateurs extraient des messages d'une file d'attente et un message n'est traité qu'une seule fois (en supposant qu'aucune erreur ne se produit). Dans certains systèmes, comme l'IoT, les événements doivent être ingérés en très grand nombre.

Une architecture orientée événements peut utiliser un modèle de publication/abonnement ou un modèle de flux d'événements.

- **Publication/abonnement :** l'infrastructure de messagerie assure le suivi des abonnements. Quand un événement est publié, il envoie l'événement à chaque abonné. Une fois un événement reçu, il ne peut pas être relu, et les nouveaux abonnés ne le voient pas.

- **Diffusion en continu des événements :** les événements sont écrits dans un journal. Ils sont strictement ordonnés (au sein d'une partition) et durables. Les clients ne s'abonnent pas au flux. Au lieu de cela, ils peuvent en lire n'importe quelle partie. Le client est chargé de la progression de sa position dans le flux. Il peut ainsi rejoindre le flux à tout moment et relire les événements.

Du côté des consommateurs, il existe des variantes courantes :

- **Traitement d'événements simple.** Un événement déclenche immédiatement une action dans le consommateur. Par exemple, vous pouvez utiliser Azure Functions avec un déclencheur Service Bus pour qu'une fonction s'exécute chaque fois qu'un message est publié dans une rubrique Service Bus.
- **Traitement d'événements complexes.** Un consommateur traite une série d'événements, en recherchant des tendances dans les données d'événements à l'aide d'une technologie telle qu'Azure Stream Analytics ou Apache Storm. Par exemple, vous pouvez agréger les lectures d'un appareil intégré sur une fenêtre de temps spécifique et générer une notification si la moyenne mobile franchit un certain seuil.
- **Traitement de flux d'événements.** Utilisez une plateforme de diffusion des données en continu, telle qu'Azure IoT Hub ou Apache Kafka, comme un pipeline pour ingérer les événements et les transmettre aux processeurs de flux. Les processeurs de flux traitent ou transforment alors le flux. Il peut y avoir plusieurs processeurs de flux pour les différents sous-systèmes de l'application. Cette approche convient bien aux scénarios d'usage liés à l'IoT.

La source des événements peut être externe au système, comme des appareils physiques dans une solution IoT. Dans ce cas, le système doit être capable d'ingérer le volume et le débit de données requis par la source de données.

Dans le schéma logique ci-dessus, chaque type de consommateur est illustré sous forme d'encadré unique. Dans la pratique, il est courant d'avoir plusieurs instances d'un consommateur afin d'éviter que le consommateur ne devienne un point de défaillance unique dans le système. Plusieurs instances peuvent également être nécessaires pour gérer le volume et la fréquence des événements. En outre, un même consommateur peut traiter des événements sur plusieurs threads. Cela peut créer des difficultés si les événements doivent être traités dans l'ordre ou requièrent la sémantique « exactly-once ». Pour plus d'informations, consultez Minimiser la coordination.

Quand utiliser cette architecture

- Plusieurs sous-systèmes doivent traiter les mêmes événements.
- Traitement en temps réel avec un décalage minimal.
- Traitement d'événements complexe, tels que le filtrage par motif ou l'agrégation sur des fenêtres de temps spécifiques.
- Volume important et débit élevé de données (IoT, par exemple).

Avantages

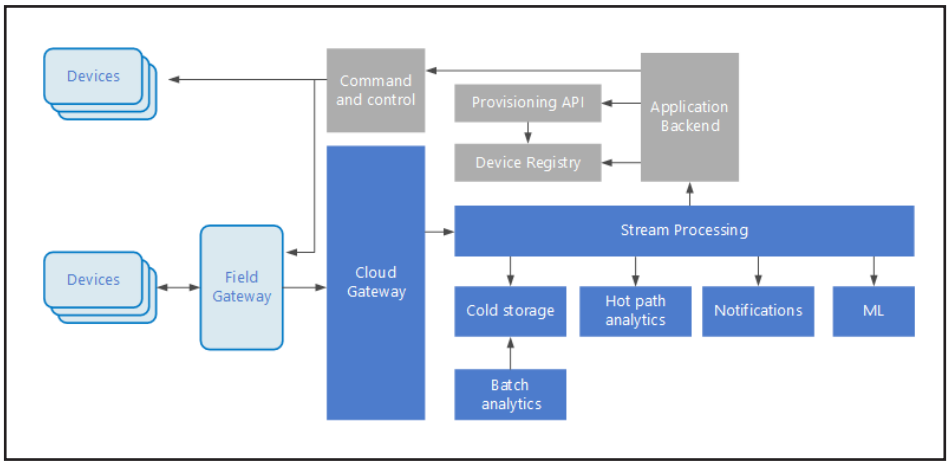
- Les producteurs et les consommateurs sont découplés.
- Aucune intégration point à point. De nouveaux consommateurs peuvent être ajoutés au système en toute simplicité.
- Les consommateurs peuvent répondre aux événements au moment même de leur arrivée.
- Cette architecture est hautement évolutive et distribuée.
- Les sous-systèmes bénéficient de vues indépendantes du flux d'événement.

Problématiques

- Livraison garantie. Dans certains systèmes, en particulier dans le cadre des scénarios IoT, il est essentiel que la livraison des événements soit garantie.
- Traitement des événements dans l'ordre ou exactement une fois. Chaque type de consommateur s'exécute généralement dans plusieurs instances à des fins de résilience et d'évolutivité. Cela peut créer des difficultés si les événements doivent être traités dans l'ordre (au sein d'un type de consommateur) ou si la logique de traitement n'est pas idempotente.

Architecture IoT

Les architectures orientées événements jouent un rôle central dans les solutions IoT. Le diagramme suivant illustre une architecture logique possible pour l'IoT. Il met l'accent sur les composants de l'architecture diffusant en continu les événements.



La **passerelle cloud** ingère les événements d'appareils à la limite du cloud à l'aide d'un système de messagerie fiable et à faible latence.

Les appareils peuvent envoyer des événements directement à la passerelle cloud ou via une passerelle de champ. Une **passerelle de champ** est un dispositif ou un logiciel spécialisé, généralement situé au même endroit que les appareils, qui reçoit les événements et les transfère à la passerelle cloud. La passerelle de champ peut également prétraiter les événements d'appareils bruts, exécutant des fonctions telles que le filtrage, l'agrégation ou la transformation de protocole.

Après l'ingestion, les événements passent par un ou plusieurs **processeurs de flux**, qui peuvent acheminer les données (par exemple, vers le stockage) ou effectuer des analyses et d'autres types de traitement.
Voici quelques types de traitement courants. (cette liste est loin d'être exhaustive) :

- Écriture des données dans un système de stockage froid à des fins d'archivage ou d'analyse par lots.
- Analyse de chemin réactif, analysant le flux d'événements en (quasi) temps réel, afin de détecter les anomalies, d'identifier les tendances sur des fenêtres de temps dynamiques ou de déclencher une alerte quand une situation spécifique se présente dans le flux.
- Manipulation des types spéciaux de messages autres que de télémétrie reçus d'appareils, tels que les notifications et les alarmes. Machine Learning.

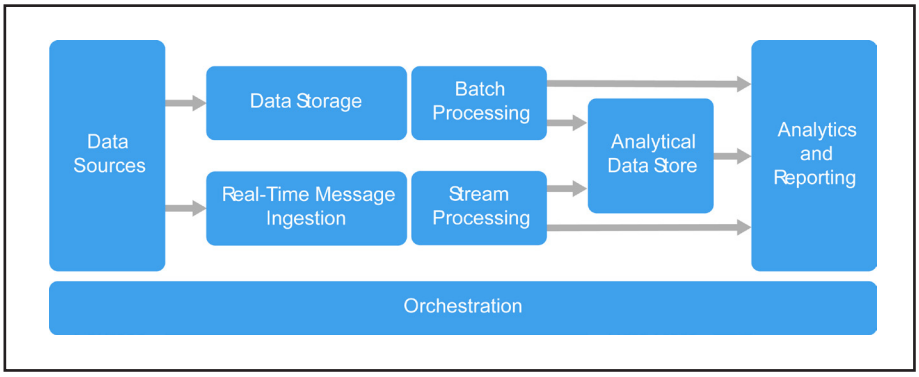
Les encadrés en gris signalent les composants d'un système IoT qui ne sont pas directement liés à la diffusion en continu des événements, mais sont inclus ici par souci d'exhaustivité.

- Le registre d'appareils est une base de données des appareils approvisionnés. Elle inclut l'ID des appareils et généralement les métadonnées de ces derniers, telles que l'emplacement.
- L'API d'approvisionnement est une interface externe commune destinée à l'approvisionnement et à l'enregistrement de nouveaux appareils.
- Certaines solutions IoT permettent l'envoi de messages de commande et de contrôle aux appareils.

Dans cette section, nous avons présenté l'IoT de façon très générale. Les subtilités et les problématiques à prendre en considération sont nombreuses. Pour en savoir plus et obtenir une architecture de référence détaillée, accédez à https://azure.microsoft.com/en-us/updates/microsoft-azure-iot-reference-architecture-available/ (téléchargement PDF).

Architecture Big Data

Une architecture Big Data est conçue pour gérer l'ingestion, le traitement et l'analyse de données qui sont trop volumineuses ou complexes pour les systèmes de base de données classiques.



Les solutions Big Data impliquent généralement un ou plusieurs des types suivants de scénarios d'usage :

- Traitement par lots des sources Big Data au repos.
- Traitement en temps réel du Big Data en mouvement.
- Exploration interactive du Big Data.
- Analyse prédictive et Machine Learning.

La plupart des architectures Big Data incluent une partie ou l'ensemble des composants suivants :

- **Sources de données :** toutes les solutions Big Data partent d'une ou plusieurs sources de données. En voici des exemples :
  - Les banques de données d'applications, telles que les bases de données relationnelles.
  - Les fichiers statiques produits par les applications, telles que les fichiers journaux de serveur web.
  - Les sources de données en temps réel, telles que les appareils IoT.

- **Stockage de données :** les données destinées aux opérations de traitement par lots sont généralement stockées dans un magasin de fichiers distribués qui peut contenir d'importants volumes de fichiers de grande taille dans différents formats. Ce type de magasin est souvent désigné sous le nom de Data Lake (ou lac de données). Les options de mise en œuvre de ce stockage incluent Azure Data Lake Store et les conteneurs d'objets blob d'Azure Storage.

- **Traitement par lots.** Comme les ensembles de données sont extrêmement volumineux, une solution Big Data doit souvent s'appuyer sur de longues tâches par lots pour filtrer, agréger et préparer de toute autre manière les données à des fins d'analyse. Ces tâches impliquent généralement la lecture et le traitement des fichiers source, ainsi que l'écriture des résultats dans de nouveaux fichiers. Les options incluent l'exécution de tâches U-SQL dans Azure Data Lake Analytics, l'utilisation de tâches Hive, Pig ou de tâches Map/Reduce personnalisées dans un cluster Hadoop HDInsight, ou encore l'utilisation de programmes Java, Scala ou Python dans un cluster Spark HDInsight.
- **Ingestion des messages en temps réel.** Si la solution inclut des sources en temps réel, l'architecture doit inclure un moyen de capturer et de stocker les messages en temps réel pour le traitement de flux. Il peut s'agir d'une simple banque de données où les messages entrants sont déposés dans un dossier aux fins de traitement. Cependant, de nombreuses solutions ont besoin d'un magasin d'ingestion de messages pour la mise en tampon de ces derniers, ainsi que pour prendre en charge le traitement par plusieurs serveurs, une livraison fiable et d'autres sémantiques de mise en file d'attente de messages. Les options incluent Azure Event Hubs, Azure IoT Hubs et Kafka.
- **Traitement de flux.** Après avoir capturé les messages en temps réel, la solution doit les traiter en filtrant, en agrégeant et en préparant de toute autre manière les données à des fins d'analyse. Les données de flux traitées sont ensuite écrites dans un récepteur de sortie. Azure Stream Analytics fournit un service géré de traitement de flux basé sur des requêtes SQL exécutées en permanence sur des flux continus. Vous pouvez également utiliser des technologies de diffusion en continu Apache, telles que Storm et Spark Streaming, dans un cluster HDInsight.
- **Banque de données analytique.** Un grand nombre de solutions Big Data préparent les données à des fins d'analyse, puis remettent les données traitées dans un format structuré qui peut être interrogé à l'aide d'outils analytiques. La banque de données analytique utilisée pour émettre ces requêtes peut être un entrepôt de données relationnelles tel que Kimball, comme dans la plupart des solutions de décisionnel classiques. Les données peuvent également être présentées par le biais d'une technologie NoSQL à faible latence, comme HBase, ou d'une base de données Hive interactive qui applique une abstraction de métadonnées sur les fichiers de données dans la banque de données distribuées. Azure SQL Data Warehouse fournit un service géré pour l'entreposage de données à grande échelle dans le cloud. HDInsight prend en charge Interactive Hive, HBase et Spark SQL, qui peut également être utilisé pour fournir les données en vue de leur analyse.
- **Analyse et création de rapports.** La plupart des solutions Big Data sont conçues pour tirer des renseignements des données à travers l'analyse et la création de rapports. Pour permettre aux utilisateurs d'analyser les données, l'architecture peut inclure une couche de modélisation des données, comme un cube OLAP multidimensionnel ou un modèle de données tabulaires dans Azure Analysis Services. Elle peut également prendre en charge le décisionnel libre-service en s'appuyant sur les technologies de modélisation et de visualisation de Microsoft Power BI ou Microsoft
- **Excel.** L'analyse et la création de rapports peuvent également prendre la forme d'une exploration interactive de données par des spécialistes ou des analystes des données. Pour ces scénarios, de nombreux services Azure prennent en charge les blocs-notes analytiques, tels que Jupyter, ce qui permet à ces utilisateurs de tirer profit de leurs compétences actuelles en matière de langage Python ou R. Pour l'exploration de données à grande échelle, vous pouvez utiliser Microsoft R Server, soit de façon autonome, soit avec Spark.
- **Orchestration.** La plupart des solutions Big Data s'appuient sur des opérations répétées de traitement de données, encapsulées dans des workflows, qui transforment les données source, déplacent les données entre plusieurs sources et récepteurs, chargent les données traitées dans une banque de données analytiques, ou transfèrent les résultats directement dans un rapport ou un tableau de bord. Pour automatiser ces workflows, vous pouvez faire appel à une technologie d'orchestration telle qu'Azure Data Factory ou Apache Oozie et Sqoop.

Azure inclut de nombreux services qui peuvent être utilisés dans une architecture Big Data. Schématiquement, ils se divisent en deux catégories :

- Les services gérés, parmi lesquels Azure Data Lake Store, Azure Data Lake Analytics, Azure Data Warehouse, Azure Stream Analytics, Azure Event Hub, Azure IoT Hub et Azure Data Factory.

- Les technologies open source basées sur la plateforme Apache Hadoop, qui comprennent HDFS, HBase, Hive, Pig, Spark, Storm, Oozie, Sqoop et Kafka. Ces technologies sont disponibles dans Azure par l'intermédiaire du service Azure HDInsight.

Ces options ne s'excluent pas mutuellement, et de nombreuses solutions associent technologies open source et services Azure.

Avantages

- **Large choix de technologies.** Vous pouvez combiner les services gérés Azure et les technologies Apache à votre guise dans des clusters HDInsight afin de mettre à profit des compétences ou des investissements technologiques existants.
- **Performances grâce au parallélisme.** Les solutions Big Data tirent parti du parallélisme, qui permet de bénéficier de solutions hautes performances capables de prendre en charge de gros volumes de données.
- **Mise à l'échelle élastique.** Tous les composants de l'architecture Big Data prennent en charge l'évolutivité horizontale de l'approvisionnement. Vous pouvez ainsi adapter votre solution aux petits ou aux grands scénarios d'usage et payer uniquement pour les ressources que vous utilisez.
- **Interopérabilité avec les solutions existantes.** Les composants de l'architecture Big Data sont également utilisés pour les solutions de traitement IoT et de décisionnel d'entreprise, vous permettant de créer une solution intégrée pour l'ensemble des scénarios d'usage de données.

Problématiques

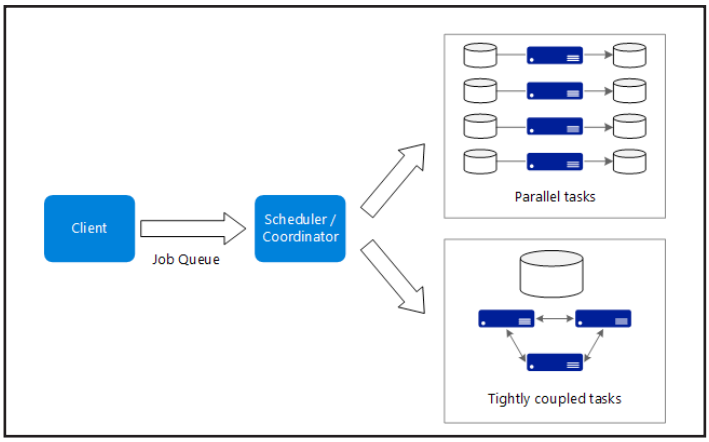
- **Complexité.** Les solutions Big Data peuvent être extrêmement complexes, associant de nombreux composants pour gérer l'ingestion des données à partir de multiples sources de données. La création, les tests et la résolution des problèmes de processus Big Data peuvent s'avérer difficiles. En outre, il peut y avoir un grand nombre de paramètres de configuration sur différents systèmes qui doivent être utilisés afin d'optimiser les performances.
- **Compétences.** De nombreuses technologies Big Data sont très spécialisées et font appel à des infrastructures et des langages qu'on ne retrouve pas dans les architectures d'applications plus générales. Cependant, les technologies Big Data évoluent avec de nouvelles API qui s'appuient sur des langages plus établis. Par exemple, le langage U-SQL utilisé dans Azure Data Lake Analytics repose sur une combinaison des langages Transact-SQL et C#. De même, des API basées sur SQL sont disponibles pour Hive, HBase et Spark.
- **Maturité technologique.** Un grand nombre des technologies utilisées pour le Big Data évoluent. Si les technologies Hadoop de base comme Hive et Pig se sont stabilisées, les technologies émergentes telles que Spark introduisent des modifications et des améliorations importantes à chaque nouvelle version. Les services gérés comme Azure Data Lake Analytics et Azure Data Factory sont relativement jeunes par rapport aux autres services Azure, et ils évolueront probablement au fil du temps.
- **Sécurité.** Les solutions Big Data s'appuient généralement sur le stockage de toutes les données statiques dans un Data Lake centralisé. La sécurisation de l'accès à ces données peut être difficile, surtout quand les données doivent être ingérées et consommées par de multiples applications et plateformes.

Meilleures pratiques

- **Tirez profit du parallélisme.** La plupart des technologies Big Data répartissent le scénario d'usage entre plusieurs unités de traitement. Pour ce faire, les fichiers de données statiques doivent être créés et stockés dans un format divisible. Les systèmes de fichiers DFS (Distributed File System) tels que HDFS permettent d'optimiser la vitesse de lecture et d'écriture. Par ailleurs, le traitement proprement dit est effectué par plusieurs nœuds de cluster en parallèle, ce qui réduit la durée globale des tâches.
- **Partitionnez les données.** Le traitement par lots se fait généralement à intervalles réguliers — par exemple, chaque semaine ou chaque mois. Partitionnez les fichiers de données et les structures de données, telles que les tables, selon des périodes correspondant à la planification du traitement. Vous simplifierez ainsi l'ingestion des données et la planification des tâches, de même que la résolution des défaillances. Par ailleurs, le partitionnement des tables utilisées dans les requêtes Hive, U-SQL ou SQL peut améliorer considérablement les performances en matière de requêtes.
- **Appliquez la sémantique « schema-on-read ».** L'utilisation d'un Data Lake vous permet de stocker des fichiers dans différents formats, qu'ils soient structurés, semi-structurés ou non structurés. Faites appel à la sémantique « schema-on-read », qui projette un schéma sur les données quand celles-ci sont en cours de traitement, et non quand elles sont stockées. De cette manière, vous disposerez d'une solution flexible et éliminerez les goulots d'étranglement causés par la validation des données et le contrôle de type pendant l'ingestion des données.
- **Traitez les données sur place.** Les solutions de décisionnel classiques s'appuient souvent sur un processus d'extraction, de transformation et de chargement (ETL) pour déplacer les données dans un entrepôt de données. Avec des volumes plus importants de données et une plus grande diversité de formats, les solutions Big Data de données utilisent des variantes du processus ETL, telles que la transformation, l'extraction et le chargement (TEL). Cette approche permet de traiter les données dans la banque de données distribuées, où elles sont transformées selon la structure requise, puis de déplacer les données transformées dans une banque de données analytiques.
- **Équilibrez les coûts liés à l'utilisation et au temps de traitement.** Pour les tâches de traitement par lots, il est important de tenir compte de deux facteurs : le coût unitaire des nœuds de calcul et le coût par minute associé à l'utilisation de ces nœuds pour exécuter la tâche. Par exemple, une tâche de traitement par lots peut prendre huit heures avec quatre nœuds de cluster. Cependant, il peut s'avérer que la tâche utilise les quatre nœuds uniquement pendant les deux premières heures et qu'après cela, seulement deux nœuds soient nécessaires. Dans ce cas, l'exécution de l'intégralité de la tâche sur deux nœuds augmenterait la durée globale de la tâche, mais elle ne la doublerait pas. Par conséquent, le coût total serait inférieur. Dans certains scénarios métier, un temps de traitement plus long peut être préférable au coût plus élevé engendré par la sous-utilisation des ressources de cluster.
- **Séparez les ressources de cluster.** Quand vous déployez des clusters HDInsight, vous obtiendrez normalement de meilleures performances en provisionnant des ressources de cluster distinctes pour chaque type de scénario d'usage. Par exemple, bien que les clusters Spark incluent Hive, si vous avez besoin d'effectuer un traitement de grande ampleur avec à la fois Hive et Spark, vous devez envisager de déployer des clusters Spark et Hadoop dédiés. De même, si vous utilisez HBase et Storm pour le traitement de flux à faible latence et Hive pour le traitement par lots, envisagez de tirer parti de clusters Storm, HBase et Hadoop distincts.
- **Orchestrez l'ingestion des données.** Dans certains cas, les applications métier existantes peuvent écrire les fichiers de données destinés au traitement par lots directement dans des conteneurs d'objets blob Azure Storage, où ils peuvent être consommés par HDInsight ou Azure Data Lake Analytics. Cependant, vous devrez souvent orchestrer l'ingestion des données issues des sources de données locales ou externes dans le Data Lake. Pour y parvenir de façon prévisible, avec une gestion centralisée, utilisez un workflow ou un pipeline d'orchestration, comme ceux pris en charge par Azure Data Factory ou Oozie.
- **Nettoyez les données sensibles de façon précoce.** Dans le cadre du workflow d'ingestion de données, les données sensibles doivent être nettoyées de façon précoce dans le processus, afin d'éviter leur stockage dans le lac de données.

Style d'architecture Big Compute

Le terme Big Compute désigne des scénarios d'usage à grande échelle qui nécessitent un grand nombre de cœurs, se chiffrant souvent en centaines ou en milliers. Ces scénarios incluent notamment le rendu d'image, la dynamique des fluides, la modélisation du risque financier, l'exploration pétrolière, la conception de médicaments et l'analyse des contraintes en ingénierie.



Voici quelques caractéristiques types des applications Big Compute :

- Le travail peut être scindé en tâches distinctes, qui peuvent être exécutées simultanément sur un grand nombre de cœurs.
- Chaque tâche est finie. Le processus utilise certaines données d'entrée, effectue un traitement et produit une sortie. L'ensemble de l'application s'exécute pour une période déterminée (allant de plusieurs minutes à plusieurs jours). Un patron courant consiste à approvisionner un grand nombre de cœurs en rafale, puis à les réduire jusqu'à zéro une fois l'application exécutée.
- L'application n'a pas besoin de rester active 24 h/24, 7 j/7. Toutefois, le système doit gérer les échecs de nœud et les arrêts d'application.
- Pour certaines applications, les tâches sont indépendantes et peuvent s'exécuter en parallèle. Dans d'autres cas, les tâches sont étroitement liées et doivent donc interagir ou échanger des résultats intermédiaires. Envisagez alors d'utiliser des technologies de mise en réseau à haute vitesse comme InfiniBand et un accès direct à la mémoire à distance (RDMA).
- Selon votre scénario d'usage, vous pouvez utiliser des tailles de machine virtuelle nécessitant beaucoup de ressources (H16r, H16mr et A9).

Quand utiliser cette architecture

- Opérations nécessitant beaucoup de ressources système, comme la simulation et l'analyse de chiffres.
- Simulations nécessitant beaucoup de ressources système et devant être réparties entre plusieurs UC de plusieurs ordinateurs (d'une dizaine à plusieurs milliers).
- Simulations qui exigent trop de mémoire pour un seul ordinateur et qui doivent être réparties entre plusieurs ordinateurs.
- Calculs de longue durée dont la réalisation prendrait trop de temps sur un seul ordinateur.
- Calculs de moindre envergure qui doivent être exécutés plusieurs centaines ou plusieurs milliers de fois, comme les simulations de Monte Carlo.

Avantages

- Haute performance avec un traitement « embarrassingly parallel ».
- Possibilité de tirer parti de centaines ou milliers de cœurs d'ordinateur pour résoudre plus rapidement de gros problèmes.
- Accès au matériel spécialisé haute performance, avec des réseaux à haut débit InfiniBand dédiés.
- Vous pouvez approvisionner des machines virtuelles selon le travail à effectuer, puis les retirer.

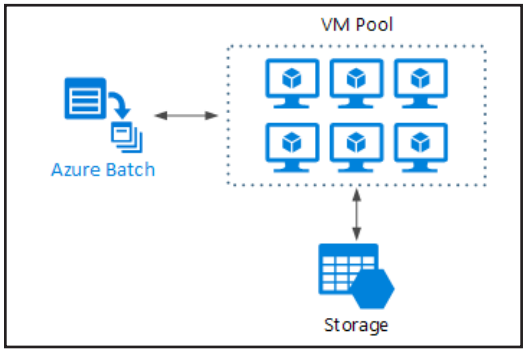
Problématique

- Gestion de l'infrastructure de machine virtuelle.
- Gestion du volume d'analyses de chiffres.
- Approvisionnement de milliers de cœurs dans des délais raisonnables.
- Pour les tâches étroitement liées, l'ajout de cœurs peut entraîner des baisses de rendement. Vous devrez peut-être effectuer quelques essais pour déterminer le nombre de cœurs optimal.

Big Compute avec Azure Batch

Azure Batch est un service géré pour l'exécution d'applications de calcul haute performance (HPC, High Performance Computing) à grande échelle.

Avec Azure Batch, vous configurez un pool de machines virtuelles et chargez les applications et fichiers de données. Ensuite, le service Batch approvisionne les machines virtuelles, attribue des tâches aux machines virtuelles, exécute les tâches et surveille la progression. Batch peut faire monter en charge les machines virtuelles automatiquement pour répondre aux besoins du scénario d'usage. Batch offre également des fonctionnalités de planification de travaux.



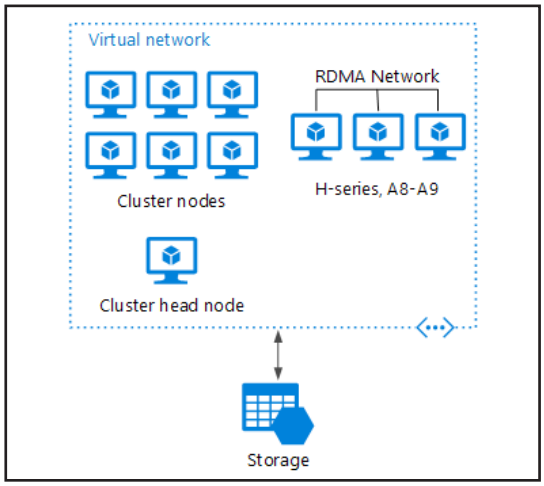
Big Compute exécuté sur des machines virtuelles

Vous pouvez utiliser Microsoft HPC Pack pour gérer un cluster de machines virtuelles et pour planifier et surveiller des travaux HPC. Avec cette approche, vous devez approvisionner et gérer les machines virtuelles et l'infrastructure réseau. Envisagez cette approche si vous avez des scénarios d'usage HPC existants et que vous souhaitez en déplacer certains (voire la totalité) vers Azure. Vous pouvez déplacer l'ensemble du cluster HPC vers Azure ou garder votre cluster HPC en local mais utiliser Azure pour la capacité de rafale. Pour plus d'informations, consultez Batch and HPC solutions for large-scale computing workloads (Solutions Batch et HPC pour les scénarios d'usage impliquant des calculs à grande échelle).

HPC Pack déployé sur Azure

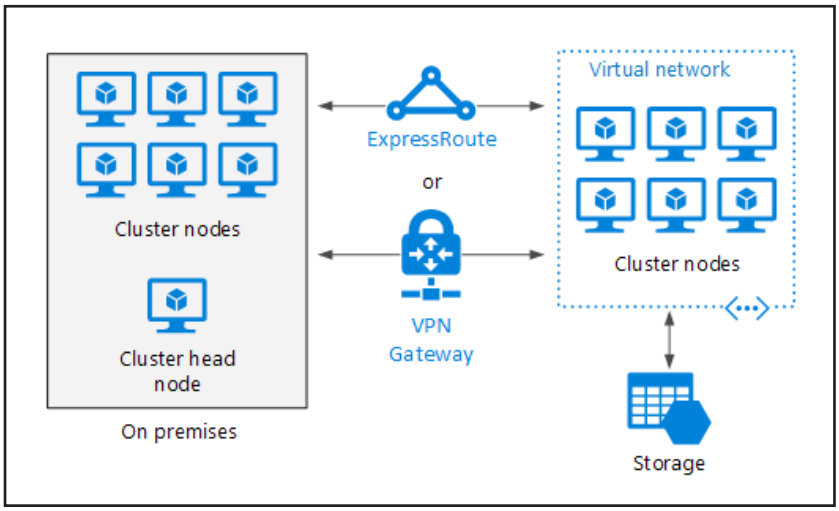
Dans ce scénario, le cluster HPC est créé entièrement dans Azure.

Le nœud principal apporte les services de gestion et de planification des travaux au cluster. Pour les tâches étroitement liées, utilisez un réseau RDMA assurant une communication à très haut débit et à faible latence entre les machines virtuelles. Pour plus d'informations, consultez Deploy an HPC Pack 2016 cluster in Azure (Déployer un cluster HPC Pack 2016 dans Azure).



Éclatement d'un cluster HPC sur Azure

Dans ce scénario, une organisation exécute HPC Pack en local et utilise des machines virtuelles Azure pour la capacité de rafale. Le nœud principal du cluster est local. Le réseau local est connecté au réseau virtuel Azure par le biais d'ExpressRoute ou d'une passerelle VPN.



Choisir les technologies de calcul et de banque de données

Choisissez les technologies appropriées aux applications Azure.

Lorsque vous concevez une solution pour Azure, vous devez établir deux choix technologiques de façon précoce dans le processus de conception. En effet, ces deux choix affectent l'ensemble de l'architecture. Ces choix portent sur les technologies de calcul et de banque de données.

Le modèle d'hébergement que vous choisissez pour les ressources informatiques sur lesquelles s'exécute votre application constitue votre option de calcul. D'une manière générale, vous pouvez choisir IaaS (Infrastructure-as-a-Service), PaaS (Platform-as-a-Service) et FaaS (Functions-as-a-Service). Le choix peut également porter sur le spectre de services sous-jacent. Actuellement, Azure offre sept options de calcul principales. Pour faire votre choix, examinez les fonctions appropriées et limites du service, la disponibilité et l'évolutivité, le coût et les considérations pour DevOps. Les tableaux de comparaison de cette section vous aideront à affiner vos choix.

La banque de données inclut tous les types de données que votre application doit gérer, ingérer ou générer ou que les utilisateurs créent. Les données métiers, les caches, les données IoT, la télémétrie et les données de journal non structurées représentent les types les plus courants. Les applications contiennent souvent plus d'un type de données. Les différents types de données impliquent des exigences de traitement différentes. Vous devez donc choisir le magasin approprié pour chaque type afin d'obtenir les meilleurs résultats. Certaines technologies de banque de données prennent en charge plusieurs modèles de stockage. Utilisez les informations de cette section pour choisir en premier lieu le modèle de stockage le mieux adapté à vos besoins. Ensuite, envisagez une banque de données spécifique au sein de cette catégorie, en fonction de facteurs tels que l'ensemble de fonctionnalités, le coût et la simplicité de gestion.

Cette section du Guide d'architecture d'applications contient les rubriques suivantes :

- Présentation des options de calcul : inclut des considérations générales sur le choix d'un service de calcul dans Azure.
- Critères de choix de l'option de calcul : comparaison de services de calcul Azure spécifiques selon différents axes comme le modèle d'hébergement, DevOps, la disponibilité et l'évolutivité.

- Choix de la banque de données appropriée : décrit les principales catégories de technologies de banque de données (SGBDR, magasin clé-valeur, base de données de documents, base de données de graphiques, etc.).
- Critères de comparaison pour le choix de la banque de données : décrit certains facteurs à prendre en compte lors du choix d'une banque de données.

Pour plus d'informations sur ces options de calcul, rendez-vous sur : https://docs.microsoft.com/en-us/azure/#pivot=services.

Présentation des options de calcul

Le terme *calcul* fait référence au modèle d'hébergement des ressources informatiques sur lesquelles s'exécute votre application.

L'IaaS (**Intrastructure-as-a-Service**) se place à une extrémité du spectre. Avec l'IaaS, vous approvisionnez les machines virtuelles dont vous avez besoin, ainsi que le réseau et les composants de stockage associés. Ensuite, vous déployez les logiciels et applications souhaités sur ces machines virtuelles. Ce modèle est le plus proche d'un environnement local traditionnel. C'est cependant Microsoft qui gère l'infrastructure. Vous continuez de gérer les machines virtuelles individuelles.

Un service PaaS (**Platform-as-a-Service**) fournit un environnement d'hébergement géré, dans lequel vous pouvez déployer votre application sans avoir à gérer des machines virtuelles ou des ressources réseau. Par exemple, au lieu de créer des machines virtuelles individuelles, vous spécifiez un nombre d'instances, et le service approvisionne, configure et gère les ressources nécessaires. Azure App Service est un exemple de service PaaS.

Il existe tout un spectre entre l'IaaS et le service PaaS pur. Par exemple, les machines virtuelles Azure peuvent faire l'objet d'une mise à l'échelle automatique grâce aux groupes de machines virtuelles identiques. Cette capacité de mise à l'échelle automatique n'est pas strictement limitée au service PaaS. Il s'agit cependant du type de fonction de gestion que peut offrir ce service.

Le service FaaS (**Fonctions-as-a-Service**) élimine lui aussi les préoccupations liées à l'environnement d'hébergement et va même encore plus loin. Au lieu de créer des instances de calcul et de déployer du code sur ces instances, vous déployez simplement votre code et le service l'exécute automatiquement. Vous n'avez pas besoin de gérer les ressources de calcul. Ces services exploitent une architecture sans serveur et assurent en toute transparence une montée ou descente en puissance selon le niveau nécessaire pour traiter le trafic. Azure Functions est un service FaaS.

L'IaaS offre les plus hauts niveaux de contrôle, de souplesse et de portabilité. Outre la simplicité et la mise à l'échelle élastique, le service FaaS peut offrir des réductions de coûts. En effet, vous ne payez que pour la durée d'exécution de votre code.

Le service PaaS se situe entre les deux. En général, plus la flexibilité offerte par un service est élevée, plus la configuration et la gestion des ressources relèvent de votre responsabilité. Les services FaaS gèrent automatiquement presque tous les aspects de l'exécution d'une application, tandis que les solutions IaaS vous obligent à approvisionner, configurer et gérer les machines virtuelles et composants réseau que vous créez.

Voici les principales options de calcul actuellement disponibles dans Azure :

- Les machines virtuelles représentent un service IaaS vous permettant de déployer et gérer des machines virtuelles dans un réseau virtuel.

- App Service est un service géré permettant d'héberger des applications web, des back-ends d'application mobile, des API RESTful ou des processus d'entreprise automatisés.
- Service Fabric est une plateforme de systèmes distribués qui peut s'exécuter dans de nombreux environnements, y compris Azure, ou en local. Service Fabric orchestre des microservices sur un cluster de machines.
- Azure Container Service vous permet de créer, configurer et gérer un cluster de machines virtuelles préconfigurées pour exécuter des applications en conteneur.
- Azure Functions est un service FaaS géré.
- Azure Batch est un service géré permettant l'exécution d'applications de calcul haute performance (HPC, High Performance Computing) parallèles à grande échelle.
- Les Services cloud sont un service géré pour l'exécution d'applications cloud. Ils utilisent un modèle d'hébergement PaaS.

Lorsque vous choisissez une option de calcul, vous devez tenir compte de certains facteurs :

- Modèle d'hébergement. Comment le service est-il hébergé ? Quelles sont les exigences et restrictions imposées par cet environnement d'hébergement ?
- DevOps. Existe-t-il une prise en charge intégrée pour les mises à niveau d'application ? Quel est le modèle de déploiement ?
- Évolutivité. Comment le service gère-t-il l'ajout et la suppression d'instances ? Peut-il assurer une mise à l'échelle automatique selon la charge et d'autres mesures ?
- Disponibilité. Quel est le contrat SLA associé au service ?
- Coût. En plus du coût du service lui-même, tenez compte du coût opérationnel qu'impliquera la gestion d'une solution basée sur ce service. Par exemple, les solutions IaaS peuvent présenter un coût opérationnel plus élevé.
- Quelles sont les limites globales de chaque service ?
- Quels sont les types d'architectures d'application appropriés pour ce service ?

Comparaison des services de calcul

Le terme calcul désigne le modèle d'hébergement des ressources informatiques sur lesquelles s'exécutent vos applications. Les tableaux suivants comparent les services de calcul Azure selon plusieurs axes. Reportez-vous à ces tableaux pour choisir une option de calcul pour votre application.

Modèle d'hébergement

Critères	Machines virtuelles	App Service	Service Fabric	Azure Functions	Azure Container Services	Services cloud	Azure Batch
Composition de l'application	Agnostique	Applications	Services, exécutables invités	Fonctions	Conteneurs	Rôles	Travaux planifiés
Densité	Agnostique	Plusieurs applications par instance via des plans d'application	Plusieurs services par machine virtuelle	Aucune instance dédiée	Plusieurs conteneurs par machine virtuelle	Une instance de rôle par machine virtuelle	Plusieurs conteneurs par machine virtuelle
Nombre minimal de nœuds	1 [2]	1	5 [3]	Aucun nœud dédié [1]	3	2	1 [4]
Gestion d'état	Sans état ou avec état	Sans état	Sans état ou avec état	Sans état	Sans état ou avec état	Sans état	Sans état
Hébergement web	Agnostique	Intégré	Auto-hébergement, IIS en conteneurs	N/A	Agnostique	Intégré (IIS)	Non
Système d'exploitation	Windows, Linux	Windows, Linux (préversion)	Windows, Linux (préversion)	N/A	Windows, Linux	Windows	Windows, Linux
Possibilité de déploiement sur un réseau virtuel dédié ?	Pris en charge	Pris en charge	Pris en charge	Non pris en charge	Pris en charge	Pris en charge [6]	Pris en charge
Connectivité hybride	Prise en charge	Prise en charge	Prise en charge	Non prise en charge	Prise en charge	Prise en charge [8]	Prise en charge

Remarques :

1. Si vous utilisez le plan App Service, les fonctions sont exécutées sur les machines virtuelles allouées à votre plan. Pour plus d'informations, rendez-vous sur https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.
2. Contrat SLA supérieur avec deux instances ou plus.
3. Pour les environnements de production.
4. Possibilité de descente en puissance jusqu'à zéro après l'exécution du travail.
5. Nécessite App Service Environment (ASE).
6. Réseau virtuel classique uniquement.
7. Nécessite ASE ou les Connexions hybrides BizTalk.
8. Réseau virtuel classique ou réseau virtuel Resource Manager via VNet Peering.

DevOps

Critères	Machines virtuelles	App Service	Service Fabric	Azure Functions	Azure Container Services	Services cloud	Azure Batch
Débogage local	Agnostique	IIS Express, autres	Cluster à nœud local	CLI Azure Functions	Runtime du conteneur local	Émulateur local	Non pris en charge
Modèle de programmation	Agnostique	Application web, tâches web pour les tâches en arrière-plan	Exécutable invité, modèle de service, modèle d'acteur, conteneurs	Fonctions avec déclencheurs	Agnostique	Rôle web, rôle de travail	Application en ligne de commande
Gestionnaire des ressources	Pris en charge	Pris en charge	Pris en charge	Pris en charge	Pris en charge	Limité [2]	Pris en charge
Mise à jour d'application	Pas de prise en charge intégrée	Emplacements de déploiement	Mise à niveau propagée (par service)	Pas de prise en charge intégrée	Selon l'orchestrateur.	Échange d'adresse IP virtuelle ou mise à jour propagée	N/A

Remarques :

1. Les options incluent IIS Express pour ASP.NET ou node.js (iisnode) ; serveur web PHP ; kit de ressources Azure pour IntelliJ, kit de ressources Azure pour Eclipse. App Service prend également en charge le débogage à distance de l'application web déployée.

2. Pour plus d'informations, rendez-vous sur https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-manager-supported-services.

Évolutivité

Critères	Machines virtuelles	App Service	Service Fabric	Azure Functions	Azure Container Services	Services cloud	Azure Batch
Mise à l'échelle automatique	Groupes de machines virtuelles identiques	Service intégré	Groupes de machines virtuelles identiques	Service intégré	Non pris en charge	Service intégré	N/A
Équilibreur de charge	Azure Load Balancer	Intégré	Azure Load Balancer	Intégré	Azure Load Balancer	Intégré	Azure Load Balancer
Limite d'échelle	Image de plateforme : 1 000 nœuds par groupe de machines virtuelles identiques, image personnalisée : 100 nœuds par groupe de machines virtuelles identiques	20 instances, 50 avec App Service Environment	100 nœuds par groupe de machines virtuelles identiques	Infinie [1]	100	Aucune limite définie (recommandation : 200 maximum)	Limite de 20 cœurs par défaut. Contactez le service client pour augmenter cette limite.

Remarques :

1. Pour plus d'informations, rendez-vous sur https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.

Disponibilité

Critères	Machines virtuelles	App Service	Service Fabric	Azure Functions	Azure Container Services	Services cloud	Azure Batch
Contrat SLA	Contrat SLA pour machines virtuelles	Contrat SLA pour App Service	Contrat SLA pour Service Fabric	Contrat SLA pour Functions	Contrat SLA pour Azure Container Service	Contrat SLA pour les Services cloud	Contrat SLA pour Azure Batch
Basculement multirégion	Traffic Manager	Traffic Manager	Traffic Manager, cluster multirégion	Non pris en charge	Traffic Manager	Traffic Manager	Non pris en charge

Remarques :

1. Pour plus d'informations sur les contrats SLA spécifiques, rendez-vous sur https://azure.microsoft.com/en-us/support/legal/sla/.

Sécurité

Critères	Machines virtuelles	App Service	Service Fabric	Azure Functions	Azure Container Services	Services cloud	Azure Batch
SSL	Configuré dans la machine virtuelle	Pris en charge	Pris en charge	Pris en charge	Configuré dans la machine virtuelle	Pris en charge	Pris en charge
RBAC	Pris en charge	Pris en charge	Pris en charge	Pris en charge	Pris en charge	Non pris en charge	Pris en charge

Autre

Critères	Machines virtuelles	App Service	Service Fabric	Azure Functions	Azure Container Services	Services cloud	Azure Batch
Coût	Windows, Linux	Tarification App service	Tarification Service Fabric	Tarification Azure Functions	Tarification Azure Container Service	Tarification Services cloud	Pris en charge
Styles d'architecture adaptés	Pris en charge	Pris en charge	Pris en charge	Pris en charge	Microservices, EDA	Traitement de file d'attente web	Big Compute

Remarques :

1. Pour obtenir des informations sur le coût spécifique, rendez-vous sur https://azure.microsoft.com/pricing/details/.

Présentation des banques de données

Choisissez la banque de données appropriée.

Les systèmes d'entreprise modernes gèrent des volumes de données croissants. Les données peuvent être ingérées à partir de services externes, générées par le système lui-même ou créées par les utilisateurs. Ces jeux de données peuvent présenter des caractéristiques et exigences de traitement extrêmement variées. Les entreprises utilisent les données à de nombreuses fins, par exemple pour évaluer les tendances, déclencher des processus d'entreprise, auditer leurs activités ou analyser le comportement des clients.

Face à cette hétérogénéité, une banque de données unique ne représente généralement pas la meilleure approche. En fait, il est souvent préférable de stocker différents types de données dans différentes banques de données, chacune axée sur un patron d'utilisation ou scénario d'usage spécifique. Le terme « persistance polyglotte » est utilisé pour décrire les solutions qui utilisent une combinaison de technologies de banque de données.

Le choix de la banque de données répondant à vos exigences représente une décision clé en matière de conception. Les bases de données SQL et NoSQL offrent des centaines d'implémentations possibles. Les banques de données sont souvent classées selon la façon dont elles structurent les données et les types d'opérations qu'elles prennent en charge. Cet article décrit plusieurs modèles de stockage parmi les plus courants. Notez que certaines technologies de banque de données peuvent prendre en charge plusieurs modèles de stockage. Par exemple, un système de gestion de base de données relationnelle (SGBDR) peut aussi prendre en charge le stockage clé/valeur ou de graphiques. En fait, on observe une tendance générale vers ce que l'on appelle la prise en charge multimodèle : un système de base de données unique prend en charge plusieurs modèles. Cependant, il reste utile d'appréhender les différents modèles de façon précise.

Toutes les banques de données d'une même catégorie ne fournissent pas le même ensemble de fonctionnalités. La plupart des banques de données fournissent des fonctionnalités côté serveur pour interroger et traiter les données. Parfois, cette fonctionnalité est intégrée au moteur de stockage de données. Dans d'autres cas, les capacités de stockage et de traitement de données sont séparées, et plusieurs options de traitement et d'analyse peuvent être disponibles. Les banques de données prennent également en charge différentes interfaces de programmation et de gestion.

En règle générale, vous devez commencer par déterminer le modèle de stockage le mieux adapté à vos exigences. Ensuite, envisagez une banque de données spécifique au sein de cette catégorie, en fonction de facteurs tels que l'ensemble de fonctionnalités, le coût et la simplicité de gestion.

Systèmes de gestion de base de données relationnelle

Les bases de données relationnelles organisent les données sous la forme d'une série de tables à deux dimensions contenant des lignes et des colonnes. Chaque table possède ses propres colonnes, et toutes les lignes d'une table présentent le même ensemble de colonnes. Il s'agit d'un modèle mathématique, et la plupart des fournisseurs proposent un dialecte de SQL (Structured Query Language) pour la récupération et la gestion des données. En général, un SGBDR implémente un mécanisme cohérent au niveau transactionnel, conforme au modèle ACID (atomicité, cohérence, isolation, durabilité) pour la mise à jour des informations.

En général, un SGBDR prend en charge un modèle « schema-on-write » : la structure de données est définie à l'avance, et toutes les opérations de lecture ou d'écriture doivent utiliser le schéma. Cette approche diffère de celle associée à la plupart des banques de données NoSQL, en particulier de type clé/valeur, pour lesquels le modèle « schema-on-read » suppose que le client appliquera son propre schéma d'interprétation aux données provenant de la base de données, sans dépendance au format des données en cours d'écriture.

Un SGBDR s'avère très utile lorsque des garanties de cohérence forte sont essentielles (toutes les modifications sont atomiques, et les données conservent un état cohérent après les transactions). Toutefois, les structures sous-jacentes ne se prêtent pas à une montée en charge par distribution du stockage et du traitement entre les machines. En outre, les informations stockées dans un SGBDR doivent être organisées selon une structure relationnelle par le biais du processus de normalisation. Même si ce processus est bien compris, il peut entraîner des manques d'efficacité. En effet, il est nécessaire de désassembler les entités logiques en lignes dans des tables distinctes, puis de réassembler les données lors de l'exécution des requêtes.

Service Azure approprié :

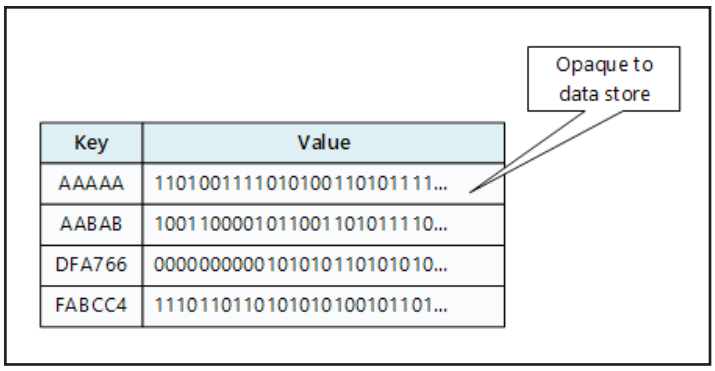
- Azure SQL Database. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/sql-database.
- Azure Database pour MySQL. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/mysql.
- Azure Database pour PostgreSQL. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/postgresql.

Magasins clé/valeur

Un magasin clé/valeur est essentiellement une table de hachage volumineuse. Vous associez chaque valeur de donnée à une clé unique, et le magasin clé/valeur utilise cette clé pour stocker les données en exploitant une fonction de hachage appropriée. La fonction de hachage est choisie pour assurer une répartition régulière des clés hachées à l'échelle du stockage des données.

La plupart des magasins clé/valeur prennent en charge uniquement les opérations d'interrogation, d'insertion et de suppression simples. Pour modifier une valeur (partiellement ou complètement), une application doit remplacer les données existantes par la valeur entière. Dans la plupart des implémentations, la lecture ou l'écriture d'une valeur unique représente une opération atomique. Si la valeur est élevée, l'écriture peut prendre un certain temps.

Une application peut stocker des données arbitraires en tant qu'ensemble de valeurs, bien que certains magasins clé/valeur imposent des limites de taille maximale pour les valeurs. Les valeurs stockées ne sont pas lisibles par les logiciels du système de stockage. Toutes les informations de schéma doivent être fournies et interprétées par l'application. Les valeurs sont essentiellement des blobs, et le magasin clé/valeur récupère ou stocke simplement la valeur par clé.



Les magasins clé/valeur sont hautement optimisés pour les applications effectuant des recherches simples, mais sont moins adaptés aux systèmes qui doivent interroger des données dans une diversité de magasins. Par ailleurs, les magasins clé/valeur ne sont pas optimisés pour les scénarios où l'interrogation par valeur est importante et où les recherches ne sont pas basées uniquement sur les clés. Par exemple, avec une base de données relationnelle, vous pouvez trouver un enregistrement à l'aide d'une clause WHERE. Les magasins clé/valeur ne bénéficient généralement pas de ce type de capacité de recherche pour les valeurs.

Un magasin clé/valeur unique peut être extrêmement évolutif. En effet, la banque de données peut facilement distribuer des données entre plusieurs nœuds sur des machines distinctes.

Services Azure appropriés :

- Cosmos DB. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/cosmos-db.
- Cache Redis Azure. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/cache.

Bases de données de documents

D'un point de vue conceptuel, une base de données de documents est semblable à un magasin clé/valeur. Cependant, elle stocke une collection de données et champs nommés (documents), pouvant être individuellement de simples éléments scalaires ou des éléments composés tels que des listes ou des collections enfants. Les données figurant dans les champs d'un document peuvent être encodées de différentes façons (XML, YAML, JSON, BSON, etc.) ou même stockées sous forme de texte brut. Contrairement aux magasins clé/valeur, les champs dans les documents sont exposés au système de gestion de stockage. Une application peut ainsi interroger et filtrer des données en utilisant les valeurs de ces champs.

En règle générale, un document contient l'ensemble des données d'une entité. Les éléments constituant une entité sont spécifiques à l'application. Par exemple, une entité peut contenir les détails d'un client, une commande ou une combinaison des deux. Un seul document peut contenir des informations qui seront réparties sur plusieurs tables relationnelles d'un SGBDR.

Un magasin de documents n'implique pas que tous les documents présentent la même structure. Cette approche de forme libre offre une grande flexibilité. Les applications peuvent stocker des données différentes dans les documents selon l'évolution des exigences de l'entreprise.

L'application peut récupérer des documents à l'aide de la clé de document. Il s'agit d'un identificateur unique du document, souvent haché, aidant à distribuer les données de façon régulière. Certaines bases de données de documents créent la clé de document automatiquement. D'autres vous permettent de spécifier un attribut du document à utiliser comme clé. L'application peut également interroger les documents selon la valeur d'un ou plusieurs champs. Certaines bases de données de documents prennent en charge l'indexation, qui favorise la recherche rapide de documents en fonction d'un ou plusieurs champs indexés.

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

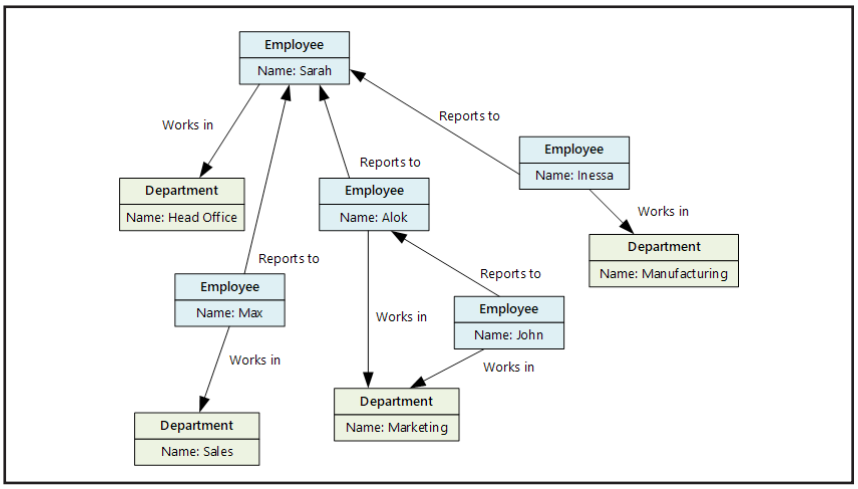
De nombreuses bases de données de documents prennent en charge les mises à jour sur place, ce qui permet à une application de modifier les valeurs de champs spécifiques d'un document sans réécrire l'ensemble du document. Les opérations de lecture et d'écriture sur plusieurs champs d'un document unique sont habituellement atomiques.

Service Azure approprié : Cosmos DB

Bases de données de graphiques

Une base de données de graphiques stocke deux types d'informations, les nœuds et les bords. Vous pouvez considérer les nœuds comme des entités. Les bords spécifient les relations entre les nœuds. Les nœuds et les bords peuvent posséder des propriétés fournissant des informations sur eux-mêmes, à l'instar des colonnes d'une table. Les bords peuvent également être associés à une direction indiquant la nature de la relation.

Une base de données de graphiques a pour objet de permettre à une application d'exécuter efficacement des requêtes qui parcourent le réseau de nœuds et de bords et d'analyser les relations entre les entités. Le schéma suivant illustre la base de données du personnel d'une organisation structurée sous forme de graphique. Les entités sont des collaborateurs et des services, et les bords indiquent les liens hiérarchiques, ainsi que le service dans lequel travaillent les collaborateurs. Dans ce graphique, les flèches sur les bords indiquent la direction des relations.



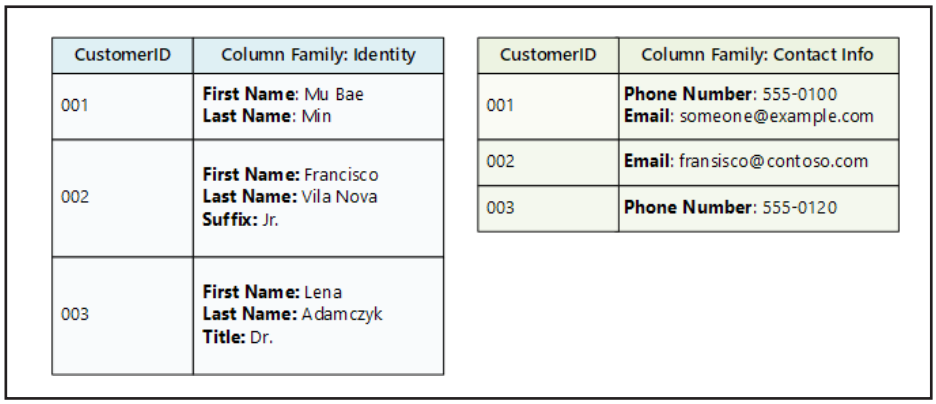
Cette structure simplifie l'exécution de requêtes comme « Trouver tous les collaborateurs sous la responsabilité directe ou indirecte de Sarah » ou « Qui travaille dans le même service que John ? ». Pour les grands graphiques contenant beaucoup d'entités et de relations, vous pouvez effectuer des analyses très complexes très rapidement. De nombreuses bases de données de graphiques offrent un langage de requête que vous pouvez utiliser pour parcourir efficacement un réseau de relations.

Service Azure approprié : Cosmos DB. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/cosmos-db

Bases de données avec familles de colonnes (column-family)

Une base de données avec familles de colonnes organise les données en lignes et colonnes. Dans sa forme la plus simple, elle peut être très similaire à une base de données relationnelle, au moins sur le plan conceptuel. Sa puissance réside principalement dans l'approche dénormalisée sur laquelle repose la structuration des données éparses.

Vous pouvez considérer ce type de base de données comme une base stockant des données tabulaires dans des lignes et des colonnes. Cependant, les colonnes sont divisées en groupes appelés familles de colonnes. Chaque famille de colonnes contient un jeu de colonnes logiquement liées et généralement récupérées ou manipulées en tant qu'unité. D'autres données accessibles séparément peuvent être stockées dans des famille de colonnes distinctes. Au sein d'une famille de colonnes, les nouvelles colonnes peuvent être ajoutées dynamiquement, et les lignes peuvent être éparses (c'est-à-dire qu'une ligne ne doit pas nécessairement comporter une valeur pour chaque colonne).



CustomerID	Column Family: Identity
001	**First Name**: Mu Bae **Last Name**: Min
002	**First Name:** Francisco **Last Name:** Vila Nova **Suffix:** Jr.
003	**First Name:** Lena **Last Name:** Adamczyk **Title:** Dr.

CustomerID	Column Family: Contact Info
001	**Phone Number**: 555-0100 **Email**: someone@example.com
002	**Email**: fransisco@contoso.com
003	**Phone Number**: 555-0120

Le schéma suivant offre un exemple avec deux familles de colonnes : *Identity* et *Contact Info*. Les données d'une même entité possèdent la même clé de ligne dans chaque famille de colonnes. Cette structure, où les lignes de tout objet donné d'une famille de colonnes peuvent varier de façon dynamique, représente un avantage majeur de l'approche par famille de colonnes. Cette forme de banque de données est ainsi hautement adaptée au stockage de données volatiles structurées. Contrairement à un magasin clé/valeur ou une base de données de documents, la plupart des bases de données avec famille de colonnes stockent les données dans l'ordre des clés et non en calculant un hachage. De nombreuses implémentations permettent de créer des index sur des colonnes spécifiques dans une famille de colonnes. Les index vous permettent de récupérer des données par valeur de colonne plutôt que par clé de ligne.

Les opérations de lecture et d'écriture pour une ligne sont habituellement atomiques et portent sur une seule famille de colonnes. Certaines implémentations fournissent une atomicité sur toute la ligne et portent sur plusieurs familles de colonnes.

Service Azure approprié : HBase dans HDInsight. Pour plus d'informations, rendez-vous sur https://azure.microsoft.com/services/cosmos-db

Analyse des données

Les magasins basés sur l'analyse de données offrent des solutions massivement parallèles pour l'ingestion, le stockage et l'analyse de données. Ces données sont réparties sur plusieurs serveurs par le biais d'une architecture sans partage (ou « shared-nothing architecture ») pour maximiser l'évolutivité et minimiser les dépendances. Comme les données sont peu susceptibles d'être statiques, ces magasins doivent être capables de gérer de grandes quantités d'informations provenant de plusieurs flux sous une variété de formats, tout en continuant à traiter les nouvelles requêtes.

Services Azure appropriés :

- SQL Data Warehouse
- Azure Data Lake

Bases de données de moteur de recherche

Une base de données de moteur de recherche permet de rechercher des informations contenues dans des services et banques de données externes. Elle peut être utilisée pour indexer des volumes de données massifs et fournir un accès en quasi-temps réel à ces index. Même si ces bases de données sont généralement considérées comme s'apparentant au web, de nombreux systèmes à grande échelle les utilisent pour offrir des capacités de recherche structurée et ad hoc en plus de leurs propres bases de données.

La capacité à stocker et indexer les informations très rapidement et le temps de réponse rapide aux demandes de recherche représentent les principales caractéristiques d'une base de données de moteur de recherche. Les index peuvent être multidimensionnels et peuvent prendre en charge des recherches en texte libre sur de grands volumes de données textuelles. L'indexation peut être effectuée à l'aide d'un modèle « pull », déclenché par la base de données de moteur de recherche, ou avec un patron « push », initié par le code de l'application externe.
La recherche peut être exacte ou approximative. Une recherche approximative trouve les documents qui correspondent à un ensemble de termes et calcule le niveau de correspondance. Certains moteurs de recherche prennent également en charge l'analyse linguistique, qui peut retourner des correspondances basées sur des synonymes, des extensions de genre (par exemple, la mise en correspondance des chiens avec les animaux de compagnie) et la recherche de radical (correspondances avec les mots de même racine).

Service Azure approprié : Recherche Azure

Bases de données de séries chronologiques

Les données de séries chronologiques constituent un ensemble de valeurs organisées de façon chronologique. Une base de données de séries chronologiques est optimisée pour ce type de données. Les bases de données de séries chronologiques doivent prendre en charge un très grand nombre d'écritures car elles recueillent généralement de grandes quantités de données provenant de nombreuses sources, et ce, en temps réel. Les mises à jour sont rares, et les suppressions sont souvent réalisées en bloc. Les enregistrements écrits dans une base de données de séries chronologiques sont généralement de petite taille. Cependant, ces enregistrements sont souvent très nombreux, et la taille totale des données peut augmenter rapidement.

Les bases de données de séries chronologiques sont adaptées au stockage de données de télémétrie. Les scénarios incluent les capteurs IoT et les compteurs d'application/système.

Service Azure approprié : Time Series Insights

Stockage d'objets

Le stockage d'objets est optimisé pour le stockage et la récupération d'objets binaires volumineux (images, fichiers, flux audio et vidéo, documents et objets de données d'applications volumineuses, images de disque de machine virtuelle). Les objets de ces types de magasin sont composés des données stockées, de certaines métadonnées et d'un ID unique pour accéder à l'objet. Les magasins d'objets permettent de gérer de très grandes quantités de données non structurées.

Service Azure approprié : Stockage Blob

Fichiers partagés

Parfois, l'utilisation de simples fichiers plats représente le moyen le plus efficace de stocker et récupérer des informations. Les partages de fichiers permettent d'accéder aux fichiers sur un réseau. Moyennant des mécanismes appropriés de sécurité et de contrôle d'accès simultanés, ce type de partage de données peut permettre aux services distribués de fournir un accès aux données hautement évolutif pour la réalisation d'opérations de base comme de simples requêtes de lecture et d'écriture.

Service Azure approprié : Stockage Fichier

Comparaison des banques de données

Critères de choix d'une banque de données

Azure prend en charge de nombreux types de solutions de stockage de données, chacune présentant des fonctionnalités et des caractéristiques différentes. Cet article décrit les critères de comparaison dont vous devez tenir compte lors de l'évaluation d'une banque de données. L'objectif est de vous aider à déterminer les types de stockage de données répondant aux exigences de votre solution.

Considérations générales

Pour commencer votre comparaison, rassemblez autant d'informations que possible sur vos besoins en données en vous basant sur les aspects suivants. Ces informations vous aideront à déterminer les types de stockage de données répondant à vos besoins.

Exigences fonctionnelles

- **Format de données.** Quel type de données envisagez-vous de stocker ? Les types courants incluent les données transactionnelles, les objets JSON, la télémétrie, les index de recherche et les fichiers plats.
- **Taille des données.** Quelle est la taille des entités que vous devez stocker ? Ces entités devront-elles être maintenues sous la forme d'un seul document, ou peuvent-elles fractionnées entre plusieurs documents, tables, collections, etc. ?
- **Échelle et structure.** De quelle capacité de stockage avez-vous besoin au total ? Envisagez-vous de partitionner vos données ?
- **Relations entre les données.** Vos données devront-elles prendre en charge les relations « un-à-plusieurs » ou « plusieurs-à-plusieurs » ? Les relations elles-mêmes représentent-elles un aspect clé des données ? Devrez-vous associer ou combiner des données provenant du même jeu de données ou de jeux de données externes ?
- **Modèle de cohérence.** Dans quelle mesure est-il important que les mises à jour apportées à un nœud apparaissent dans d'autres nœuds avant que d'autres modifications puissent être apportées ? Pouvez-vous accepter la cohérence éventuelle ? Avez-vous besoin de garanties ACID pour les transactions ?
- **Flexibilité du schéma.** Quel type de schéma allez-vous appliquer à vos données ? Allez-vous utiliser un schéma fixe, une approche « schema-on-write » ou une approche « schema-on-read » ?

- **Accès concurrentiel.** Quel type de mécanisme d'accès concurrentiel souhaitez-vous utiliser lors de la mise à jour et de la synchronisation des données ? L'application effectuera-t-elle de nombreuses mises à jour susceptibles d'entraîner des conflits ? Le cas échéant, vous aurez peut-être besoin d'un verrouillage des enregistrements et d'un contrôle d'accès concurrentiel pessimiste. Sinon, pouvez-vous accepter des contrôles d'accès concurrentiel optimistes ? Si c'est le cas, le simple contrôle d'accès concurrentiel basé sur l'horodatage est-il suffisant, ou avez-vous également besoin du contrôle d'accès concurrentiel multiversion ?
- **Déplacement de données.** Votre solution devra-t-elle effectuer des tâches d'ETL pour déplacer des données vers d'autres magasins ou entrepôts de données ?
- **Cycle de vie des données.** Les données reposent-elles sur un disque optique non réinscriptible ? Peuvent-elles être déplacées vers un stockage froid ?
- **Autres fonctions prises en charge.** Avez-vous besoin d'autres fonctions spécifiques comme la validation de schéma, l'agrégation, l'indexation, la recherche en texte intégral, MapReduce ou d'autres fonctions d'interrogation ?

Exigences non fonctionnelles

- **Performance et évolutivité.** Quelles sont vos exigences en termes de performance des données ? Avez-vous des exigences particulières concernant les taux d'ingestion et de traitement des données ? Quels sont les temps de réponse acceptables pour l'interrogation et l'agrégation des données après ingestion ? Dans quelle mesure la banque de données doit-elle évoluer ? Votre scénario d'usage repose-t-il principalement sur les lectures ou les écritures ?
- **Fiabilité.** Quel contrat SLA global devez-vous prendre en charge ? Quel niveau de tolérance aux pannes devez-vous offrir aux consommateurs des données ? De quel type de capacités de sauvegarde et restauration avez-vous besoin ?
- **Réplication.** Vos données devront-elles être distribuées entre plusieurs réplicas ou régions ? De quel type de capacités de réplication de données avez-vous besoin ?
- **Limites.** Les limites d'une banque de données spécifique s'accorderont-elles avec vos exigences en termes d'échelle, de nombre de connexions et de débit ?

Gestion et coût

- **Service géré.** Dans la mesure du possible, utilisez un service de données gérées, sauf si vous avez besoin de fonctionnalités spécifiques offertes uniquement par une banque de données hébergée par IaaS.
- **Disponibilité dans la région.** Pour les services gérés, le service est-il disponible dans toutes les régions Azure ? Votre solution doit-elle être hébergée dans certaines régions Azure ?
- **Portabilité.** Vos données devront-elles migrer vers des environnements locaux, des Datacenters externes ou d'autres environnements d'hébergement cloud ?
- **Gestion des licences.** Avez-vous une préférence concernant le type de licence (propriétaire ou licence OSS) ? Le type de licence possible est-il soumis à d'autres restrictions externes ?
- **Coût global.** Quel est le coût global de l'utilisation du service au sein de votre solution ? Vos exigences en termes de temps d'activité et de débit impliquent l'exécution de combien d'instances ? Tenez compte des coûts opérationnels dans ce calcul. Le faible coût opérationnel représente l'une des raisons incitant à privilégier les services gérés.
- **Rentabilité.** Pouvez-vous partitionner vos données pour les stocker de façon plus rentable ? Par exemple, pouvez-vous déplacer des objets volumineux d'une base de données relationnelle onéreuse vers un magasin d'objets ?

Sécurité

- **Sécurité.** De quel type de chiffrement avez-vous besoin ? Avez-vous besoin du chiffrement au repos ? Quel mécanisme d'authentification souhaitez-vous utiliser pour vous connecter à vos données ?
- **Audits.** Quel type de journal d'audit devez-vous générer ?
- **Exigences réseau.** Avez-vous besoin de restreindre ou de gérer l'accès à vos données depuis d'autres ressources réseau ? Les données doivent-elles être accessibles uniquement depuis l'intérieur de l'environnement Azure ? Les données doivent-elles être accessibles à partir de sous-réseaux ou d'adresses IP spécifiques ? Doivent-elles être accessibles à partir des applications ou services hébergés localement ou dans d'autres Datacenters externes ?

DevOps

- **Compétences.** Votre équipe privilégie-t-elle l'utilisation de certains langages de programmation, systèmes d'exploitation ou autres technologies ? Votre équipe aurait-elle du mal à travailler avec certains autres langages de programmation, systèmes d'exploitation ou technologies ?
- **Clients.** La prise en charge des clients est-elle efficace avec vos langages de développement ?

Les sections suivantes comparent différents modèles de banque de données en termes de profil de scénario d'usage, de types de données et de cas d'utilisation.

Systèmes de gestion de base de données relationnelle (SGBDR)

Scénario d'usage

- Les créations d'enregistrements et mises à jour de données existantes sont fréquentes.
- Plusieurs opérations doivent être exécutées dans le cadre d'une transaction unique.
- Des fonctions d'agrégation sont nécessaires pour créer un tableau croisé.
- Une forte intégration aux outils de génération de rapports est nécessaire.
- Les relations sont appliquées à l'aide de contraintes de base de données.
- Des index sont utilisés pour optimiser les performances des requêtes.
- Un accès à des sous-ensembles de données spécifiques est autorisé.

Type de données

- Les données sont hautement normalisées.
- Des schémas de base de données sont requis et appliqués.
- Relations « plusieurs-à-plusieurs » entre des entités de données dans la base de données.
- Les contraintes sont définies dans le schéma et imposées à toutes les données de la base de données.
- Les données nécessitent une intégrité élevée. Les index et les relations doivent être tenus à jour avec précision.

- Les données nécessitent une cohérence forte. Les transactions fonctionnent de façon à garantir que toutes les données sont totalement cohérentes pour l'ensemble des utilisateurs et processus.
- Les entrées de données individuelles sont supposées être petites à moyennes.

Exemples

- Cœur de métier (gestion du capital humain, gestion de la relation client, gestion intégrée)
- Gestion des stocks
- Base de données de création de rapports
- Comptabilité
- Gestion des ressources
- Gestion de fonds
- Gestion des commandes

Bases de données de documents

Scénario d'usage

- Usage général.
- Les opérations d'insertion et de mise à jour sont courantes. Les créations d'enregistrements et mises à jour de données existantes sont fréquentes.
- Aucune discordance d'impédance relationnelle objet. Les documents peuvent mieux correspondre aux structures d'objet utilisées dans le code d'application.
- L'accès concurrentiel optimiste est plus couramment utilisé.
- Les données doivent être modifiées et traitées par l'application de consommation.
- Les données nécessitent un index sur plusieurs champs.
- Les documents individuels sont récupérés et écrits en tant que bloc unique.

Type de données

- Les données peuvent être gérées de manière dénormalisée.
- La taille des données de documents individuels est relativement réduite.
- Chaque type de document peut utiliser son propre schéma.
- Les documents peuvent inclure des champs facultatifs.
- Les données de document sont semi-structurées, ce qui signifie que les types de données de chaque champ ne sont pas strictement définis.
- L'agrégation de données est prise en charge.

Exemples

- Catalogue de produits
- Comptes d'utilisateur
- Nomenclatures
- Personnalisation
- Gestion de contenu
- Données opérationnelles
- Gestion des stocks
- Données de l'historique des transactions
- Vue matérialisée d'autres magasins NoSQL. Remplace l'indexation fichier/blob.

Magasins clé/valeur

Scénario d'usage

- Les données sont identifiées et accessibles à l'aide d'une clé ID unique, comme dans un dictionnaire.
- Évolutivité à grande échelle.
- Aucune jointure, aucun verrouillage et aucune union ne sont nécessaires.
- Aucun mécanisme d'agrégation n'est utilisé.
- Les index secondaires ne sont généralement pas utilisés.

Type de données

- La taille des données est généralement élevée.
- Chaque clé est associée à une valeur unique, qui est un blob de données non gérées.
- Aucun schéma n'est appliqué.
- Aucune relation entre les entités.

Exemples

- Mise en cache de données
- Gestion des sessions
- Gestion des préférences et profils des utilisateurs
- Recommandation de produit et serveur d'annonces
- Dictionnaires

Magasins clé/valeur

Scénario d'usage

- Les données sont identifiées et accessibles à l'aide d'une clé ID unique, comme dans un dictionnaire.
- Évolutivité à grande échelle.
- Aucune jointure, aucun verrouillage et aucune union ne sont nécessaires.
- Aucun mécanisme d'agrégation n'est utilisé.
- Les index secondaires ne sont généralement pas utilisés.

Type de données

- La taille des données est généralement élevée.
- Chaque clé est associée à une valeur unique, qui est un blob de données non gérées.
- Aucun schéma n'est appliqué.
- Aucune relation entre les entités.

Exemples

- Mise en cache de données
- Gestion des sessions
- Gestion des préférences et profils des utilisateurs
- Recommandation de produit et serveur d'annonces
- Dictionnaires

Bases de données de graphiques

Scénario d'usage

- Les relations entre les éléments de données sont très complexes. Elles impliquent de nombreux tronçons entre les éléments de données connexes.
- Les relations entre les éléments de données sont dynamiques et changent au fil du temps.
- Les relations entre les objets reposent sur des entités de première classe. Le parcours ne requiert ni clés étrangères ni jointures.

Type de données

- Les données sont composées de nœuds et de relations.
- Les nœuds sont semblables aux lignes de table ou à des documents JSON.
- Les relations sont tout aussi importantes que les nœuds et sont exposées directement dans le langage de requête.
- Les objets composites, comme une personne possédant plusieurs numéros de téléphone, ont tendance à être fractionnés en nœuds distincts, plus petits, combinés avec des relations pouvant être parcourues.

Exemples

- Organigrammes
- Graphes sociaux
- Détection des fraudes
- Analyse
- Moteurs de recommandation

Bases de données avec familles de colonnes (column-family)

Scénario d'usage

- La plupart bases de données avec familles de colonnes effectuent des opérations d'écriture extrêmement rapidement.
- Les opérations de mise à jour et de suppression sont rares.
- Conçues pour fournir un accès à haut débit et à faible latence.
- Favorise l'accès en interrogation à un ensemble de champs spécifique au sein d'un enregistrement beaucoup plus volumineux.
- Évolutivité à grande échelle.

Type de données

- Les données sont stockées dans les tables comprenant une colonne clé et une ou plusieurs familles de colonnes.
- Certaines colonnes peuvent varier par leurs lignes individuelles.
- Les cellules individuelles sont accessibles par l'intermédiaire des commandes get et put.
- Plusieurs lignes sont retournées à l'aide d'une commande scan.

Exemples

- Recommandations
- Personnalisation
- Données de capteurs
- Télémétrie
- Messagerie
- Analyse des médias sociaux
- Analyse web
- Pilotage des activités
- Météo et autres données de séries chronologiques

Bases de données de moteur de recherche

Scénario d'usage

- Indexation de données provenant de multiples sources et services.
- Les requêtes sont ad hoc et peuvent s'avérer complexes.
- Nécessite l'agrégation.
- La recherche en texte intégral est requise.
- La requête ad hoc en libre-service est requise.
- L'analyse de données avec index sur tous les champs est requise.

Type de données

- Semi-structurées ou non structurées
- Texte
- Texte avec référence aux données structurées

Exemples

- Catalogues de produits
- Recherche de site
- Journalisation
- Analyse
- Sites d'achat

Data warehouse

Scénario d'usage

- Analyse des données
- Aide à la décision d'entreprise

Type de données

- Données historiques de plusieurs sources.
- Généralement dénormalisées dans un schéma en « étoile » ou « flocon de neige » comprenant des tables de faits et de dimension.
- Nouvelles données généralement chargées sur une base régulière.
- Les tables de dimension comprennent souvent plusieurs versions historiques d'une entité, appelées dimensions à variation lente.

Exemples

- Un entrepôt de données d'entreprise qui fournit des données pour des tableaux de bord, rapports et modèles analytiques.

Bases de données de séries chronologiques

Scénario d'usage

- La quasi-totalité des opérations (95-99 %) sont des opérations d'écriture.
- Les enregistrements sont généralement ajoutés de façon séquentielle et chronologique.
- Les mises à jour sont rares.
- Les suppressions sont effectuées en bloc et concernent des blocs ou enregistrements contigus.
- Les requêtes de lecture peuvent être supérieures à la mémoire disponible.
- Il arrive fréquemment que plusieurs lectures aient lieu simultanément.
- Les données sont lues de façon séquentielle, par ordre chronologique croissant ou décroissant.

Type de données

- Un horodatage qui est utilisé comme clé primaire et mécanisme de tri.
- Mesures à partir de l'entrée ou des descriptions de ce que l'entrée représente.
- Balises qui définissent des informations supplémentaires sur l'entrée, notamment concernant le type et l'origine.

Exemples

- Surveillance et télémétrie des événements.
- Données de capteur ou autres données IoT.

Stockage d'objets

Scénario d'usage

- Identifié par clé.
- Les objets peuvent être accessibles publiquement ou en privé.
- Le contenu est généralement une ressource comme une feuille de calcul, une image ou un fichier vidéo.
- Le contenu doit être durable (persistant) et externe à toute couche Application ou machine virtuelle.

Type de données

- La taille des données est importante.
- Données de blob.
- La valeur est opaque.

Exemples

- Images, vidéos, documents Office, fichiers PDF
- CSS, scripts, CSV
- HTML statique, JSON
- Fichiers journaux et d'audit
- Sauvegardes de base de données

Fichiers partagés

Scénario d'usage

- Migration à partir d'applications existantes qui interagissent avec le système de fichiers.
- Requiert une interface SMB.

Type de données

- Fichiers dans un ensemble de dossiers hiérarchique.
- Accessible avec des bibliothèques d'E/S standard.

Exemples

- Fichiers hérités.
- Contenu partagé accessible parmi un certain nombre de machines virtuelles ou instances d'application.

Concevoir votre application Azure : principes de conception

Maintenant que vous avez choisi votre architecture et vos technologies de calcul et de banque de données, vous êtes prêt à commencer à concevoir et générer votre application cloud. Cette section et les deux suivantes fournissent des conseils et ressources pour concevoir une application pour le cloud de façon optimale.

Cette section décrit dix principes de conception à garder à l'esprit lors de la génération d'application. Ils vous aideront à générer une application plus évolutive, plus résiliente et plus facile à gérer.

1. **Conception en faveur de la réparation spontanée.** Un système distribué fait inévitablement l'objet de défaillances. Concevez votre application de sorte qu'elle puisse se réparer spontanément en cas de défaillance.

2. **Assurer la redondance de chaque élément.** Assurez la redondance de votre application pour éviter les points de défaillance uniques.

3. **Minimiser la coordination.** Minimisez la coordination entre les services d'application pour garantir l'évolutivité.

4. **Conception en faveur de la montée en charge.** Concevez votre application de sorte qu'elle bénéficie d'une évolutivité horizontale en ajoutant ou supprimant de nouvelles instances selon la demande.

5. **Partition pour contourner les limites.** Utilisez le partitionnement pour contourner les limites de calcul, de réseau et de base de données.

6. **Conception pour des opérations.** Concevez votre application de sorte que l'équipe des opérations dispose des outils dont elle a besoin.

7. **Utilisation de services gérés.** Dans la mesure du possible, utilisez un service PaaS (platform as a service) et non IaaS (infrastructure as a service).

8. **Utilisation du meilleur magasin de données pour la tâche.** Choisissez la technologie de stockage la mieux adaptée à vos données et à l'usage prévu.

9. **Conception pour l'évolution.** Toutes les applications réussies changent au fil du temps. La capacité à innover en continu repose sur une conception évolutive.

10. **Développement pour les besoins de l'entreprise.** Chaque décision de conception doit être justifiée par une exigence de l'entreprise.

Conception en faveur de la réparation spontanée

Concevez votre application de sorte qu'elle puisse se réparer spontanément en cas de défaillance

Un système distribué fait inévitablement l'objet de défaillances. Le matériel peut faire l'objet de défaillances. Le réseau peut présenter des défaillances passagères. L'ensemble d'un service ou d'une région peut-être confronté à une perturbation. Même si le cas est rare, il doit être prévu.

Par conséquent, vous devez concevoir votre application de sorte qu'elle puisse se réparer spontanément en cas de défaillance. Cela nécessite une approche en trois volets :

- Détection des défaillances.
- Réponse appropriée aux défaillances.
- Consignation et surveillance des défaillances pour bénéficier de renseignements opérationnels.

La façon dont vous répondez à un type de défaillance particulier peut dépendre des exigences de disponibilité de votre application. Par exemple, si vous avez besoin d'une disponibilité très élevée, vous pouvez basculer automatiquement vers une région secondaire lors d'une panne régionale. Cependant, ceci entraînera un coût plus élevé qu'un déploiement sur une seule région.

Par ailleurs, ne considérez pas uniquement les événements majeurs comme les pannes régionales, qui sont généralement rares. Concentrez-vous tout autant (si ce n'est plus) sur la gestion des défaillances locales et de courte durée comme les défaillances de connectivité réseau ou les échecs de connexion aux bases de données.

Recommandations

Recommencez les opérations ayant échoué. Pour plus d'informations, consultez Patron Retry (Nouvelle tentative) et rendez-vous sur https://docs.microsoft.com/en-us/azure/architecture/best-practices/transient-faults.

Protégez les services distants défaillants (disjoncteur). Il est judicieux de procéder à une nouvelle tentative après une défaillance passagère. Cependant, si la défaillance persiste, vous pouvez vous retrouver avec trop d'appelants martelant un service défaillant. Ceci peut entraîner des défaillances en cascade, à mesure que les demandes sont refoulées. Utilisez le patron Circuit Breaker (Disjoncteur) pour basculer rapidement (sans appel distant) lorsqu'une opération est susceptible d'échouer.

Isolez les ressources critiques (cloison). Les défaillances affectant un sous-système peuvent parfois se répercuter en cascade. Cela peut se produire si, à la suite d'une défaillance, certaines ressources (threads ou sockets, par exemple) ne sont pas libérées suffisamment rapidement, conduisant à l'épuisement des ressources. Pour éviter cela, partitionnez un système en groupes isolés, de sorte qu'une défaillance affectant une partition n'arrête pas l'ensemble du système.

Procédez au nivellement de charge. Les applications peuvent rencontrer de soudains pics de trafic qui peuvent submerger les services sur le principal. Pour éviter cela, utilisez le patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente) pour mettre en file d'attente les éléments de travail en vue d'une exécution asynchrone. La file d'attente agit comme un tampon qui atténue les pics de charge.

Assurez le basculement. Si une instance n'est pas joignable, basculez vers une autre instance. Pour les éléments sans état, comme un serveur web, placez plusieurs instances derrière un équilibreur de charge ou un gestionnaire de trafic. Pour les éléments qui stockent un état, comme une base de données, utilisez des réplicas et le basculement. Selon la banque de données et le mode de réplication, ceci peut contraindre l'application à gérer une cohérence éventuelle.

Compensez les transactions ayant échoué. D'une manière générale, évitez les transactions distribuées, car elles nécessitent une coordination entre les services et les ressources. Au lieu de cela, composez une opération à partir de transactions individuelles plus petites. Si l'opération échoue à mi-chemin, utilisez des transactions de compensation pour annuler toute étape déjà achevée.

Soumettez les transactions de longue durée à des points de contrôle. Les points de contrôle peuvent fournir une résilience en cas d'échec d'une opération de longue durée. Lorsque l'opération redémarre (si, par exemple, elle est reprise par une autre machine virtuelle), elle peut reprendre à partir du dernier point de contrôle.

Dégradez le service de façon appropriée. Parfois, lorsque vous ne pouvez pas contourner un problème, vous pouvez tout de même offrir des fonctionnalités limitées mais utiles. Imaginez une application affichant un catalogue de livres. Si l'application ne peut pas récupérer l'image miniature de la couverture, elle peut afficher une image d'espace réservé. L'ensemble d'un sous-système n'est pas nécessairement critique pour l'application. Par exemple, pour un site de commerce électronique, l'affichage des recommandations sur les produits est probablement moins critique que le traitement des commandes.

Limitez la bande passante des clients. Parfois, même un petit nombre d'utilisateurs peut créer une charge excessive, ce qui peut réduire la disponibilité de votre application pour les autres utilisateurs. Dans ce cas, limitez la bande passante du client durant une certaine période. Voir Patron Throttling (Limitation).

Bloquez les mauvais acteurs. Le fait que vous limitiez la bande passante d'un client ne signifie pas nécessairement qu'il s'agissait d'un client malveillant. Cela signifie simplement que le client a dépassé son quota de service. Cependant, lorsqu'un client dépasse constamment son quota ou adopte un comportement inapproprié, vous pouvez le bloquer. Définissez un traitement hors bande pour permettre à l'utilisateur de demander un déblocage.

Utilisez l'élection de leader. Lorsque vous devez coordonner une tâche, utilisez l'élection de leader pour sélectionner un coordinateur. De cette façon, le coordinateur n'est pas un point de défaillance unique. Si le coordinateur échoue, un autre coordinateur est sélectionné. Au lieu d'implémenter un algorithme d'élection de leader en partant de zéro, envisagez une solution disponible dans le commerce comme Zookeeper.

Effectuez des tests avec injection d'erreurs. Trop souvent, le chemin de la réussite est bien testé mais pas le chemin de l'échec. Un système peut être exécuté en production pendant une longue période avant qu'une voie de défaillance ne soit découverte. Utilisez l'injection d'erreurs pour tester la résilience du système aux défaillances, en déclenchant des défaillances réelles ou simulées.

Acceptez l'ingénierie du chaos. L'ingénierie du chaos étend la notion d'injection d'erreurs en injectant de façon aléatoire des défaillances ou des conditions anormales dans des instances de production.

Pour connaître une approche structurée permettant à vos applications de se réparer spontanément, consultez Conception d'applications résilientes pour Azure.

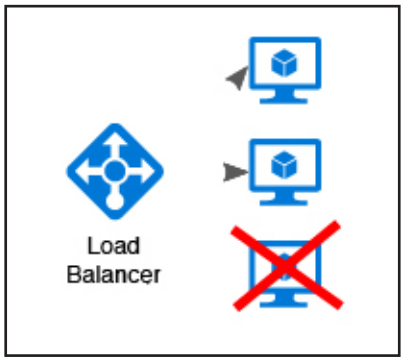
Assurer la redondance de chaque élément

Assurez la redondance de votre application pour éviter les points de défaillance uniques

Une application résiliente contourne les défaillances. Identifiez les chemins critiques dans votre application. Une redondance est-elle prévue à chaque point du chemin ? Si un sous-système échoue, l'application basculera-t-elle ?

Recommandations

Examinez les besoins de l'entreprise. Le niveau de redondance intégrée à un système peut affecter le coût et la complexité. Votre architecture doit être basée sur les exigences de votre entreprise, comme l'objectif de délai de récupération. Par exemple, un déploiement multirégion est plus onéreux et plus compliqué à gérer qu'un déploiement sur une seule région. Vous aurez besoin de procédures opérationnelles pour gérer le basculement et la restauration automatique. Le coût et la complexité supplémentaires peuvent se justifier dans certains scénarios professionnels, mais pas de façon systématique.



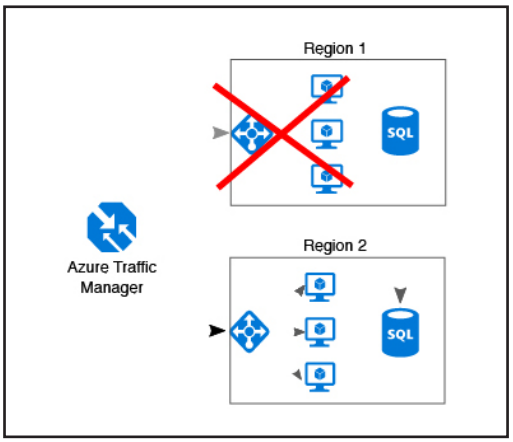
Placez les machines virtuelles derrière un équilibreur de charge. N'utilisez pas une seule machine virtuelle pour les scénarios d'usage stratégiques. Il est préférable de placer plusieurs machines virtuelles derrière un équilibreur de charge. Si une machine virtuelle n'est plus disponible, l'équilibreur de charge répartit le trafic sur les machines virtuelles saines restantes. Pour savoir comment déployer cette configuration, consultez Multiple VMs for scalability and availability (Plusieurs machines virtuelles pour l'évolutivité et la disponibilité).

Répliquez les bases de données. Azure SQL Database et Cosmos DB répliquent automatiquement les données au sein d'une région. Vous pouvez activer la géoréplication entre les régions. Si vous utilisez une solution de base de données IaaS, choisissez-en une qui prend en charge la réplication et le basculement, par exemple les groupes de disponibilité AlwaysOn SQL Server. Pour plus d'informations, rendez-vous sur https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/always-on-availability-groups-sql-server.

Activez la géoréplication. La géoréplication pour Azure SQL Database et Cosmos DB crée des réplicas secondaires lisibles de vos données dans une ou plusieurs régions secondaires. En cas de panne, la base de données peut basculer vers la région secondaire pour les écritures. Pour plus d'informations sur Azure SQL Database, rendez-vous sur https://docs.microsoft.com/en-us/azure/sql-database/sql-database-geo-replication-overview. Pour plus d'informations sur Cosmos DB, rendez-vous sur https://docs.microsoft.com/en-us/azure/documentdb/documentdb-distribute-data-globally.

Partition pour la disponibilité. Le partitionnement de base de données est souvent utilisé pour améliorer l'évolutivité. Cependant, il peut également améliorer la disponibilité. Lorsqu'une partition tombe en panne, les autres partitions restent accessibles. Une défaillance affectant une partition perturbera uniquement un sous-ensemble des transactions.

Effectuez le déploiement sur plusieurs régions. Pour une disponibilité maximale, déployez l'application sur plusieurs régions. De cette façon, dans le cas improbable où un problème affecterait l'ensemble d'une région, l'application pourra basculer vers une autre région. Le schéma suivant montre une application multirégion qui utilise Azure Traffic Manager pour gérer le basculement.



Synchronisez le basculement du frontal et du principal. Utilisez Azure Traffic Manager pour assurer le basculement du frontal. Si le frontal devient inaccessible dans une région, Traffic Manager acheminera les nouvelles demandes vers la région secondaire. Selon votre solution de base de données, vous devrez peut-être coordonner le basculement de la base de données.

Utilisez le basculement automatique mais la restauration manuelle. Utilisez Traffic Manager pour le basculement automatique, mais pas pour la restauration automatique. La restauration automatique fait peser un certain risque : vous pouvez basculer vers la région primaire avant que celle-ci ne soit totalement saine. Vérifiez plutôt que tous les sous-systèmes d'application sont sains avant d'effectuer une restauration manuelle. En outre, selon la base de données, il peut être nécessaire de vérifier la cohérence des données avant la restauration.

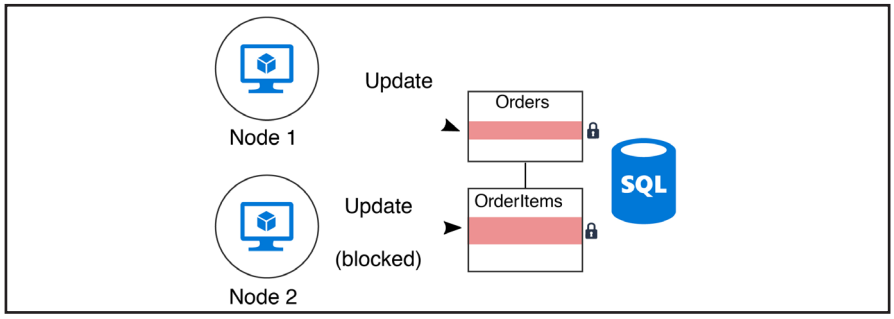
Ajoutez la redondance pour Traffic Manager. Traffic Manager représente un point de défaillance potentiel. Examinez le contrat SLA de Traffic Manager et déterminez si son utilisation seule répond aux besoins de votre entreprise en termes de haute disponibilité. Si ce n'est pas le cas, envisagez d'ajouter une autre solution de gestion de trafic pour la restauration automatique. Si le service Azure Traffic Manager échoue, modifiez vos enregistrements CNAME dans le DNS pour les faire pointer vers l'autre service de gestion de trafic.

Minimiser la coordination

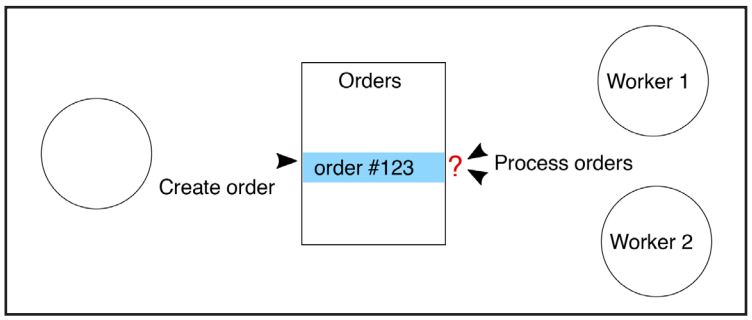
Minimisez la coordination entre les services d'application pour garantir l'évolutivité.

La plupart des applications cloud se composent de plusieurs services d'application : sites web frontaux, bases de données, processus d'entreprise, génération de rapports et analyse, etc. L'évolutivité et la fiabilité des services impliquent que chacun d'eux s'exécute sur plusieurs instances.

Que se passe-t-il lorsque deux instances essaient d'effectuer des opérations simultanées affectant un état partagé ? Dans certains cas, une coordination entre les nœuds est nécessaire, par exemple, pour préserver les garanties ACID. Dans ce schéma, Node2 attend que Node1 déverrouille la base de données :



La coordination limite les avantages de l'évolutivité horizontale et crée des goulots d'étranglement. Dans cet exemple, la contention de verrouillage augmente à mesure que vous faites monter l'application en charge et que vous ajoutez des instances. Dans le pire des cas, les instances frontales passent la majeure partie du temps à patienter durant les verrouillages. La sémantique « exactly-once » (exactement une fois) représente une autre source fréquente de coordination. Par exemple, une commande doit être traitée exactement une fois. Deux workers surveillent les nouvelles commandes. Worker1 sélectionne une commande à traiter. L'application doit veiller à ce que Worker2 ne duplique pas le travail et à ce que la commande ne soit pas abandonnée en cas de blocage de Worker1.



Vous pouvez utiliser un patron comme Scheduler Agent Supervisor pour assurer la coordination entre les workers. Cependant, dans ce cas, le fractionnement du travail peut représenter une meilleure approche. Une certaine plage de commandes est assignée à chaque worker (par exemple, par région facturation). En cas de blocage d'un worker, une nouvelle instance reprend le travail où l'instance précédente s'était arrêtée, sans conflit entre plusieurs instances.

Recommandations

Acceptez la cohérence éventuelle. Lorsque les données sont distribuées, une coordination est nécessaire pour appliquer des garanties de cohérences fortes. Par exemple, supposons qu'une opération mette à jour deux bases de données. Au lieu de la placer dans une étendue de transaction unique, il est préférable que le système accepte une cohérence éventuelle en utilisant, par exemple, le patron Compensating Transaction (Transaction de compensation) pour assurer une restauration logique après une défaillance.

Utilisez les événements de domaine pour synchroniser l'état. Un événement de domaine est un événement qui enregistre les faits significatifs survenant au sein du domaine. Les services intéressés peuvent écouter l'événement au lieu d'utiliser une transaction globale pour assurer la coordination entre plusieurs services. Si cette approche est utilisée, le système doit tolérer la cohérence éventuelle (voir la section précédente).

Envisagez des patrons comme CQRS et l'Event Sourcing. Ces deux patrons peuvent contribuer à réduire la contention entre les scénarios d'usage de lecture et d'écriture.

- Le patron CQRS sépare les opérations de lecture des opérations d'écriture. Dans certaines implémentations, les données de lecture sont physiquement séparées des données d'écriture.

- Dans le patron Event Sourcing (Matérialisation d'événements), les changements d'état sont enregistrés sous la forme d'une série d'événements à ajouter dans une banque de données « append-only ». L'ajout d'un événement au flux est une opération atomique en exigeant un verrouillage minimal.

Ces deux patrons se complètent mutuellement. Si le magasin en écriture seule (patron CQRS) utilise l'Event Sourcing, le magasin en lecture seule peut écouter les mêmes événements pour créer une capture instantanée lisible de l'état actuel, optimisée pour les requêtes. Cependant, avant d'adopter le patron CQRS ou l'Event Sourcing, envisagez les défis que pose cette approche. Pour plus d'informations, consultez CQRS architecture style (Style d'architecture CQRS).

Partitionnez les données. Évitez de placer toutes vos données dans un schéma de données unique partagé par une multitude de services d'application. Une architecture de microservices applique ce principe en rendant chaque service responsable de sa propre banque de données. Dans une base de données unique, le partitionnement des données peut améliorer la concurrence. En effet, un service effectuant une écriture sur une partition n'affecte pas les services effectuant des écritures sur une autre partition.

Concevez des opérations idempotentes. Dans la mesure du possible, concevez les opérations de sorte qu'elles soient idempotentes. De cette façon, elles peuvent être traitées à l'aide de la sémantique « at-least-once » (au moins une fois). Par exemple, vous pouvez placer des éléments de travail dans une file d'attente. En cas de blocage d'un worker durant une opération, un autre worker reprend simplement l'élément de travail.

Utilisez le traitement parallèle asynchrone. Si une opération requiert l'exécution asynchrone de plusieurs étapes (appels de service à distance, par exemple), vous pouvez être en mesure de les appeler en parallèle, puis d'agréger les résultats. Cette approche suppose que chaque étape est indépendante des résultats de l'étape précédente.

Utilisez l'accès concurrentiel optimiste dans la mesure du possible. Le contrôle d'accès concurrentiel pessimiste utilise les verrous de base de données pour prévenir les conflits. Cela peut considérablement altérer les performances et réduire la disponibilité. Avec le contrôle d'accès concurrentiel optimiste, chaque transaction modifie une copie ou une capture instantanée des données. Lorsque la transaction est validée, le moteur de base de données valide la transaction et rejette toute transaction qui affecterait la cohérence de la base de données. Azure SQL Database et SQL Server prennent en charge l'accès concurrentiel optimiste par le biais de l'isolement de capture instantanée. Pour plus d'informations, rendez-vous sur https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql. Certains services de stockage Azure prennent en charge l'accès concurrentiel optimiste grâce à l'utilisation d'Etags (notamment l'API DocumentDB et le stockage Azure). Pour plus d'informations sur l'API DocumentDB, rendez-vous sur https://docs.microsoft.com/en-us/azure/documentdb/documentdb-faq.

Envisagez l'utilisation de MapReduce ou d'autres algorithmes parallèles distribués. Selon les données et le type de travail à effectuer, vous pouvez être en mesure de fractionner le travail en tâches indépendantes qui peuvent être effectuées par plusieurs nœuds travaillant en parallèle. Voir Style d'architecture Big Compute.

Utilisez l'élection de leader pour la coordination. Lorsque vous devez coordonner les opérations, veillez à ce que le coordinateur ne devienne pas un point de défaillance unique dans l'application. Lorsque vous utilisez le patron Leader Election (Élection du leader), une seule instance est leader au même moment et agit à titre de coordinateur. Si le leader échoue, une nouvelle instance est élue pour être le leader.

Conception en faveur de la montée en charge

Concevez votre application de sorte qu'elle bénéficie d'une évolutivité horizontale

L'un des avantages clés du cloud réside dans la mise à l'échelle élastique : la possibilité d'utiliser le niveau de capacité dont vous avez besoin (augmentation de la taille des instances lorsque la charge augmente et diminution de la taille des instances lorsque la capacité supplémentaire n'est pas nécessaire). Concevez votre application de sorte qu'elle bénéficie d'une évolutivité horizontale en ajoutant ou supprimant de nouvelles instances selon la demande.

Recommandations

Évitez l'adhérence des instances. L'adhérence, ou affinité de session, désigne la situation dans laquelle les requêtes du même client sont systématiquement acheminées vers le même serveur. L'adhérence limite la capacité de montée en charge de l'application. Par exemple, le trafic provenant d'un utilisateur à volume élevé ne sera pas distribué entre les instances. L'adhérence peut être causée par le stockage de l'état de session en mémoire et l'utilisation de clés spécifiques à l'ordinateur pour le chiffrement. Veillez à ce que chaque instance puisse traiter n'importe quelle demande.

Identifiez les goulots d'étranglement. La montée en charge n'est pas une solution imparable à tout problème de performances. Par exemple, si votre base de données principale est le goulot d'étranglement, l'ajout de serveurs web ne sera pas très utile. Identifiez et éliminez les goulots d'étranglement dans le système pour résoudre le problème avant d'ajouter des instances. Les parties avec état du système représentent la cause la plus probable des goulots d'étranglement.

Décomposez les scénarios d'usage selon les exigences d'évolutivité. Les applications incluent souvent plusieurs scénarios d'usage présentant différentes exigences en termes de mise à l'échelle. Par exemple, une application peut avoir un site public et un site d'administration distinct. Le site public peut faire l'objet de hausses de trafic soudaines et le site d'administration peut présenter une charge plus petite et plus prévisible.

Déchargez les tâches nécessitant de nombreuses ressources. Dans la mesure du possible, les tâches qui nécessitent beaucoup de ressources en termes d'UC ou d'E/S doivent être déplacées vers les travaux en arrière-plan pour minimiser la charge sur le frontal qui gère les requêtes des utilisateurs.

Utilisez des fonctionnalités de mise à l'échelle automatique intégrées. De nombreux services de calcul Azure bénéficient d'une prise en charge intégrée de la mise à l'échelle automatique. Si l'application possède un scénario d'usage prévisible et régulière, planifiez la montée en charge. Par exemple, effectuez la montée en charge durant les heures d'ouverture. Dans le cas contraire, si le scénario d'usage n'est pas prévisible, utilisez des mesures de performance (UC ou longueur de la file d'attente des requêtes, par exemple) pour déclencher la mise à l'échelle automatique. Pour connaître les bonnes pratiques en matière de mise à l'échelle automatique, consultez Autoscaling (Mise à l'échelle automatique). Pour connaître les bonnes pratiques en matière de mise à l'échelle automatique, rendez-vous sur https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling.

Envisagez une mise à l'échelle automatique agressive pour les scénarios d'usage critiques. Pour les scénarios d'usage critiques, il est judicieux d'anticiper la demande. Il est préférable d'ajouter de nouvelles instances rapidement en condition de charge lourde pour gérer le trafic supplémentaire, puis d'effectuer une descente en puissance progressive.

Prévoyez la conception en vue de la diminution de la taille des instances. N'oubliez pas qu'avec la mise à l'échelle élastique, l'application connaîtra des périodes de diminution de la taille des instances (et de suppression des instances). L'application doit gérer les instances en cours de suppression de façon appropriée. Voici différentes méthodes de traitement de la diminution de la taille des instances :

- Écoutez les événements d'arrêt (lorsque disponible) et fermez proprement.
- Les clients/consommateurs d'un service devraient prendre en charge la gestion d'erreur temporaire et réessayer.
- Pour les tâches de longue durée, envisagez de diviser votre travail, à l'aide de points de contrôle ou du patron Pipes and Filters (Filtres et tubes).
- Placez les éléments de travail dans une file d'attente afin qu'une autre instance puisse sélectionner le travail, si une instance est supprimée en cours de traitement.

Partition pour contourner les limites

Utilisez le partitionnement pour contourner les limites de calcul, de réseau et de base de données.

L'un des avantages clés du Cloud réside dans la mise à l'échelle élastique : la possibilité d'utiliser le niveau de capacité dont vous avez besoin (augmentation de la taille des instances lorsque la charge augmente et diminution de la taille des instances lorsque la capacité supplémentaire n'est pas nécessaire). Concevez votre application de sorte qu'elle bénéficie d'une évolutivité horizontale en ajoutant ou supprimant de nouvelles instances selon la demande.

Dans le cloud, la capacité à monter en puissance de tous les services est limitée. Les limites des services Azure sont documentées dans les contraintes, les quotas et les limites des services de l'abonnement Azure. Les limites incluent le nombre de cœurs, la taille de la base de données, le débit de requête et le débit réseau. Si votre système devient suffisamment important, vous pouvez atteindre une ou plusieurs de ces limites. Utilisez le partitionnement pour contourner ces limites.

Il existe de nombreuses façons de partitionner un système, telles que les suivantes :

- Partitionner une base de données pour éviter les limites de taille de base de données, d'E/S de données ou de nombre de sessions simultanées.
- Partitionner une file d'attente ou un bus de messages pour éviter les limites sur le nombre de demandes ou le nombre de connexions simultanées.
- Partitionner une application web App Service pour éviter les limites sur le nombre d'instances par plan App Service.

Une base de données peut être partitionnée *horizontalement, verticalement,* ou *sur le plan fonctionnel.*

- Dans le partitionnement horizontal, chaque partition contient des données pour un sous-ensemble du jeu de données total. Les partitions partagent le même schéma de données. Par exemple, les clients dont l'initiale est comprise entre A et M vont dans une partition, ceux dont l'initiale est comprise entre N et Z vont dans une autre partition.
- Dans le partitionnement vertical, chaque partition contient un sous-ensemble des champs pour les éléments dans le magasin de données. Par exemple, les champs fréquemment consultés sont placés dans une partition et ceux moins fréquemment consultés dans une autre.
- Dans la répartition fonctionnelle, les données sont partitionnées en fonction de leur mode d'utilisation par chaque contexte délimité dans le système. Par exemple, les données de facturation sont stockées dans une partition et les données d'inventaire des produits dans une autre. Les schémas sont indépendants.

Pour plus d'informations, consultez https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning.

Recommandations

Partitionnez différentes parties de l'application. Les bases de données sont un candidat évident pour le partitionnement, mais envisagez également le stockage, les caches, les files d'attente et les instances de calcul.

Créez la clé de partition pour éviter les zones réactives. Si vous partitionnez une base de données, mais qu'une partition obtient encore la majorité des demandes, alors vous n'avez pas résolu votre problème. Idéalement, la charge est répartie uniformément sur toutes les partitions. Par exemple, hachez par ID client et non par la première lettre du nom du client, car certaines lettres sont plus fréquentes. Le même principe s'applique lors du partitionnement d'une file d'attente de messages. Choisissez une clé de partition qui mène à une répartition égale des messages sur l'ensemble des files d'attente. Pour en savoir plus, consultez Partitionnement.

Partitionnez pour contourner un abonnement Azure et des limites de service. Les services et composants individuels ont des limites, mais il existe aussi des limites pour les abonnements et les groupes de ressources. Pour des demandes très volumineuses, vous devrez peut-être partitionner pour contourner ces limites.

Partitionnez à différents niveaux. Imaginez un serveur de base de données déployé sur une machine virtuelle. La machine virtuelle a un VHD qui est pris en charge par le stockage Azure. Le compte de stockage appartient à un abonnement Azure. Notez que chaque étape inhérente à la hiérarchie a des limites. Le serveur de base de données peut avoir une limite de pool de connexions. Les machines virtuelles ont des limites réseau et de processeur. Le stockage a des limites d'ES par seconde. L'abonnement a des limites sur le nombre de cœurs VM. En règle générale, il est plus facile de partitionner plus bas dans la hiérarchie. Seules les grandes applications devraient avoir besoin d'une partition au niveau de l'abonnement.

Conception pour des opérations

Concevez une application de sorte que l'équipe des opérations dispose des outils dont elle a besoin.

Le cloud a radicalement changé le rôle de l'équipe des opérations. Elle n'est plus chargée de gérer le matériel et l'infrastructure qui héberge l'application. Cela dit, les opérations demeurent un élément essentiel de l'exécution d'une application cloud réussie. Certaines des fonctions importantes de l'équipe des opérations incluent les suivantes :

- Déploiement
- Surveillance
- Escalade
- Réponse aux incidents
- Audits de sécurité

Un suivi et une journalisation robustes sont particulièrement importants dans les applications cloud. Impliquez l'équipe des opérations dans la conception et la planification afin de garantir que l'application lui donne les données et l'aperçu dont elle a besoin pour réussir.

Recommandations

Rendez tous les éléments observables. Une fois qu'une solution est déployée et en cours d'exécution, les fichiers journaux et les suivis sont votre aperçu principal sur le système. Le suivi enregistre un chemin à travers le système et est utile pour identifier les goulots d'étranglement, les problèmes de performances et les points d'échec. La journalisation capture des événements individuels tels que les exceptions, les erreurs et les modifications d'état d'application. Ouvrez une session dans la production, sinon vous perdez l'aperçu aux moments spécifiques où vous en avez le plus besoin.

Instrument de surveillance. La surveillance donne un aperçu de la bonne (ou mauvaise) performance d'une application, en termes de disponibilité, de performance et d'intégrité du système. Par exemple, la surveillance vous indique si vous êtes en conformité avec votre SLA. La surveillance se produit pendant le fonctionnement normal du système. Afin que le personnel des opérations puisse réagir aux problèmes rapidement, il doit être aussi proche que possible du temps réel. Idéalement, la surveillance peut aider à prévenir les problèmes avant qu'ils ne conduisent à un échec critique. Pour plus d'informations, consultez https://docs.microsoft.com/en-us/azure/architecture/best-practices/monitoring.

Instrument pour l'analyse des causes profondes. L'analyse des causes profondes est le processus consistant à identifier la cause sous-jacente des échecs. Elle a lieu après qu'un échec s'est déjà produit.

Utilisez le suivi distribué. Utilisez un système de suivi distribué conçu pour l'échelle cloud, concurrence et asynchronisme. Les suivis devraient inclure un ID de corrélation qui va au-delà des limites de service. Une seule opération peut impliquer des appels à plusieurs services d'application. Si une opération échoue, l'ID de corrélation contribue à identifier la cause de l'échec.

Standardisez les journaux et les mesures. L'équipe des opérations aura besoin d'ajouter des journaux depuis les divers services dans votre solution. Si chaque service utilise son propre format de journalisation, il devient difficile, voire impossible, d'obtenir des informations utiles de leur part. Définissez un schéma commun incluant des champs tels que l'ID de corrélation, le nom de l'événement, l'adresse IP de l'expéditeur et ainsi de suite. Les services individuels peuvent déduire des schémas personnalisés qui reprennent le schéma de base et contiennent des champs supplémentaires.

Automatisez les tâches de gestion. Incluez la mise en service, le déploiement et la surveillance. Automatiser une tâche la rend renouvelable et moins sujette aux erreurs humaines.

Traitez la configuration comme étant du code. Vérifiez les fichiers de configuration dans un système de contrôle de versions afin que vous puissiez suivre et indiquer la version de vos modifications et restaurer si nécessaire.

Utilisation de services gérés

Dans la mesure du possible, utilisez un service PaaS (platform as a service) et non IaaS (infrastructure as a service).

IaaS revient à avoir une boîte de composants. Vous pouvez construire n'importe quoi, mais vous devez assembler vous-même. Les services gérés sont plus faciles à configurer et à administrer. Vous n'avez pas besoin d'approvisionner des machines virtuelles, de configurer des réseaux virtuels, de gérer des correctifs et des mises à jour et toutes les autres surcharges associées à l'exécution d'un logiciel sur une machine virtuelle.

Par exemple, supposons que votre application ait besoin d'une file d'attente de messages. Vous pouvez configurer votre propre service de messagerie sur une machine virtuelle en utilisant quelque chose comme RabbitMQ. Mais Azure Service Bus fournit déjà une messagerie fiable en tant que service qui en outre est plus simple à configurer. Il suffit de créer un espace de noms Service Bus (ce qui peut être fait dans le cadre d'un script de déploiement), puis d'appeler le Service Bus en utilisant le kit de développement client.

Bien sûr, votre application peut avoir des exigences spécifiques qui rendent plus appropriée une approche IaaS. Toutefois, même si votre demande est fondée sur l'IaaS, cherchez les endroits où il est naturel d'intégrer des services gérés. Ils peuvent inclure le stockage de données, les files d'attente et le cache.

Au lieu d'exécuter...

- Active Directory
- Elasticsearch
- Hadoop
- IIS
- Mongo DBC
- Redis
- SQL Server

Envisagez d'utiliser...

- Azure Active Directory Domain Services
- Azure Search
- HDInsight
- App Service
- Cosmos DB
- Cache Redis Azure
- Azure SQL Database

Utilisation du meilleur magasin de données pour la tâche

Choisissez la technologie de stockage la mieux adaptée à vos données et à l'usage prévu.

IaaS revient à avoir une boîte de composants. Vous pouvez construire n'importe quoi, mais vous devez assembler vous-même. Les services gérés sont plus faciles à configurer et à administrer. Vous n'avez pas besoin d'approvisionner des machines virtuelles, de configurer des réseaux virtuels, de gérer des correctifs et des mises à jour et toutes les autres surcharges associées à l'exécution d'un logiciel sur une machine virtuelle.

Par exemple, supposons que votre application ait besoin d'une file d'attente de messages. Vous pouvez configurer votre propre service de messagerie sur une machine virtuelle en utilisant quelque chose comme RabbitMQ. Mais Azure Service Bus fournit déjà une messagerie fiable en tant que service qui en outre est plus simple à configurer. Il suffit de créer un espace de noms Service Bus (ce qui peut être fait dans le cadre d'un script de déploiement), puis d'appeler le Service Bus en utilisant le kit de développement client.

Bien sûr, votre application peut avoir des exigences spécifiques qui rendent plus appropriée une approche IaaS. Toutefois, même si votre demande est fondée sur l'IaaS, cherchez les endroits où il est naturel d'intégrer des services gérés. Ils peuvent inclure le stockage de données, les files d'attente et le cache.

L'époque où il vous suffisait de placer toutes vos données dans une grande base de données SQL relationnelle est révolue. Les bases de données relationnelles sont très efficaces dans leur domaine : fournir des garanties ACID pour les transactions sur les données relationnelles. Mais elles sont onéreuses :

- Les requêtes peuvent nécessiter des jonctions coûteuses.
- Les données doivent être normalisées et conformes à un schéma prédéfini (schéma pour écriture).
- La contention de verrouillage peut affecter les performances.

Dans n'importe quelle solution de grande taille, il est probable qu'une technologie de magasin de données unique ne va pas combler tous vos besoins. Les alternatives aux bases de données relationnelles incluent les magasins de clés/valeurs ainsi que les bases de données de documents, de moteurs de recherche, de séries chronologiques, de familles de colonnes et de graphiques.

Chacune a des avantages et des inconvénients, et les différents types de données s'intègrent plus naturellement dans l'une ou l'autre.

Par exemple, vous pouvez stocker un catalogue de produits dans une base de données de documents, tels que Cosmos DB, ce qui permet un schéma flexible. Dans ce cas, la description de chaque produit est un document autonome. Pour les requêtes sur l'ensemble du catalogue, vous pouvez indexer le catalogue et stocker l'index dans la recherche Azure. L'inventaire des produits pourrait aller dans une base de données SQL, car ces données nécessitent des garanties ACID.

N'oubliez pas que les données comprennent davantage que de simples données d'application persistantes. Elles incluent également des journaux d'application, des événements, des messages et des caches.

Recommandations

N'utilisez pas une base de données relationnelle pour tout. Envisagez d'autres magasins de données lorsque cela est approprié. Consultez Choisissez le magasin de données approprié.

Incluez la persistance polyglotte. Dans n'importe quelle solution de grande taille, il est probable qu'une technologie de magasin de données unique ne va pas combler tous vos besoins.

Examinez le type de données. Par exemple, placez des données transactionnelles dans SQL, des documents JSON dans une base de données de documents, des données de télémétrie dans une base de données de série chronologique, des journaux d'applications dans Elasticsearch et des objets blobs dans le stockage Blob Azure.

Préférez la disponibilité à la cohérence (forte). Le théorème CAP implique qu'un système distribué doive faire des compromis entre la disponibilité et la cohérence. (Les partitions réseau, l'autre branche du théorème CAP, ne peuvent jamais être complètement évitées.) Souvent, vous pouvez obtenir une plus grande disponibilité en adoptant un modèle final de cohérence.

Examinez l'ensemble des compétences de l'équipe de développement. Il y a des avantages à l'utilisation de la persistance polyglotte, mais il est possible d'exagérer. L'adoption d'une nouvelle technologie de stockage de données nécessite un nouvel ensemble de compétences. L'équipe de développement doit comprendre comment optimiser la technologie. Elle doit comprendre les modes d'utilisation appropriés, comment optimiser les requêtes, ajuster les performances et ainsi de suite. C'est un élément à inclure lorsque l'on envisage des technologies de stockage.

Utilisez des transactions de compensation. Un effet secondaire de la persistance polyglotte est qu'une transaction unique peut écrire des données dans plusieurs magasins. Si quelque chose échoue, utilisez des transactions de compensation pour annuler toute étape déjà terminée.

Regardez les contextes limités. Un contexte limité est un terme de conception orientée domaine. Un contexte délimité est une limite explicite autour d'un modèle de domaine qui définit à quelles parties du domaine le patron s'applique. Idéalement, un contexte délimité correspond à un sous-domaine du domaine d'affaires. Les contextes délimités dans votre système sont un emplacement naturel pour envisager une persistance polyglotte. Par exemple, les « produits » peuvent apparaître dans les deux sous-domaines Catalogue des produits et Inventaire des produits, mais il est très probable que ces deux sous-domaines ont des exigences différentes en matière de stockage, de mise à jour et d'interrogation des produits.

Conception pour l'évolution

La capacité à innover en continu repose sur une conception évolutive

Toutes les applications réussies changent au cours du temps, qu'il s'agisse de corriger des bogues, d'ajouter de nouvelles fonctions, d'apporter de nouvelles technologies ou de rendre des systèmes existants plus extensibles et plus résistants. Si toutes les parties d'une application sont étroitement couplées, il devient très difficile d'introduire des changements dans le système. Un changement dans une partie de l'application peut arrêter une autre partie, ou entraîner des modifications par ondulation dans l'ensemble du code base.

Ce problème n'est pas limité aux applications monolithiques. Une application peut être décomposée en services, mais continuer de présenter le type de couplage serré qui aboutit sur un système rigide et cassant. Mais lorsque des services sont conçus pour évoluer, les équipes peuvent innover et apporter sans cesse de nouvelles fonctions.

Les microservices deviennent un moyen populaire d'atteindre une conception évolutive, parce qu'ils traitent nombre des considérations énumérées ici.

Recommandations

Appliquez une forte cohésion et assouplissez le couplage. Un service est cohésif s'il fournit des fonctionnalités qui vont logiquement de pair. Les services sont faiblement couplés si vous pouvez modifier un service sans changer l'autre. Une cohésion forte signifie généralement que les changements apportés à une fonction exigeront des changements dans d'autres fonctions connexes. Si vous découvrez que la mise à jour d'un service nécessite d'apporter des mises à jour coordonnées à d'autres services, cela peut être un signe d'un manque de cohésion entre vos services. Un des objectifs de conception orientée domaine (DDD) consiste à identifier ces limites.

Encapsulez des connaissances du domaine. Quand un client consomme un service, la responsabilité de faire respecter les règles métier du domaine ne doit pas incomber au client. Au lieu de cela, le service doit encapsuler toutes les connaissances du domaine qui relèvent de sa responsabilité. Dans le cas contraire, chaque client est tenu d'appliquer les règles métier, et vous vous retrouvez avec des connaissances du domaine réparties sur différentes parties de l'application.

Utilisez la messagerie asynchrone. La messagerie asynchrone est une manière de découpler le producteur de messages du consommateur. Le producteur ne dépend pas du consommateur répondant au message ou prenant des mesures particulières. Avec une architecture publication/abonnement, le producteur peut ne même pas savoir qui consomme le message. De nouveaux services peuvent facilement consommer les messages sans aucune modification au producteur.

Ne développez pas de connaissances du domaine dans une passerelle. Les passerelles peuvent être utiles dans une architecture de microservices, pour des éléments tels que le routage de demandes, la translation de protocole, l'équilibrage de charge ou l'authentification. Toutefois, la passerelle doit être limitée à ce genre de fonctionnalité d'infrastructure. Pour éviter de devenir une dépendance lourde, elle ne doit implémenter aucune connaissance du domaine.

Exposez des interfaces ouvertes. Évitez de créer des couches de translation personnalisée qui se trouvent entre des services. Au lieu de cela, un service doit exposer une API avec un contrat d'API bien défini. L'API doit être avec version afin que vous puissiez développer l'API tout en conservant une compatibilité descendante. De cette façon, vous pouvez mettre à jour un service sans coordonner les mises à jour de tous les services en amont qui en dépendent. Les services publics doivent exposer une API RESTful sur HTTP. Pour des raisons de performances, les services principaux sont susceptibles d'utiliser un protocole de messagerie de type RPC.

Concevez et testez au regard des contrats de service. Lorsque des services exposent des API bien définies, vous pouvez développer et tester au regard de ces API. De cette façon, vous pouvez développer et tester un service individuel sans accélérer tous ses services dépendants. (Bien sûr, vous devrez tout de même procéder à une intégration et un test de charge au regard de services réels.)

Éloignez l'infrastructure de la logique de domaine. Ne laissez pas la logique de domaine se mélanger avec des fonctionnalités liées aux infrastructures, telles que la messagerie ou la persistance. Dans le cas contraire, des changements dans la logique de domaine nécessiteront des mises à jour pour les couches d'infrastructure et inversement.

Déchargez des préoccupations transversales à un service distinct. Par exemple, si plusieurs services doivent authentifier des demandes, vous pouvez déplacer cette fonctionnalité dans son propre service. Puis vous pouvez faire évoluer le service d'authentification, par exemple, en ajoutant un nouveau flux d'authentification, sans toucher aucun des services qui l'utilisent.

Déployez des services indépendamment. Quand l'équipe de DevOps peut déployer un service unique indépendamment d'autres services dans l'application, des mises à jour peuvent arriver plus rapidement et en toute sécurité. Des corrections de bogues et de nouvelles fonctionnalités peuvent être proposées plus régulièrement. Concevez à la fois l'application et le processus de mise en production pour prendre en charge des mises à jour indépendantes.

Développement pour les besoins de l'entreprise

Chaque décision de conception doit être justifiée par une exigence de l'entreprise.

Ce principe de conception peut sembler évident, mais il est essentiel de le garder à l'esprit lorsque vous concevez une solution. Envisagez-vous des millions d'utilisateurs, ou des milliers ? Une interruption de l'application d'une heure est-elle acceptable ? Attendez-vous des pointes importantes de trafic, ou une charge de travail très prévisible ? En fin de compte, chaque décision de conception doit être justifiée par une exigence de l'entreprise.

Recommandations

Définissez des objectifs d'affaires. dont l'objectif de délai de récupération (RTO), l'objectif de point de récupération (RPO) et l'interruption maximale tolérable (MTO). Ces chiffres devraient éclairer les décisions sur l'architecture. Par exemple, pour atteindre un RTO faible, vous pouvez implémenter un basculement automatisé dans une région secondaire. Mais si votre solution peut tolérer un RTO plus élevé, ce degré de redondance peut être inutile.

Documentez les contrats de niveau de service (SLA) et les objectifs de niveau de service (SLO). Incluez les mesures de disponibilité et de performances. Vous pouvez développer une solution qui offre une disponibilité de 99,95 %. Est-ce suffisant ? La réponse est une décision d'entreprise.

Modélisez l'application autour du domaine d'entreprise. Commencez par analyser les besoins métier. Utilisez ces exigences pour modéliser l'application. Envisagez d'utiliser une approche de conception orientée domaine (DDD) pour créer des modèles de domaine qui reflètent les processus d'affaires et les cas d'utilisation.

Capturez les exigences fonctionnelles et non fonctionnelles. Les exigences fonctionnelles vous permettent de juger si l'application fonctionne correctement. Les exigences non fonctionnelles vous permettent de juger si l'application fonctionne correctement. En particulier, assurez-vous que vous comprenez vos exigences d'évolutivité, de disponibilité et de latence. Ces exigences influenceront les décisions de conception et le choix de la technologie.

Décomposez par charge de travail. Dans ce contexte, le terme « charge de travail » signifie une tâche informatique ou une capacité discrète, qui peut être différenciée logiquement des autres tâches. Des charges de travail différentes peuvent avoir des exigences différentes en termes de disponibilité, d'évolutivité, de cohérence des données et de récupération d'urgence.

Plan de croissance. Une solution pourrait répondre à vos besoins actuels, en termes de nombre d'utilisateurs, de volume des transactions, de stockage de données et ainsi de suite. Toutefois, une application robuste peut gérer la croissance sans changements architecturaux majeurs. Voir Conception pour monter en charge et Partition pour contourner les limites. Envisagez également que le modèle et les besoins de votre entreprise changeront probablement avec le temps. Si des patrons de données et un modèle de service d'application sont trop rigides, il devient difficile de faire évoluer l'application pour de nouveaux scénarios et cas d'utilisation. Voir Conception pour l'évolution.

Gérez les coûts. Dans une application locale traditionnelle, vous payez à l'avance pour le matériel (CAPEX). Dans une application cloud, vous payez les ressources que vous consommez. Assurez-vous de comprendre le modèle de tarification pour les services que vous consommez. Le coût total comprendra l'utilisation de la bande passante réseau, le stockage, les adresses IP, la consommation de services et d'autres facteurs. Consultez la tarification Azure pour plus d'informations. Pensez également à vos coûts des opérations. Dans le cloud, vous n'avez pas à gérer le matériel ou d'autres infrastructures, mais vous devez encore gérer vos applications, telles que DevOps, la réponse aux incidents, récupération d'urgence et ainsi de suite.

Conception d'applications résilientes pour Azure

Plutôt que d'acheter du matériel haut de gamme pour monter en puissance, dans un environnement cloud vous devez monter en charge. Les coûts pour les environnements cloud restent modiques et le but est de minimiser les conséquences d'un échec.

Dans un système distribué, des échecs vont se produire. Le matériel peut faire l'objet de défaillances. Le réseau peut présenter des défaillances passagères. L'ensemble d'un service ou d'une région peut-être confronté à une perturbation. Même si le cas est rare, il doit être prévu.

Développer une application fiable dans le cloud est différent de créer une application fiable dans un environnement d'entreprise. Bien que vous ayez pu acheter par le passé du matériel plus haut de gamme pour monter en puissance, dans un environnement cloud, vous devez au lieu de cela monter en charge. Les coûts pour les environnements cloud restent modiques grâce à l'utilisation du matériel de base. Au lieu de focaliser sur la prévention des échecs et l'optimisation du « délai moyen entre les échecs », dans ce nouvel environnement, l'accent se déplace sur le « délai moyen de restauration ». L'objectif est de minimiser les conséquences d'un échec.

Cet article présente la façon de développer des applications résilientes dans Microsoft Azure. Il commence par une définition du terme résilience et des concepts connexes. Il décrit ensuite un processus permettant d'atteindre la résilience, en utilisant une approche structurée au cours de la durée de vie d'une application, depuis la conception et la mise en œuvre jusqu'au déploiement et aux opérations.

Qu'est-ce que la résilience ?

La **résilience** est la capacité d'un système à récupérer après des échecs pour continuer à fonctionner. Il ne s'agit pas d'éviter les échecs, mais d'y répondre d'une manière qui évite la perte de données ou les temps d'arrêt. La résilience vise à permettre à l'application de refonctionner pleinement suite à un échec.

Deux aspects importants de la résilience sont la haute disponibilité et la récupération d'urgence.

- **La haute disponibilité** (HA) est la capacité de l'application à continuer de s'exécuter dans un état d'intégrité correct, sans temps d'arrêt important. Par « état d'intégrité correct », nous entendons que l'application répond et que les utilisateurs peuvent se connecter à l'application et interagir avec elle.

- **La récupération d'urgence** (DR) est la possibilité de récupérer après des incidents rares mais importants : échecs non passagers, à grande échelle, tels que l'interruption de service qui affecte l'ensemble d'une région. La récupération d'urgence comprend l'archivage et la sauvegarde des données. Elle peut inclure des interventions manuelles, telles que la restauration d'une base de données à partir d'une sauvegarde.

Une façon de penser à la HA par rapport à la DR est que la DR commence lorsque l'impact d'une erreur dépasse la capacité de la conception HA à y faire face. Par exemple, placer plusieurs machines virtuelles derrière un équilibreur de charge fournira de la disponibilité si une machine virtuelle échoue, mais pas si elles sont toutes en panne en même temps.

Lorsque vous concevez une application pour qu'elle soit résiliente, vous devez comprendre vos besoins de disponibilité. Combien de temps d'arrêt est acceptable ? Cela dépend en partie du coût. Combien l'arrêt potentiel coûtera-t-il à votre entreprise ? Combien devriez-vous investir pour rendre l'application hautement disponible ? Vous devez également définir ce qu'implique le fait que l'application soit disponible. Par exemple, l'application est-elle « en panne », si un client peut soumettre une commande, mais que le système ne peut pas la traiter dans les délais normaux ? Envisagez également la probabilité d'un type particulier de panne se produisant, et si une stratégie d'atténuation est rentable.

Un autre terme courant est la **continuité de service** (BC), qui est la capacité d'exécuter des fonctions commerciales essentielles pendant et après des conditions défavorables, telles qu'une catastrophe naturelle ou un service en panne. La BC couvre l'ensemble de l'opération de l'entreprise, y compris l'informatique, les transports, les communications, les personnes et les installations physiques. Cet article se concentre sur des applications cloud, mais la planification de la résilience doit se faire dans le contexte de l'ensemble des besoins de BC. Pour plus d'informations, consultez le [Contingency Planning Guide] [capacity-planning-guide] du National Institute of Science and Technology (NIST).

Processus pour atteindre la résilience

La résilience n'est pas une extension. Elle doit être conçue dans le système et mise en pratique. Voic un modèle général à suivre :

1. **Définissez** vos besoins en disponibilité en fonction des besoins de l'entreprise.
2. **Concevez** l'application en vue de sa résilience. Commencez par une architecture conforme à des pratiques éprouvées, puis identifiez les points d'échec possibles dans cette architecture.
3. **Mettez en œuvre** des stratégies de détection et de récupération des échecs.
4. **Testez** la mise en œuvre en simulant des erreurs et en déclenchant des basculements forcés.
5. **Déployez** l'application en production à l'aide d'un procédé fiable et reproductible.
6. **Surveillez** l'application pour détecter les échecs. En surveillant le système, vous pouvez évaluer l'intégrité de l'application et réagir aux incidents si nécessaire.
7. **Répondez** en cas d'incidents nécessitant des interventions manuelles.

Dans le reste de cet article, nous abordons chacune de ces étapes plus en détail.

Définition de vos besoins de résilience

La planification de la résilience commence avec les besoins de l'entreprise. Voici quelques approches pour réfléchir à la résilience en ces termes.

Décomposez par charge de travail

Plusieurs solutions cloud se composent de plusieurs charges de travail d'application. Dans ce contexte, le terme « charge de travail » signifie une capacité discrète ou la tâche informatique, qui se distingue logiquement des autres tâches, en termes d'exigences de stockage des données et de logique métier. Par exemple, une application de commerce électronique peut inclure les charges de travail suivantes :

- Parcourir un catalogue de produits et y rechercher des informations.
- Créer et suivre des commandes.
- Consulter des recommandations.

Ces charges de travail peuvent avoir des exigences différentes pour la disponibilité, l'évolutivité, la cohérence des données, la récupération d'urgence et ainsi de suite. Encore une fois, il s'agit de décisions professionnelles.

Envisagez également des patrons d'utilisation. Y a-t-il des périodes critiques où le système doit être disponible ? Par exemple, un service de déclaration de revenus ne peut tomber en panne juste avant la date limite de dépôt, un service de flux vidéo doit rester actif pendant un important événement sportif grand et ainsi de suite. Pendant les périodes critiques, vous pouvez avoir des déploiements redondants sur plusieurs régions, ce qui fait que l'application est susceptible de basculer si une région a échoué. Toutefois, un déploiement sur plusieurs régions est plus cher, donc pendant les périodes moins critiques, vous pouvez exécuter l'application dans une seule région.

RTO et RPO

Deux paramètres importants à considérer sont l'objectif de délai de récupération et l'objectif de point de récupération.

- **L'objectif de délai de récupération** (RTO) est la durée maximale acceptable durant laquelle une application peut être indisponible après un incident. Si votre RTO est de 90 minutes, vous devez être en mesure de restaurer l'application dans les 90 minutes suivant le début d'un sinistre. Si vous avez un RTO très bas, vous pouvez conserver un deuxième déploiement en mode veille continu pour garantir une protection contre une panne régionale.
- **L'objectif de point de récupération** (RPO) est la durée maximale de perte de données qui est acceptable en cas de sinistre. Par exemple, si vous stockez des données dans une base de données unique, sans réplication vers d'autres bases de données et que vous effectuez des sauvegardes horaires, vous risquez de perdre jusqu'à une heure de données.

Les RTO et RPO sont des impératifs de l'entreprise. Une évaluation des risques peut vous aider à définir le RTO et le RPO de l'application. Une autre mesure commune est le **délai moyen de récupération** (MTTR), qui correspond au temps moyen qu'il faut pour rétablir l'application après un échec. Le MTTR est un fait empirique concernant un système. Si le MTTR dépasse le RTO, alors un échec du système provoquera une désorganisation inacceptable de l'entreprise, car il ne sera pas possible de restaurer le système selon le RTO défini.

SLA

Dans Azure, le contrat de niveau de service (SLA) décrit les engagements de Microsoft en termes de disponibilité et de connectivité. Si le SLA pour un service particulier est de 99,9 %, cela signifie que vous devriez vous attendre à ce que le service soit disponible 99,9 % du temps.

> Remarques :
>
> Le SLA d'Azure comprend également des dispositions pour l'obtention d'un avoir service si le SLA n'est pas respecté, ainsi que des définitions précises de la « disponibilité » de chaque service. Cet aspect du SLA agit comme une stratégie d'application.

SLAD	Temps d'arrêt par semaine	Temps d'arrêt par mois	Temps d'arrêt par an
99%	1,68 heure	7,2 heures	365 jours
99.9%	10,1 minutes	43,2 minutes	8,76 heures
99.95%	5 minutes	21,6 minutes	4,38 heures
99.99%	1,01 minute	4,32 minutes	52,56 minutes
99.999%	6 secondes	25,9 secondes	5,26 minutes

Bien sûr, une plus grande disponibilité est préférable, toutes choses égales par ailleurs. Mais alors que vous cherchez à obtenir davantage de 9, le coût et la complexité pour atteindre ce niveau de disponibilité se développent. Une disponibilité de 99,99 % se traduit par environ 5 minutes de temps d'arrêt total par mois. Le coût et la complexité supplémentaires pour atteindre cinq 9 sont-ils rentables ? La réponse dépend des exigences de l'entreprise.

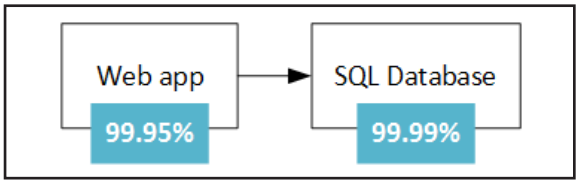
Voici quelques autres considérations lorsque vous définissez un SLA :

- Pour atteindre quatre 9 (99,99 %), vous ne pouvez probablement pas compter sur intervention manuelle pour récupérer après des échecs. L'application doit procéder à un auto-diagnostic et une réparation spontanée.
- Au-delà de quatre 9, il est difficile de détecter les pannes suffisamment rapidement pour se conformer au SLA.
- Pensez à la fenêtre de temps de mesure de votre SLA. Plus la fenêtre est étroite, plus les tolérances sont serrées. Il n'est probablement pas judicieux de définir vos SLA en termes de durée de fonctionnement horaire ou journalière.

SLA composites

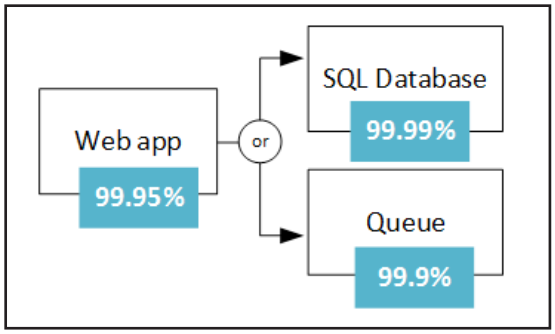
Considérez une application web App Service qui écrit dans Azure SQL Database. Au moment de la rédaction du présent texte, ces services Azure ont les SLA suivants :

- App Service Web Apps = 99,95 %
- SQL Database = 99,99 %



Quel est le temps d'indisponibilité maximale que vous pouvez attendre pour cette application ? Si un service échoue, l'ensemble de l'application échoue. En général, la probabilité d'échec de chaque service est indépendante, donc le SLA composite pour cette application est de 99,95 % × 99,99 % = 99,94 %. C'est plus bas que les SLA individuels, ce qui n'est pas surprenant, car une application qui s'appuie sur plusieurs services possède davantage de points d'échec potentiels.

En revanche, vous pouvez améliorer le SLA composite en créant des voies de secours indépendantes. Par exemple, si SQL Database n'est pas disponible, placez des transactions dans une file d'attente pour les traiter ultérieurement.



Grâce à cette conception, l'application reste disponible, même si elle ne peut pas se connecter à la base de données. Cependant, elle échoue si la base de données et la file d'attente échouent toutes deux en même temps. Le pourcentage de temps prévu pour un échec simultané est 0,0001 × 0,001, donc le SLA composite pour ce chemin d'accès combiné est comme suit :

- Base de données OU file d'attente = 1,0 − (0,0001 × 0,001) = 99,99999 %

Le SLA composite total est comme suit :

- Application web ET (base de données OU file d'attente) = 99,95 % × 99,99999 % = ~99,95 %

Mais il y a des compromis à cette approche. La logique d'application est plus complexe, vous payez pour la file d'attente, et il peut y avoir des problèmes de cohérence de données à prendre en considération.

SLA pour les déploiements multi-région. Une autre technique HA consiste à déployer l'application dans plusieurs régions et à utiliser Azure Traffic Manager pour basculer si l'application connaît un échec dans une région. Pour un déploiement dans deux régions, le SLA composite est calculé comme suit.

Considérons *N*, le SLA composite pour l'application déployée dans une région. Le risque attendu que l'application connaisse un échec dans les deux régions en même temps est le suivant : (1 − N) × (1 − N). Par conséquent,

- Le SLA combiné pour les deux régions = 1 − (1 − N)(1 − N) = N + (1 − N)N

Enfin, vous devez factoriser le SLA pour Traffic Manager. Au moment de la rédaction du présent texte, le SLA pour Traffic Manager est de 99,99 %.

- SLA composite = 99,99 % × (SLA combiné pour les deux régions)

En outre, un basculement n'est pas instantané et peut entraîner des temps d'arrêt en cours de basculement. Voir Basculement et surveillance de point de terminaison de Traffic Manager.

Le nombre calculé de SLA est une base utile, mais il ne fournit pas toutes les informations sur la disponibilité. Souvent, une application peut connaître un échec normal lorsqu'un chemin d'accès non critique échoue. Imaginez une application affichant un catalogue de livres. Si l'application ne peut pas récupérer l'image miniature de la couverture, elle peut afficher une image d'espace réservé. Dans ce cas, ne pas obtenir l'image ne réduit pas la durée de fonctionnement de l'application, bien qu'elle affecte l'expérience de l'utilisateur.

Conception en vue de la résilience

Au cours de la phase de conception, vous devez effectuer une analyse du mode d'échec (FMA). Une FMA vise à identifier des points d'échec possibles et à définir comment l'application va répondre à ces échecs.

- Comment l'application va-t-elle détecter ce type d'échec ?
- Comment l'application va-t-elle répondre à ce type d'échec ?
- Comment allez-vous consigner et surveiller ce type d'échec ?

Pour plus d'informations sur le processus FMA, avec des recommandations précises pour Azure, reportez-vous à Aide à la résilience Azure : analyse du mode d'échec.

Au cours de la phase de conception, vous devez effectuer une analyse du mode d'échec (FMA). Une FMA vise à identifier des points d'échec possibles et à définir comment l'application va répondre à ces échecs.

- Comment l'application va-t-elle détecter ce type d'échec ?
- Comment l'application va-t-elle répondre à ce type d'échec ?
- Comment allez-vous consigner et surveiller ce type d'échec ?

Pour plus d'informations sur le processus FMA, avec des recommandations précises pour Azure, reportez-vous à Aide à la résilience Azure : analyse du mode d'échec.

Mode d'échec	Stratégie de détection
Le service n'est pas disponible	HTTP 5xx
Limitation	HTTP 429 (trop de demandes)
Authentification	HTTP 401 (non autorisé)
Réponse lente	Expiration de la demande

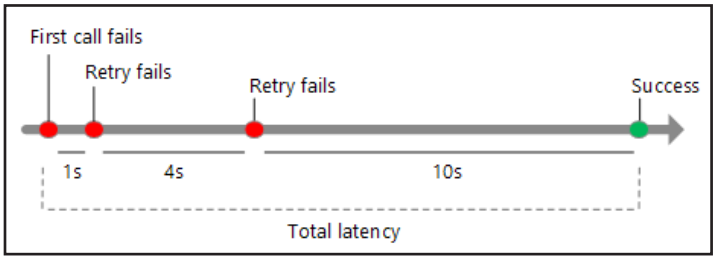
Stratégies de résilience

Cette section fournit une enquête concernant certaines stratégies de résilience. La plupart ne se limitent pas à une technologie particulière. Les descriptions de cette section résument l'idée générale derrière chaque technique, avec des liens vers d'autres lectures.

Réessayer des échecs passagers

Les échecs passagers peuvent résulter de la perte momentanée de la connectivité réseau, d'un abandon de connexion de base de données ou d'un délai d'expiration lorsqu'un service est occupé. Souvent, un échec passager peut être résolu simplement par une nouvelle tentative de la demande. Pour de nombreux services Azure, le kit de développement logiciel client implémente des relances automatiques, d'une manière qui est transparente à l'appelant ; voir Aide spécifique pour réessayer un service.

Chaque tentative ajoute à la latence totale. En outre, un trop grand nombre d'échecs de demandes peut générer un goulot d'étranglement, car les demandes en attente s'accumulent dans la file d'attente. Ces demandes bloquées peuvent contenir des ressources système critiques telles que la mémoire, les fils de discussion, les connexions de base de données et ainsi de suite, susceptibles de provoquer des échecs en cascade. Pour éviter cela, augmentez l'intervalle entre chaque tentative et limitez le nombre total d'échecs de demandes.



Pour plus d'informations, consultez Patron Retry (Nouvelle tentative).

Équilibrer la charge sur plusieurs instances

À des fins d'évolutivité, une application cloud devrait pouvoir monter en charge en ajoutant des instances. Cette approche améliore également la résilience, car les instances défectueuses peuvent être retirées de la rotation.

Par exemple :

- Placez deux ou plusieurs machines virtuelles derrière un équilibreur de charge. L'équilibreur de charge répartit le trafic pour toutes les machines virtuelles. Voir Exécuter des machines virtuelles à charge équilibrée à des fins d'évolutivité et de disponibilité.
- Monter en charge une application Azure App Service sur plusieurs instances. App Service équilibre automatiquement la charge sur les instances. Voir Application web de base.
- Utilisez Azure Traffic Manager pour répartir le trafic sur un ensemble de points de terminaison.

Répliquer des données

La réplication de données est une stratégie générale pour le traitement d'échecs non passagers dans un magasin de données. Plusieurs technologies de stockage fournissent une réplication intégrée, y compris Azure SQL Database, Cosmos DB et Apache Cassandra.

Il est important de considérer les deux chemins : de lecture et d'écriture. Selon la technologie de stockage, vous pouvez avoir plusieurs réplicas accessibles en écriture, ou un seul réplica accessible en écriture et plusieurs réplicas en lecture seule.

Pour maximiser la disponibilité, les réplicas peuvent être placés dans plusieurs régions. Cependant, cela augmente la latence lors de la réplication des données. En général, une réplication sur toutes les régions est réalisée de façon asynchrone, ce qui implique un éventuel modèle de cohérence et la perte potentielle de données en cas d'échec d'un réplica.

Dégrader le service de façon appropriée

Si un service échoue et qu'il n'y a aucun chemin de basculement, l'application peut être capable de dégrader correctement tout en offrant une expérience utilisateur acceptable.

Par exemple :

- Placer un élément de travail dans une file d'attente pour le traiter plus tard.
- Retourner une valeur estimée.
- Utiliser des données mises en cache localement.
- Montrer à l'utilisateur un message d'erreur. (Cette option est préférable à un arrêt de réponse aux demandes de la part de l'application).

Limiter les utilisateurs à volumes élevés

Parfois, un petit nombre d'utilisateurs crée une charge excessive. Cela peut avoir un impact sur les autres utilisateurs en réduisant la disponibilité globale de votre application.

Lorsqu'un seul client génère un nombre excessif de demandes, l'application peut limiter ce client pendant une certaine période de temps. Au cours de la période de limitation, l'application refuse tout ou partie des demandes du client (en fonction de la stratégie de limitation exacte). Le seuil pour la limitation peut dépendre du niveau de service du client.

La limitation n'implique pas que le client a nécessairement agi avec malveillance, seulement qu'il a dépassé son quota de service. Dans certains cas, un consommateur peut dépasser systématiquement son quota ou mal se comporter d'une autre manière. Dans ce cas, vous pouvez aller plus loin et bloquer l'utilisateur. En général, cela se fait en bloquant une clé API ou une plage d'adresses IP.

Pour plus d'informations, consultez Patron Throttling (Limitation).

Utiliser un disjoncteur

Le patron Circuit Breaker (Disjoncteur) peut empêcher une application d'essayer à plusieurs reprises une opération qui est susceptible d'échouer. Cela est similaire à un disjoncteur physique, un interrupteur qui stoppe le flux de courant lorsqu'un circuit est surchargé.

Le disjoncteur encapsule des appels à un service. Il a trois états :

- **Fermé.** Il s'agit de l'état normal. Le disjoncteur envoie des requêtes au service, et un compteur suit le nombre d'échecs récents. Si le nombre d'échecs dépasse un certain seuil dans un laps de temps donné, le disjoncteur bascule vers l'état ouvert.
- **Ouvert.** Dans cet état, le disjoncteur met immédiatement en échec toutes les demandes, sans appeler le service. L'application doit utiliser un chemin d'accès d'atténuation, tel que la lecture des données à partir d'un réplica ou simplement retourner une erreur à l'utilisateur. Quand le disjoncteur passe en ouvert, il démarre une minuterie. Lorsque la minuterie expire, le disjoncteur bascule vers l'état semi-ouvert.
- **Semi-ouvert.** Dans cet état, le disjoncteur laisse un nombre limité de demandes accéder au service. Si elles passent, le service est supposé être récupéré et le disjoncteur revient à l'état fermé. Dans le cas contraire, il revient à l'état ouvert. L'état semi-ouvert empêche un service de récupération d'être soudainement inondé de demandes.

Pour plus d'informations, consultez Patron Circuit Breaker (Disjoncteur).

Utiliser le nivellement de charge pour lisser les pics de trafic

Les applications peuvent rencontrer de soudains pics de trafic, ce qui est susceptible de submerger les services sur le serveur principal. Si un service de serveur principal ne peut répondre aux demandes assez rapidement, cela peut entraîner une mise en file d'attente des demandes (sauvegarde), ou une limitation de l'application par le service.

Pour éviter cela, vous pouvez utiliser une file d'attente comme tampon. Lorsqu'il y a un nouvel élément de travail, au lieu d'appeler le service principal immédiatement, l'application place en file d'attente un élément de travail à exécuter de façon asynchrone. La file d'attente agit comme un tampon qui atténue les pics de charge.

Pour de plus amples informations, voir patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente).

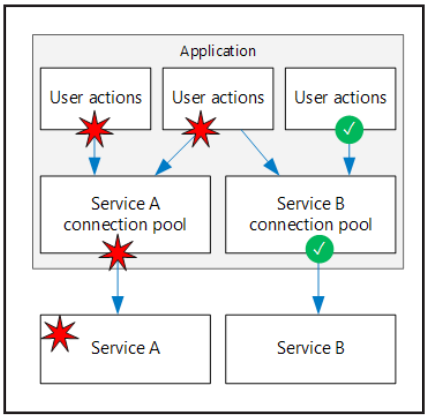
Isoler les ressources critiques

Les échecs dans un sous-système peuvent parfois s'afficher en cascade, entraînant des échecs dans d'autres parties de l'application. Cela peut se produire si, à la suite d'une défaillance, certaines ressources (threads ou sockets, par exemple) ne sont pas libérées suffisamment rapidement, conduisant à l'épuisement des ressources.

Pour éviter cela, vous pouvez partitionner un système en groupes isolés, de sorte qu'un échec affectant une partition n'arrête pas l'ensemble du système. Cette technique est parfois appelée le patron de cloison.

Exemples :

- Partitionnez une base de données (par exemple, par client) et attribuez un pool distinct d'instances de serveur web pour chaque partition.
- Utilisez des pools de threads séparés pour isoler les appels aux différents services. Cela aide à prévenir les échecs en cascade si l'un des services échoue. Pour obtenir un exemple, voir la bibliothèque Hystrix de Netflix.
- Utilisez des conteneurs pour limiter les ressources disponibles à un sous-système particulier.



Appliquer des transactions de compensation

Une transaction de compensation permet d'annuler les effets d'une autre transaction terminée.

Dans un système distribué, il peut être très difficile d'atteindre une cohérence transactionnelle solide. Les transactions de compensation sont un moyen d'atteindre la cohérence à l'aide d'un ensemble d'opérations plus petites, individuelles qui peuvent être annulées à chaque étape.

Par exemple, pour réserver un voyage, un client peut réserver une voiture, une chambre d'hôtel et un vol. Si l'une de ces étapes échoue, l'ensemble de l'opération échoue. Au lieu d'essayer d'utiliser une transaction distribuée unique pour l'ensemble de l'opération, vous pouvez définir une transaction de compensation pour chaque étape. Par exemple, pour annuler une réservation de voiture, vous annulez la réservation. Afin de compléter l'ensemble de l'opération, un coordinateur exécute chaque étape. Si une étape échoue, le coordinateur applique des transactions de compensation pour annuler toute étape qui a été réalisée.

Pour plus d'informations, consultez Patron Compensating Transaction (Transaction de compensation).

Essais de résilience

En règle générale, vous ne pouvez pas tester la résilience de la même manière que vous testez la fonctionnalité de l'application (en exécutant des tests unitaires et ainsi de suite). Vous devez au lieu de cela tester les performances de la charge de travail de bout en bout dans des conditions d'échec qui se produisent uniquement par intermittence.

Les essais sont un processus itératif. Testez l'application, mesurez les résultats, analysez et gérez tous les échecs qui en résultent, puis répétez le processus.

Essais d'injection d'erreurs. Testez la résilience du système pendant les échecs, soit en déclenchant des échecs réels, soit en les simulant. Voici quelques scénarios courants d'échecs à tester :

- Fermez les instances de machine virtuelle.
- Bloquez des processus.
- Faites expirer des certificats.
- Modifiez les touches d'accès rapide.
- Arrêtez le service DNS sur les contrôleurs de domaine.
- Limitez les ressources système disponibles, telles que la RAM ou le nombre de fils de discussion.
- Démontez les disques.
- Redéployez une machine virtuelle.

Mesurez les temps de récupération et vérifiez que vos exigences professionnelles sont respectées. Testez également des combinaisons de modes d'échec. Assurez-vous que les échecs ne s'affichent pas en cascade et sont gérés de façon isolée.

Il y a une autre raison justifiant l'importance de l'analyse des points d'échec possibles durant la phase de conception. Les résultats de cette analyse devraient être entrés dans votre plan de test.

Test de charge. Effectuez un test de charge de l'application à l'aide d'un outil tel que Visual Studio Team Services ou Apache JMeter. Le test de charge est essentiel pour l'identification des échecs qui ne se produisent que sous la charge, comme une surcharge de la base de données principale ou une limitation de service. Testez la charge de pointe, à l'aide de données de production ou synthétiques qui soient le plus près possible des données production. L'objectif est de voir comment l'application se comporte dans des conditions réelles.

Déploiement durable

Une fois qu'une application est déployée en production, les mises à jour sont une source d'erreurs possible. Dans le pire des cas, une mauvaise mise à jour peut provoquer des temps d'arrêt. Pour éviter cela, le processus de déploiement doit être prévisible et renouvelable. Le déploiement inclut la fourniture de ressources Azure, le déploiement de code d'application et l'application de paramètres de configuration. Une mise à jour peut impliquer les trois, ou un sous-ensemble.

Le point crucial est que les déploiements manuels sont sujets à l'erreur. Par conséquent, il est recommandé d'avoir un processus idempotent automatisé, que vous pouvez exécuter à la demande et réexécuter en cas d'échec.

- Utilisez des modèles Resource Manager pour automatiser la mise en service de ressources Azure.
- Utilisez Configuration de l'état souhaité (DSC) Azure Automation pour configurer des machines virtuelles.
- Utilisez un processus de déploiement automatisé pour le code d'application.

Deux concepts liés au déploiement durable sont une infrastructure comme code et une *infrastructure non modifiable*.

- **L'infrastructure en tant que code** est la pratique consistant à utiliser du code pour fournir et configurer l'infrastructure. L'infrastructure en tant que code peut utiliser une approche déclarative ou une démarche impérative (ou une combinaison des deux). Les modèles Resource Manager sont un exemple d'une approche déclarative. Les scripts PowerShell sont un exemple d'une approche impérative.
- Une **infrastructure non modifiable** est le principe selon lequel vous ne devez pas modifier l'infrastructure après qu'elle est déployée en production. Sinon, vous pouvez aboutir à un état où des modifications ad hoc ont été appliquées, et il est donc difficile de savoir exactement ce qui a changé et compliqué de raisonner sur le système.

Une autre question est de savoir comment déployer une mise à jour de l'application. Nous recommandons des techniques telles que le déploiement bleu-vert ou des mises en circulation du contrôle de validité, qui envoie des mises à jour d'une manière très contrôlée afin de minimiser les impacts possibles d'un mauvais déploiement.

- Le déploiement bleu-vert est une technique impliquant qu'une mise à jour soit déployée dans un environnement de production distinct de l'application en temps réel. Après avoir validé le déploiement, basculez l'acheminement du trafic vers la version mise à jour. Par exemple, Azure App Service Web Apps permet cela avec des emplacements intermédiaires.
- Les mises en circulation du contrôle de validité ressemblent aux déploiements bleu-vert. Au lieu de passer tout le trafic vers la version mise à jour, vous déployez la mise à jour sur un petit pourcentage d'utilisateurs, en acheminant une partie du trafic vers le nouveau déploiement. S'il y a un problème, interrompez et revenez à l'ancien déploiement. Dans le cas contraire, acheminez davantage de trafic vers la nouvelle version, jusqu'à ce qu'à atteindre 100 % du trafic.

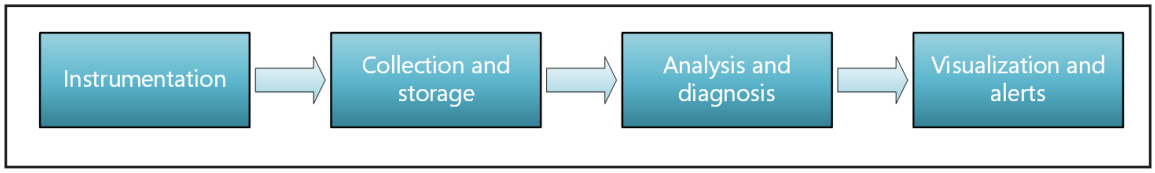
Quelle que soit l'approche que vous adoptez, assurez-vous de pouvoir restaurer le dernier déploiement correct connu, dans le cas où la nouvelle version ne fonctionnerait pas. De même, en cas d'erreur, les journaux des applications doivent indiquer quelle version a provoqué l'erreur.

Surveillance et diagnostics

La surveillance et le diagnostic sont essentiels pour la résilience. En cas d'échec, vous devez en avoir connaissance, et vous avez besoin de comprendre la cause de l'échec.

La surveillance d'un système distribué à grande échelle pose un défi important. Pensez à une application qui s'exécute sur quelques douzaines de machines virtuelles. Il n'est pas pratique de vous connecter à chaque machine virtuelle, une à la fois, pour consulter les fichiers journaux afin d'essayer de résoudre un problème. En outre, le nombre d'instances de machine virtuelle n'est probablement pas statique. Les machines virtuelles sont ajoutées et supprimées à mesure que la taille de l'application diminue et augmente et, occasionnellement, une instance peut échouer et avoir besoin d'être réapprovisionnée. De plus, une application cloud typique peut utiliser plusieurs magasins de données (stockage Azure, SQL Database, Cosmos DB, cache Redis), et une action utilisateur unique peut couvrir plusieurs sous-systèmes.

Vous pouvez considérer le processus de surveillance et de diagnostic comme un pipeline avec plusieurs étapes distinctes :



- **Instrumentation.** Les données brutes pour la surveillance et le diagnostic proviennent de diverses sources, y compris des journaux d'application, des journaux de serveur web, des compteurs de performances de système d'exploitation, des journaux de base de données et des diagnostics intégrés à la plateforme Azure. La plupart des services Azure ont une fonction de diagnostic que vous pouvez utiliser pour déterminer la cause des problèmes.
- **Collecte et stockage.** Des données d'instrumentation brutes peuvent être conservées dans divers emplacements et sous différents formats (par exemple, des journaux de suivi d'application, des journaux IIS, des compteurs de performances). Ces sources disparates sont collectées, consolidées et placées dans un stockage fiable.
- **Analyse et diagnostic.** Après que les données sont consolidées, elles peuvent être analysées afin de résoudre des problèmes et de fournir une vision globale de l'intégrité de l'application.
- **Visualisation et alertes.** À cette étape, les données de télémétrie sont présentées de telle sorte qu'un opérateur peut rapidement remarquer des problèmes ou des tendances. Des exemples incluent des tableaux de bord ou des alertes par e-mail.

La surveillance n'est pas la même que la détection des échecs. Par exemple, votre application peut détecter une erreur temporaire et réessayer, ce qui n'entraîne aucun temps d'arrêt. Mais elle doit aussi consigner l'opération de nouvelle tentative, afin que vous puissiez surveiller le taux d'erreur, en vue d'obtenir une image globale de l'intégrité de l'application.

Les journaux des applications sont une source importante de données de diagnostic. Les meilleures pratiques pour la journalisation de l'application incluent les suivantes :

- Ouvrez une session dans la production. Sinon, vous perdez l'aperçu là où vous en avez le plus besoin.
- Journalisez les événements aux limites de service. Incluez un ID de corrélation qui va au-delà des limites de service. Si une transaction traverse plusieurs services et que l'un d'eux échoue, l'ID de corrélation vous aidera à identifier pourquoi la transaction a échoué.
- Utilisez une journalisation sémantique, aussi connue comme journalisation structurée. Avec des journaux non structurées, il est difficile d'automatiser la consommation et l'analyse des données du journal, qui sont nécessaires à l'échelle du cloud.
- Utilisez la journalisation asynchrone. Dans le cas contraire, le système de journalisation lui-même peut entraîner l'échec de l'application en provoquant la sauvegarde de demandes, car elles se bloquent en attendant d'écrire un événement de journalisation.
- La journalisation de l'application est différente de l'audit. L'audit peut être effectué pour des raisons réglementaires ou de conformité. Ainsi, les enregistrements d'audit doivent être complets, et il n'est pas acceptable d'en laisser tomber lors du traitement des transactions. Si une application a besoin d'un audit, cela doit être séparé de la journalisation des diagnostics.

Pour plus d'informations sur la surveillance et les diagnostics, voir Aide à la surveillance et aux diagnostics.

Réponses de défaillance manuelles

Les sections précédentes ont mis l'accent sur des stratégies de récupération automatique, qui sont essentielles pour une haute disponibilité. Cependant, une intervention manuelle est parfois nécessaire.

- **Alertes.** Surveillez tout signe d'avertissement de votre application pouvant nécessiter une intervention proactive. Par exemple, si vous voyez que SQL Database ou Cosmos DB limite constamment votre application, vous devez peut-être augmenter votre capacité de base de données ou optimiser vos requêtes. Dans cet exemple, même si l'application peut gérer des erreurs de limitation de manière transparente, votre télémétrie doit tout de même déclencher une alerte afin que vous puissiez assurer le suivi.

- **Basculement manuel.** Certains systèmes ne peuvent pas basculer automatiquement et ont besoin d'un basculement manuel.

- **Test de préparation opérationnelle.** Si votre application bascule vers une région secondaire, vous devez effectuer un test de préparation opérationnelle avant la restauration automatique sur la région principale. Ce test doit vérifier que la région principale est intègre et prête à recevoir à nouveau du trafic.

- **Vérification de la cohérence des données.** Si un échec se produit dans un magasin de données, il peut y avoir incohérences de données quand le magasin redevient disponible, surtout si les données ont été répliquées.

- **Restauration à partir d'une sauvegarde.** Par exemple, si SQL Database subit une panne régionale, vous pouvez géo-restaurer la base de données depuis la dernière sauvegarde.

Documentez et testez votre plan de récupération d'urgence. Évaluez l'impact commercial des échecs d'application. Automatisez le processus autant que possible et documentez toutes les étapes manuelles, telles que le basculement manuel ou la restauration de données à partir de sauvegardes. Testez régulièrement votre processus de récupération d'urgence pour valider et améliorer le plan.

Résumé

Cet article aborde la résilience d'un point de vue global, en mettant l'accent sur certains des défis uniques du cloud. Ces derniers incluent la nature distribuée du cloud computing, l'utilisation du matériel de base et la présence d'erreurs réseau temporaires.

Voici les principaux points à retirer de cet article :

- La résilience améliore la disponibilité et diminue les temps moyens de récupération après des échecs.
- L'obtention de la résilience dans le cloud requiert un ensemble de techniques différent des solutions locales traditionnelles.
- La résilience ne se produit pas par accident. Elle doit être conçue et intégrée dès le début.
- La résilience touche toutes les parties du cycle de vie de l'application, depuis la planification et le codage jusqu'aux opérations.
- Testez et surveillez.

Conception de votre application Azure : utilisez ces piliers de qualité

Évolutivité, disponibilité, résilience, gestion et sécurité sont les cinq piliers de logiciels de qualité. Vous concentrer sur ces piliers vous aidera à concevoir une application cloud réussie. Vous pouvez utiliser les listes de contrôle dans ce guide pour examiner votre application au regard de ces piliers.

Pilier	Description
Évolutivité	Concevez le système afin qu'il puisse monter en charge en ajoutant de nouvelles instances.
Disponibilité	Définissez un objectif de niveau de service (SLO) définissant clairement la disponibilité prévue, et comment elle est mesurée. Utilisez le chemin critique pour définir. Sur une année, l'accumulation de points de pourcentage supplémentaires de disponibilité peut offrir des heures ou des jours de durée de fonctionnement supplémentaires.
Résilience	Les applications cloud connaissent parfois des échecs et doivent être construites afin de pouvoir récupérer suite à ces derniers. Construisez des atténuations de résilience dans votre application à tous les niveaux. Concentrez-vous d'abord sur des atténuations tactiques et incluez la surveillance.
Gestion	Automatisez les déploiements et faites-en un processus systématique et rapide afin d'accélérer le déblocage de nouvelles fonctions ou corrections de bogues. Développez des opérations de restauration ou restauration par progression et incluez la surveillance et les diagnostics.
Sécurité	Développez une gestion de l'identité, une protection des infrastructures ainsi qu'une souveraineté et un chiffrement des données dans votre application et dans vos processus DevOps.

Évolutivité

L'évolutivité est la capacité d'un système à gérer la charge accrue. Il existe deux façons principales pour qu'une application s'adapte. La montée en charge verticale (montée en puissance) signifie augmenter la capacité d'une ressource, par exemple en utilisant une machine virtuelle de plus grande taille. La mise à l'échelle horizontale (montée en charge) qui consiste à ajouter de nouvelles instances d'une ressource, telles que des machines virtuelles ou des réplicas de base de données.

La mise à l'échelle horizontale présente de nets avantages par rapport à la montée en charge verticale :

- Échelle cloud véritable. Les applications peuvent être conçues pour fonctionner sur des centaines, voire des milliers de nœuds, atteignant des échelles qui ne sont pas possibles sur un seul nœud.
- La mise à l'échelle horizontale est élastique. Vous pouvez ajouter des instances si la charge augmente, ou les retirer pendant les périodes plus calmes.
- La montée en charge peut être déclenchée automatiquement, soit selon un calendrier, soit en réponse aux changements de charge.
- La montée en charge peut être moins chère que la montée en puissance. L'exécution de plusieurs petites machines virtuelles peut coûter moins cher qu'une grande machine virtuelle unique.
- La mise à l'échelle horizontale peut également améliorer la résilience, en ajoutant la redondance. Si une instance tombe en panne, l'application continue à fonctionner.

Un avantage de la montée en charge verticale est que vous pouvez y procéder sans apporter de modifications à l'application. Mais à un moment donné, vous atteindrez une limite, et vous ne pourrez plus augmenter la charge. À ce moment-là, toute mise à l'échelle supplémentaire devra être horizontale.

La mise à l'échelle horizontale doit être conçue dans le système. Par exemple, vous pouvez monter en charge des machines virtuelles en les plaçant derrière un équilibreur de charge. Mais chaque machine virtuelle dans le pool doit être en mesure de gérer toute demande du client. L'application doit donc être sans état ou stocker un état en externe (par exemple, dans un cache distribué). Une mise à l'échelle horizontale et une mise à l'échelle automatique sont souvent intégrées dans les services PaaS gérés. La facilité de mise à l'échelle de ces services est un atout majeur de l'utilisation des services PaaS.

Toutefois, le seul ajout d'instances supplémentaires ne signifie pas qu'une application va se mettre à l'échelle. Elle pourrait simplement déplacer le goulot d'étranglement. Par exemple, si vous mettez à l'échelle un serveur web frontal pour traiter plusieurs demandes des clients, cela pourraient déclencher des conflits de verrou dans la base de données. Vous seriez alors tenu d'envisager des mesures supplémentaires, telles que l'accès concurrentiel optimiste ou le partitionnement de données, pour permettre un plus gros débit vers la base de données.

Effectuez toujours des tests de performances et de charge pour trouver ces goulots d'étranglement potentiels. Les éléments avec état d'un système, tels que les bases de données, sont les causes les plus courantes des goulots d'étranglement et nécessitent une conception rigoureuse pour une mise à l'échelle horizontale. La résolution d'un goulot d'étranglement peut révéler d'autres goulots d'étranglement ailleurs.

Utilisez la liste de contrôle d'évolutivité pour revoir votre conception du point de vue de l'évolutivité.

Aide à l'évolutivité

- Patrons de conception pour l'évolutivité et les performances
- Meilleures pratiques : mise à l'échelle automatique, tâches en arrière-plan, mise en cache, CDN, partitionnement de données

Meilleures pratiques

- https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/background-jobs
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/cdn
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning

Disponibilité

La disponibilité est la proportion de temps durant laquelle le système est fonctionnel et en fonctionnement. Elle est habituellement mesurée en pourcentage de durée de fonctionnement. Des erreurs d'application, des problèmes d'infrastructure et la charge du système sont autant de facteurs susceptibles de réduire la disponibilité.

Une application cloud devrait avoir un objectif de niveau de service (SLO) qui définit clairement la disponibilité prévue et la façon dont la disponibilité est mesurée. Lorsque vous définissez la disponibilité, regardez le chemin critique. Le serveur web frontal pourrait être en mesure de traiter des demandes de client, mais si chaque transaction échoue parce qu'elle ne peut pas se connecter à la base de données, l'application n'est pas disponible pour les utilisateurs.

La disponibilité est souvent décrite en termes de « 9 » — par exemple, « quatre 9 » signifie 99,99 % de durée de fonctionnement. Le tableau suivant indique les temps d'arrêt cumulatifs potentiels à différents niveaux de disponibilité.

SLAD	Temps d'arrêt par semaine	Temps d'arrêt par mois	Temps d'arrêt par an
99%	1,68 heure	7,2 heures	365 jours
99.9%	10,1 minutes	43,2 minutes	8,76 heures
99.95%	5 minutes	21,6 minutes	4,38 heures
99.99%	1,01 minute	4,32 minutes	52,56 minutes
99.999%	6 secondes	25,9 secondes	5,26 minutes

Notez que 99 % de durée de fonctionnement pourrait se traduire par une panne de service de presque 2 heures par semaine. Pour de nombreuses applications, notamment les applications orientées consommateur, cet SLO n'est pas acceptable. En revanche, cinq 9 (99,999 %) signifient un maximum de 5 minutes de temps d'arrêt par an. Il est suffisamment compliqué de détecter une panne aussi rapidement, sans parler de résoudre le problème. Pour obtenir une très haute disponibilité (99,99 % ou plus), vous ne pouvez pas compter sur une intervention manuelle pour une récupération après des échecs. L'application doit procéder à un auto-diagnostic et une réparation spontanée, et c'est là que la résilience devient cruciale.

Aide à l'évolutivité

- Patrons de conception pour la disponibilité

Meilleures pratiques

- https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/background-jobs

Résilience

La résilience est la capacité du système à récupérer après des échecs pour continuer à fonctionner. La résilience vise à permettre à l'application de refonctionner pleinement suite à un échec. La résilience est étroitement liée à la disponibilité.

Dans un développement d'applications traditionnel, l'accent a été mis sur la réduction du délai moyen entre les échecs (MTBF). Des efforts ont été mis en place pour empêcher un échec du système. En raison de plusieurs facteurs, le cloud computing implique une mentalité différente :

- Les systèmes distribués sont complexes, et un échec à un moment donné peut potentiellement s'afficher en cascade dans tout le système.
- Les coûts pour les environnements cloud sont maîtrisés grâce à l'utilisation de matériel de base. Des défaillances matérielles occasionnelles sont donc à envisager.
- Les applications dépendent souvent de services externes, qui peuvent devenir temporairement indisponibles ou limiter les utilisateurs de volumes élevés.
- Les utilisateurs d'aujourd'hui s'attendent à ce qu'une application soit disponible 24h/24, 7 j/7, sans jamais être déconnectée.

Tous ces facteurs signifient que les applications cloud doivent être conçues pour s'attendre à des échecs occasionnels et à assurer la récupération suite à ces derniers. Azur possède de nombreuses fonctionnalités de résilience déjà intégrées dans la plateforme. Par exemple,

- Le stockage Azure, SQL Database et Cosmos DB fournissent tous une réplication de données intégrée, à la fois au sein d'une région et entre les régions.
- Afin de limiter les effets des défaillances matérielles, les Azure Managed Disks sont automatiquement placés dans différentes unités d'échelle de stockage.
- Les machines virtuelles dans un groupe à haute disponibilité sont réparties sur plusieurs domaines d'erreur. Un domaine d'erreur est un groupe de machines virtuelles qui partagent une source d'alimentation et un commutateur réseau communs. La propagation de machines virtuelles sur plusieurs domaines d'erreur limite l'impact des défaillances du matériel physique, des indisponibilités du réseau ou des coupures de courant.

Cela dit, vous avez encore besoin de développer la résilience de votre application. Des stratégies de résilience peuvent être appliquées à tous les niveaux de l'architecture. Certaines atténuations sont plus tactiques par nature, par exemple, une nouvelle tentative d'appel distant après un échec de réseau passager. D'autres atténuations sont plus stratégiques, comme le basculement de l'ensemble de l'application sur une région secondaire. Les atténuations tactiques peuvent faire une grande différence. S'il est rare qu'une région tout entière connaisse une coupure, des problèmes temporaires tels que la surcharge du réseau sont plus courants, c'est pourquoi il est intéressant de les cibler en premier. Il est également important de disposer de la bonne surveillance et des bons diagnostics, aussi bien pour détecter les pannes lorsqu'elles se produisent que pour trouver les causes racines.

Lorsque vous concevez une application pour qu'elle soit résiliente, vous devez comprendre vos besoins de disponibilité. Combien de temps d'arrêt est acceptable ? Cela dépend en partie du coût. Combien l'arrêt potentiel coûtera-t-il à votre entreprise ? Combien devriez-vous investir pour rendre l'application hautement disponible ?

Utilisez la liste de contrôle de résilience pour revoir votre conception du point de vue de la résilience.

Aide à la résilience

- Pour plus d'informations sur la conception d'applications durables pour Azure, consultez https://docs.microsoft.com/en-us/azure/architecture/resiliency/index.
- Patrons de conception pour la résilience
- Meilleures pratiques : gestion des erreurs temporaires, aide à la nouvelle tentative pour des services spécifiques

Meilleures pratiques

- https://docs.microsoft.com/en-us/azure/architecture/best-practices/transient-faults
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/retry-service-specific

Gestion et DevOps

Ce pilier couvre les processus d'opérations qui permettent à une application de s'exécuter en production.
Les déploiements doivent être fiables et prévisibles. Ils doivent être automatisés pour réduire le risque d'erreur humaine. Le processus doit être rapide et systématique afin de ne pas ralentir le déblocage de nouvelles fonctions ou corrections de bogues. Tout aussi important, vous devez être en mesure de réaliser rapidement des opérations de restauration ou de restauration par progression si une mise à jour rencontre des problèmes.

La surveillance et les diagnostics sont essentiels. Pour les meilleures pratiques en termes de surveillance et de diagnostics, consultez https://docs.microsoft.com/en-us/azure/architecture/best-practices/monitoring. Les applications cloud s'exécutent dans un centre de données distant où vous n'avez pas le contrôle total de l'infrastructure ou, dans certains cas, du système d'exploitation. Dans une application volumineuse, il n'est pas pratique de se connecter à des machines virtuelles pour résoudre un problème ou de passer au crible les fichiers journaux. Avec les services PaaS, il se peut qu'il n'y ait même pas une machine virtuelle dédiée à laquelle se connecter. La surveillance et les diagnostics donnent un aperçu du système, afin que vous sachiez quand et où les défaillances se produisent. Tous les systèmes doivent être observables. Utilisez un schéma de journalisation commun et cohérent qui vous permet de mettre en corrélation des événements entre les systèmes.

Le processus de surveillance et de diagnostics comprend plusieurs phases distinctes :

- Instrumentation. Génération des données brutes à partir de journaux d'application, de journaux de serveur web, de diagnostics intégrés dans plateforme Azure et d'autres sources.
- Collecte et stockage. Consolidation des données en un seul emplacement.
- Analyse et diagnostic. Pour résoudre les problèmes et voir l'intégrité générale.
- Visualisation et alertes. Utilisation de données de télémétrie pour repérer des tendances ou prévenir l'équipe des opérations.

Utilisez la liste de contrôle DevOps pour analyser votre conception d'un point de vue de DevOps et de gestion.

Aide à la gestion et à DevOps

- Patrons de conception pour la gestion et la surveillance
- Meilleures pratiques : surveillance et diagnostics

Sécurité

Vous devez penser sécurité tout au long du cycle de vie complet d'une application, de la conception et la mise en œuvre au déploiement et aux opérations. La plateforme Azure fournit des protections contre diverses menaces, telles que l'intrusion réseau et des attaques DDoS. Mais vous avez encore besoin de développer la sécurité dans votre application et dans vos processus DevOps.

Voici quelques domaines généraux de sécurité à envisager.

Gestion des identités

Envisagez d'utiliser Azure Active Directory (Azure AD) pour authentifier et autoriser des utilisateurs. Azure AD est un service entièrement géré de gestion des identités et des accès. Vous pouvez l'utiliser pour créer des domaines qui existent uniquement sur Azure, ou pour une intégration à vos identités Active Directory locales. Azure AD s'intègre également avec Office365, Dynamics CRM Online et de nombreuses applications SaaS tierces. Pour les applications orientées consommateur, Azure Active Directory B2C permet aux utilisateurs de s'authentifier avec leurs comptes de réseaux sociaux existants (comme Facebook, Google ou LinkedIn), ou de créer un nouveau compte utilisateur qui est géré par Azure AD.

Si vous souhaitez intégrer un environnement Active Directory local avec un réseau Azure, plusieurs approches sont possibles, en fonction de vos exigences. Pour de plus amples informations, consultez nos architectures de référence pour la gestion des identités.

Protection de votre infrastructure

Contrôlez l'accès aux ressources Azure que vous déployez. Chaque abonnement Azure a une relation d'approbation avec un locataire Azure AD. Utilisez un contrôle d'accès en fonction du rôle (RBAC) pour accorder aux utilisateurs au sein de votre organisation les autorisations appropriées aux ressources Azure. Accordez l'accès en attribuant le rôle RBAC aux utilisateurs ou aux groupes dans une certaine étendue. L'étendue peut être un abonnement, un groupe de ressources ou une ressource unique. Vérifiez toutes les modifications apportées à l'infrastructure. Pour plus d'informations, consultez https://docs.microsoft.com/en-us/azure/active- directory/.

Sécurité des applications

En général, les meilleures pratiques de sécurité pour le développement d'applications s'appliquent toujours dans le cloud. Ces dernières comprennent des éléments tels que l'utilisation de SSL partout, la protection contre les attaques CSRF et XSS, la prévention des attaques par injection de code SQL et ainsi de suite.

Des applications cloud utilisent souvent des services gérés disposant de touches d'accès rapide. Ne les sélectionnez jamais dans le contrôle de code source. Envisagez de stocker les secrets de l'application dans Azure Key Vault.

Chiffrement et souveraineté des données

Assurez-vous que vos données demeurent dans la zone géopolitique correcte lors de l'utilisation d'Azure à haut niveau de disponibilité. Le stockage géorépliqué d'Azure utilise le concept d'une région appariée dans la même région géopolitique.

Utilisez Key Vault pour protéger les secrets et les clés de chiffrement. En utilisant Key Vault, vous pouvez chiffrer les clés et les secrets en utilisant les clés qui sont protégées par des modules de sécurité matériels (HSM). De nombreux services de stockage Azure et DB Services prennent en charge le chiffrement des données au repos, y compris le stockage Azure, Azure SQL Database, Azure SQL Data Warehouse et Cosmos DB.

Pour en savoir plus, consultez :

- https://docs.microsoft.com/en-us/azure/storage/storage-service-encryption
- https://docs.microsoft.com/en-us/azure/sql-database/sql-database-always-encrypted-azure-key-vault
- https://docs.microsoft.com/en-us/azure/data-lake-store/data-lake-store-security-overview#data-protection
- https://docs.microsoft.com/en-us/azure/cosmos-db/database-security

Ressources de sécurité

- Azure Security Center {aucun lien} fournit la surveillance de sécurité intégrée et la gestion de stratégie sur l'ensemble de vos abonnements Azure. Accédez à https://azure.microsoft.com/en-us/services/security-center/.
- Pour plus d'informations sur la façon de protéger vos applications dans le cloud, consultez https://docs.microsoft.com/en-us/azure/security/.

Conception de votre application Azure : patrons de conception

Ces patrons de conception sont utiles pour la création d'applications fiables, évolutives et sécurisées dans le cloud.

Chaque patron décrit le problème que le patron traite, des considérations relatives à l'application du patron et un exemple basé sur Microsoft Azure. La plupart des patrons incluent des exemples ou des extraits de code qui montrent comment implémenter le patron sur Azure. Cependant, la plupart des patrons sont pertinents sur n'importe quel système distribué, qu'il soit hébergé sur Azure ou sur d'autres plateformes cloud.

Défis de développement cloud

Disponibilité

La disponibilité définit la proportion de temps durant laquelle le système est fonctionnel et en fonctionnement. Elle sera affectée par des erreurs système, des problèmes d'infrastructure, des attaques malveillantes et la charge du système. Elle est habituellement mesurée en pourcentage de durée de fonctionnement. Les applications cloud fournissent généralement aux utilisateurs un contrat de niveau de service (SLA), ce qui signifie que ces applications doivent être conçues et implémentées d'une manière qui maximise la disponibilité.

Patron	Résumé
Surveillance Health Endpoint	Implémentez des contrôles fonctionnels dans une application auxquels des outils externes ont accès via des points de terminaison exposés à intervalles réguliers.
Nivellement de charge basé sur la file d'attente	Utilisez une file d'attente qui agit comme un tampon entre une tâche et un service qu'elle appelle pour lisser les charges lourdes intermittentes.
Limitation	Contrôlez la consommation de ressources utilisée par une instance d'une application, un locataire individuel ou un service complet.

Gestion des données

La gestion des données est l'élément clé des applications cloud et influe sur la plupart des attributs de qualité. Les données sont généralement hébergées dans différents emplacements et sur plusieurs serveurs pour des raisons telles que la performance, l'évolutivité ou la disponibilité, et cela peut présenter un éventail de défis. Par exemple, la cohérence des données doit être maintenue, et les données devront généralement être synchronisées sur différents emplacements.

Patron	Résumé
Cache-Aside	Chargez des données à la demande dans un cache à partir d'un magasin de données
CQRS	Séparez les opérations qui lisent des données depuis des opérations mettant à jour des données à l'aide d'interfaces distinctes.
Approvisionnement d'événements	Utilisez un magasin ajout uniquement pour enregistrer la série complète des événements qui décrivent des actions effectuées sur des données d'un domaine.
Table d'index	Créez des index sur les champs dans des magasins de données qui sont souvent référencés par requêtes.
Vue matérialisée	Générez des vues préremplies sur les données dans un ou plusieurs magasins de données lorsque les données ne sont pas idéalement formatées pour des opérations de requête requises.
Partitionnement	Divisez un magasin de données en un ensemble de partitions horizontales ou de partitions.
Hébergement de contenu statique	Déployez du contenu statique dans un service de stockage basé sur le cloud qui peut les fournir directement au client.
Clé à accès restreint	Utilisez un jeton ou une clé qui fournit aux clients un accès direct restreint à un service ou une ressource spécifique.

Conception et mise en œuvre

Une bonne conception englobe des facteurs tels que la cohérence dans le déploiement et la conception des composants, la maintenabilité pour simplifier l'administration et le développement, ainsi que la réutilisabilité pour permettre l'utilisation de composants et de sous-systèmes dans d'autres applications et dans d'autres scénarios. Les décisions prises au cours de la phase de conception et d'implémentation ont un impact énorme sur la qualité et le coût total de possession des applications et des services hébergés dans le cloud.

Patron	Résumé
Ambassadeur	Créez des services d'assistance qui envoient des requêtes réseau au nom d'une application ou d'un service consommateur.
Couche anticorruption	Implémentez une couche de façade ou d'adaptateur entre une application moderne et un système hérité.
Serveurs principaux pour serveurs frontaux	Créez des services de serveur principal distincts devant être consommés par des applications ou des interfaces de serveur frontal spécifiques.

CQRS	Séparez les opérations qui lisent des données depuis des opérations mettant à jour des données à l'aide d'interfaces distinctes.
Consolidation de la ressource de calcul	Consolidez plusieurs tâches ou opérations dans une seule unité de calcul.
Magasin de configuration externe	Extrayez les informations de configuration du package de déploiement de l'application vers un emplacement centralisé.
Agrégation de passerelle	Utilisez une passerelle pour agréger plusieurs requêtes individuelles en une seule requête.
Déchargement de passerelle	Déchargez la fonctionnalité de service partagé ou spécialisé vers un proxy de passerelle.
Routage de passerelle	Acheminez les requêtes vers plusieurs services à l'aide d'un point de terminaison unique.
Élection de leader	Coordonnez les actions effectuées par un ensemble d'instances de tâches de collaboration dans une application distribuée en choisissant une instance comme le leader qui endosse la responsabilité de la gestion des autres instances.
Pipes and Filters (Filtres et tubes)	Répartissez une tâche qui effectue un traitement complexe en une série d'éléments distincts qui peuvent être réutilisés.
Sidecar	Déployez des composants d'une application dans un conteneur ou un processus distinct pour fournir l'isolation et l'encapsulation.
Hébergement de contenu statique	Déployez du contenu statique dans un service de stockage basé sur le cloud qui peut les fournir directement au client.
Étrangleur	Migrez progressivement un système hérité en remplaçant progressivement des éléments spécifiques de fonctionnalité par de nouveaux services et applications.

Messagerie

La nature distribuée des applications cloud nécessite une infrastructure de messagerie qui relie les composants et les services, idéalement d'une manière faiblement couplée afin d'optimiser l'évolutivité. La messagerie asynchrone est couramment employée et offre de nombreux avantages, mais apporte également des défis tels que le classement des messages, la gestion des messages incohérents, l'idempotence et plus encore.

Patron	Résumé
Consommateurs concurrents	Permettez à plusieurs consommateurs simultanés de traiter des messages reçus sur le même canal de messagerie.
Pipes and Filters (Filtres et tubes)	Répartissez une tâche qui effectue un traitement complexe en une série d'éléments distincts qui peuvent être réutilisés.
File d'attente prioritaire	Classez les requêtes par ordre de priorité envoyées aux services afin que celles avec une priorité plus élevée soient reçues et traitées plus rapidement que celles dont la priorité est plus basse.

Nivellement de charge basé sur la file d'attente	Utilisez une file d'attente qui agit comme un tampon entre une tâche et un service qu'elle appelle pour lisser les charges lourdes intermittentes.
Superviseur d'agent planificateur	Coordonnez un ensemble d'actions sur un ensemble distribué de services et d'autres ressources distantes.

Gestion et surveillance

Les applications cloud s'exécutent dans un centre de données distant où vous n'avez pas le contrôle total de l'infrastructure ou, dans certains cas, du système d'exploitation. Cela peut rendre la gestion et la surveillance plus compliquées que sur un déploiement local. Les applications doivent exposer des informations d'exécution que les administrateurs et les opérateurs peuvent utiliser pour gérer et surveiller le système, ainsi que prendre en charge l'évolution des besoins de l'entreprise et la personnalisation sans exiger l'arrêt ou le redéploiement de l'application.

Patron	Résumé
Ambassadeur	Créez des services d'assistance qui envoient des requêtes réseau au nom d'une application ou d'un service consommateur.
Couche anticorruption	Implémentez une couche de façade ou d'adaptateur entre une application moderne et un système hérité.
Magasin de configuration externe	Extrayez les informations de configuration du package de déploiement de l'application vers un emplacement centralisé.
Agrégation de passerelle	Utilisez une passerelle pour agréger plusieurs requêtes individuelles en une seule requête.
Déchargement de passerelle	Déchargez la fonctionnalité de service partagé ou spécialisé vers un proxy de passerelle.
Routage de passerelle	Acheminez les requêtes vers plusieurs services à l'aide d'un point de terminaison unique.
Surveillance Health Endpoint	Implémentez des contrôles fonctionnels dans une application auxquels des outils externes ont accès via des points de terminaison exposés à intervalles réguliers.
Sidecar	Déployez des composants d'une application dans un conteneur ou un processus distinct pour fournir l'isolation et l'encapsulation.
Étrangleur	Migrez progressivement un système hérité en remplaçant progressivement des éléments spécifiques de fonctionnalité par de nouveaux services et applications.

Performance et évolutivité

La performance est une indication de la réactivité d'un système pour exécuter toute action dans un intervalle de temps donné, même si l'évolutivité est la capacité d'un système à gérer l'augmentation de la charge sans impact sur les performances, ou à permettre un accroissement facile des ressources disponibles. Les applications cloud rencontrent généralement des charges de travail variables et des pics d'activité. La prédiction de ces derniers, en particulier dans un scénario avec plusieurs locataires, est presque impossible. Au lieu de cela, les applications devraient pouvoir monter en charge dans les limites permettant de répondre aux pics de demande et diminuer la taille des instances lorsque la demande chute. Les problèmes d'évolutivité calculent non seulement les instances, mais d'autres éléments tels que le stockage des données, l'infrastructure de messagerie et bien plus.

Patron	Résumé
Cache-Aside	Chargez des données à la demande dans un cache à partir d'un magasin de données
CQRS	Séparez les opérations qui lisent des données depuis des opérations mettant à jour des données à l'aide d'interfaces distinctes.
Approvisionnement d'événements	Utilisez un magasin ajout uniquement pour enregistrer la série complète des événements qui décrivent des actions effectuées sur des données d'un domaine.
Table d'index	Créez des index sur les champs dans des magasins de données qui sont souvent référencés par requêtes.
Vue matérialisée	Générez des vues préremplies sur les données dans un ou plusieurs magasins de données lorsque les données ne sont pas idéalement formatées pour des opérations de requête requises.
File d'attente prioritaire	Classez les requêtes par ordre de priorité envoyées aux services afin que celles avec une priorité plus élevée soient reçues et traitées plus rapidement que celles dont la priorité est plus basse.
Nivellement de charge basé sur la file d'attente	Utilisez une file d'attente qui agit comme un tampon entre une tâche et un service qu'elle appelle pour lisser les charges lourdes intermittentes.
Partitionnement	Divisez un magasin de données en un ensemble de partitions horizontales ou de partitions.
Hébergement de contenu statique	Déployez du contenu statique dans un service de stockage basé sur le cloud qui peut les fournir directement au client.
Limitation	Contrôlez la consommation de ressources utilisée par une instance d'une application, un locataire individuel ou un service complet.

Résilience

La résilience est la capacité d'un système à gérer les échecs, et à assurer la récupération, normalement. La nature de l'hébergement cloud, où des applications sont souvent mutualisées, utilisent des services de plateforme partagée, sont en concurrence pour des ressources et de la bande passante, communiquent sur Internet et s'exécutent sur du matériel de base implique un accroissement du risque d'erreurs temporaires et de davantage d'erreurs permanentes. La détection des échecs, et une récupération rapide et efficace, sont nécessaires pour maintenir la résilience.

Patron	Résumé
Cloison	Isolez les éléments d'une application dans des pools de sorte que si l'un tombe en panne, les autres continueront à fonctionner.
Disjoncteur	Gérez les erreurs dont la correction pourrait avoir une durée variable lors de la connexion à une ressource ou un service distant.
Transaction de compensation	Annulez le travail effectué par une série d'étapes, qui définissent ensemble une opération finalement cohérente.
Surveillance Health Endpoint	Implémentez des contrôles fonctionnels dans une application auxquels des outils externes ont accès via des points de terminaison exposés à intervalles réguliers.
Élection de leader	Coordonnez les actions effectuées par un ensemble d'instances de tâches de collaboration dans une application distribuée en choisissant une instance comme le leader qui endosse la responsabilité de la gestion des autres instances.
Nivellement de charge basé sur la file d'attente	Utilisez une file d'attente qui agit comme un tampon entre une tâche et un service qu'elle appelle pour lisser les charges lourdes intermittentes.
Nouvelle tentative	Permettez à une application de gérer des défaillances temporaires anticipées lorsqu'elle essaie de se connecter à un service ou à une ressource réseau, en réessayant de façon transparente d'effectuer une opération qui a échoué par le passé.
Superviseur d'agent planificateur	Coordonnez un ensemble d'actions sur un ensemble distribué de services et d'autres ressources distantes.

Sécurité

La sécurité est l'aptitude d'un système à empêcher des actions malveillantes ou accidentelles en dehors de l'utilisation prévue et à empêcher la divulgation ou la perte d'informations. Les applications cloud sont exposées sur Internet en dehors des limites locales de confiance sont souvent ouvertes au public et peuvent servir des utilisateurs non approuvés. Les applications doivent être conçues et déployées d'une manière qui les protège contre les attaques malveillantes, qui restreigne l'accès uniquement aux utilisateurs autorisés et qui protège les données sensibles.

Patron	Résumé
Identité fédérée	Déléguez l'authentification à un fournisseur d'identité externe.
Opérateur de contrôle d'appels	Protégez les applications et les services à l'aide d'une instance de l'hôte dédiée qui sert d'intermédiaire entre les clients et l'application ou le service, valide et assainit les requêtes et transmet des requêtes et des données entre eux.
Clé à accès restreint	Utilisez un jeton ou une clé qui fournit aux clients un accès direct restreint à un service ou une ressource spécifique.

Catalogue des patrons

Patron Ambassador (Ambassadeur)

Créez des services d'assistance qui envoient des requêtes réseau au nom d'une application ou d'un service consommateur. Un service ambassadeur peut être considéré comme un proxy hors processus qui est colocalisé avec le client.

Ce patron peut être utile pour le déchargement de tâches courantes de connectivité client tels que la surveillance, la journalisation, le routage, la sécurité (telle que TLS) et les patrons de résilience de manière indépendante du langage. Il est souvent utilisé avec des applications héritées, ou d'autres applications qui sont difficiles à modifier, afin d'étendre leurs capacités de mise en réseau. Il peut également permettre à une équipe spécialisée d'implémenter ces fonctions.

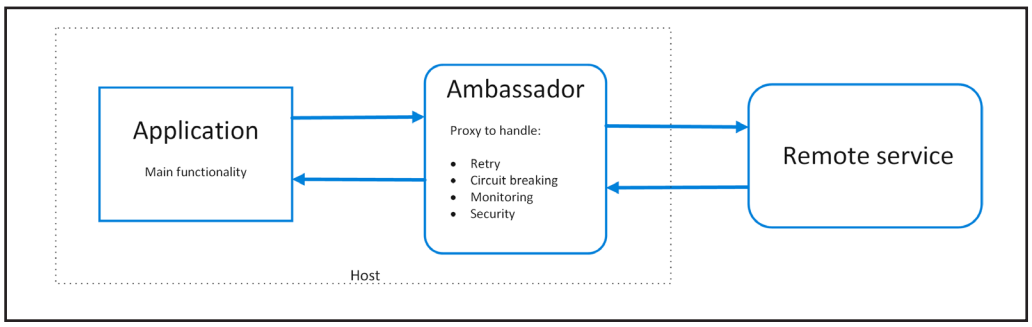
Contexte et problème

Les applications cloud résilientes requièrent des fonctions telles que la rupture du circuit, le routage, le contrôle et la capacité à procéder à des mises à jour de configuration liées au réseau. Il peut être difficile ou impossible de mettre à jour des applications héritées ou des bibliothèques de code existantes pour ajouter ces fonctions car le code n'est plus maintenu, ou ne peut pas être facilement modifié par l'équipe de développement.

Des appels réseau peuvent aussi exiger une configuration importante pour la connexion, l'authentification et l'autorisation. Si ces appels sont utilisés sur plusieurs applications, développées à l'aide de plusieurs langages et infrastructures, les appels doivent être configurés pour chacune de ces instances. En outre, des fonctions de sécurité et de réseau devront peut-être être gérées par une équipe centrale au sein de votre organisation. Avec une base de code large, il peut être risqué pour cette équipe mettre à jour un code d'application qu'elle ne maîtrise pas bien.

Solution

Mettez les infrastructures et bibliothèques clientes dans un processus externe qui sert de proxy entre votre application et des services externes. Déployez le proxy sur le même environnement hôte que votre application pour permettre le contrôle du routage, de la résilience, des fonctions de sécurité et éviter toutes restrictions de l'accès hôte. Vous pouvez également utiliser le patron Ambassador (Ambassadeur) pour standardiser et étendre l'instrumentation. Le proxy peut surveiller les métriques de performance telles que la latence ou l'utilisation des ressources, et cette surveillance se produit dans le même environnement hôte que l'application.



Les fonctions qui sont déchargées à l'ambassadeur peuvent être gérées indépendamment de l'application. Vous pouvez mettre à jour et modifier l'ambassadeur sans perturber la fonctionnalité héritée de l'application. Cela permet également à des équipes distinctes, spécialisées d'implémenter et de maintenir la sécurité, la mise en réseau ou les fonctions d'authentification qui ont été déplacées vers l'ambassadeur.

Les services ambassadeur peuvent être déployés comme un side-car pour accompagner le cycle de vie d'un service ou d'une application de consommation. Par ailleurs, si un ambassadeur est partagé par plusieurs processus distincts sur un hôte commun, il peut être déployé en tant que démon ou service Windows. Si le service de consommation est conteneurisé, l'ambassadeur doit être créé comme conteneur distinct sur le même hôte, avec les liens appropriés configurés pour la communication.

Problèmes et considérations

- Le proxy ajoute une surcharge de latence. Examinez si une bibliothèque cliente, appelée directement par l'application, est une meilleure approche.
- Considérez l'impact possible lié à l'inclusion de fonctions généralisées dans le proxy. Par exemple, l'ambassadeur pourrait gérer des tentatives, mais cela pourrait ne pas être sécurisé à moins que toutes les opérations soient idempotentes.
- Envisagez un mécanisme permettant au client de passer une partie du contexte au proxy, ainsi qu'au client. Par exemple, incluez des en-têtes de requête HTTP pour refuser une nouvelle tentative ou spécifier le nombre maximal de fois de nouvelles tentatives.
- Examinez comment vous allez créer un package et un déploiement du proxy.
- Étudiez si vous utilisez une instance partagée unique pour tous les clients ou une instance pour chaque client.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

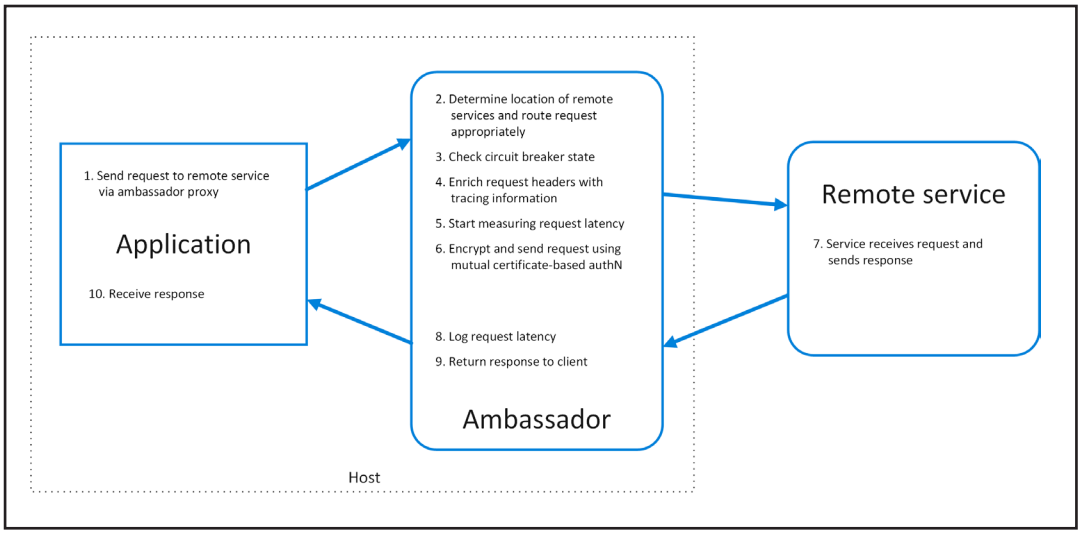
- Nécessité de construire un ensemble commun de fonctions de connectivité client pour plusieurs langages ou infrastructures.
- Nécessité de décharger des préoccupations transversales de connectivité client à des développeurs de l'infrastructure ou d'autres équipes plus spécialisées.
- Nécessité de prendre en charge des exigences de connectivité du cluster ou cloud dans une application héritée ou une application qui est difficile à modifier.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Lorsque la latence de requête réseau est critique. Un proxy présentera une légère surcharge, bien que minime, et dans certains cas, cela peut affecter l'application.
- Les fonctions de connectivité client sont consommées par un seul langage. Dans ce cas, une meilleure option pourrait être une bibliothèque cliente qui est distribuée aux équipes de développement sous forme de package.
- Les fonctions de connectivité ne peuvent pas être généralisées et nécessitent une intégration plus approfondie avec l'application cliente.

Exemple

Le diagramme suivant présente une application soumettant une requête à un service distant via un proxy ambassadeur. L'ambassadeur fournit les éléments suivants : routage, coupure de circuit et journalisation. Il appelle le service distant puis renvoie la réponse à l'application cliente :



Patron Anti-Corruption Layer (niveau de lutte contre la corruption)

Implémentez une couche de façade ou d'adaptateur entre une application moderne et un système hérité dont elle dépend. Cette couche déplace des requêtes entre l'application moderne et le système hérité. Utilisez ce patron pour vérifier que la conception d'une application n'est pas limitée par des dépendances sur des systèmes hérités.

Contexte et problème

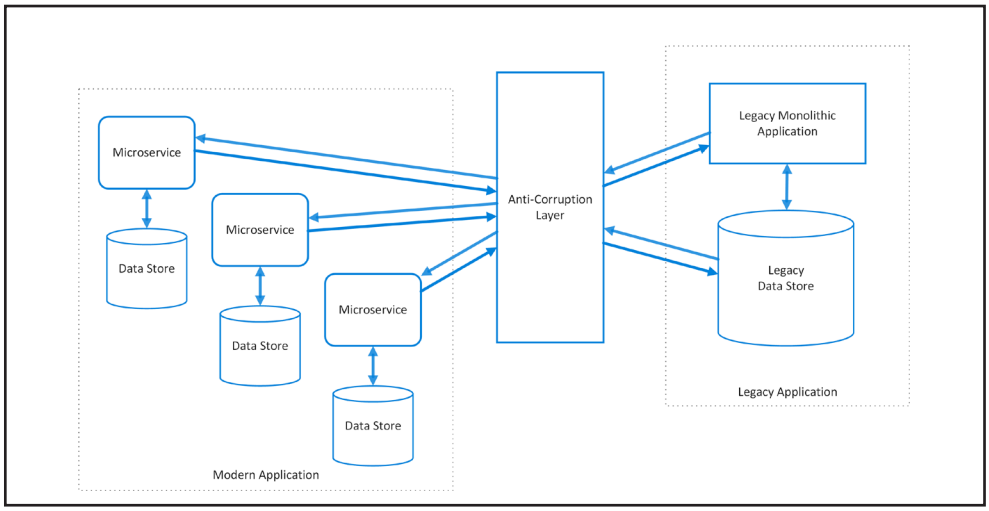
La plupart des applications s'appuient sur d'autres systèmes pour certaines données ou fonctions. Par exemple, lorsqu'une application héritée est migrée vers un système moderne, elle peut avoir encore besoin de ressources héritées existantes. De nouvelles fonctions doivent être en mesure d'appeler le système hérité. Cela est particulièrement vrai pour les migrations progressives, où différentes fonctions d'une application plus grande sont déplacées vers un système moderne au fil du temps.

Ces systèmes hérités souffrent souvent de problèmes de qualité tels que des schémas de données alambiquées ou des API obsolètes. Les fonctions et les technologies utilisées dans les systèmes hérités peuvent varier considérablement par rapport à des systèmes plus modernes. Pour interagir avec le système hérité, la nouvelle application peut devoir prendre en charge des API, des modèles de données, des protocoles, des infrastructures obsolètes ou d'autres fonctions que vous ne placeriez sinon pas dans une application moderne.

Le maintien de l'accès entre les systèmes nouveaux et hérités peut forcer le nouveau système à respecter au moins une partie des API ou d'autres sémantiques du système hérité. Lorsque ces fonctions héritées ont des problèmes de qualité, les prendre en charge « corrompt » ce qui pourrait autrement être une application moderne correctement conçue.

Solution

Isolez les systèmes hérités et modernes en plaçant une couche anticorruption entre eux. Cette couche déplace les communications entre les deux systèmes, permettant au système hérité de demeurer inchangé, tandis que l'application moderne peut éviter de compromettre sa conception et son approche technologique.



La communication entre l'application moderne et la couche anticorruption utilise toujours l'architecture et le modèle de données de l'application. Les appels entre la couche anticorruption et le système hérité sont conformes aux méthodes ou au modèle de données de ce système. La couche anticorruption contient toute la logique nécessaire pour se déplacer entre les deux systèmes. Cette couche peut être implémentée comme un composant dans l'application ou comme un service indépendant.

Problèmes et considérations

- La couche anticorruption peut ajouter de la latence aux appels effectués entre les deux systèmes.
- La couche anticorruption ajoute un service qui doit être géré et tenu à jour.
- Envisagez comment votre couche anticorruption va évoluer.
- Envisagez si vous avez besoin de plus d'une couche anticorruption. Vous voudrez peut-être décomposer la fonctionnalité en plusieurs services à l'aide de différents langages ou technologies, ou il peut y avoir d'autres raisons de partitionner la couche anticorruption.
- Envisagez comment la couche anticorruption sera gérée par rapport à vos autres applications ou services. Comment sera-t-elle intégrée dans votre processus de surveillance, de lancement et de configuration ?
- Assurez-vous que la cohérence des données et la transaction sont tenues à jour et peuvent être surveillées.
- Examinez si la couche anticorruption doit gérer toutes les communications entre les systèmes hérité et moderne, ou seulement un sous-ensemble de fonctions.
- Examinez si la couche anticorruption est destinée à être permanente, ou finalement supprimée après que toutes les fonctions héritées ont été migrées.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Une migration est prévue en plusieurs étapes, mais l'intégration entre les systèmes nouveau et hérité doit être maintenue.
- Les systèmes nouveau et hérité ont différentes sémantiques, mais encore besoin de communiquer.

Ce patron peut ne pas convenir s'il n'y a aucune différence sémantique nette entre les systèmes nouveau et hérité.

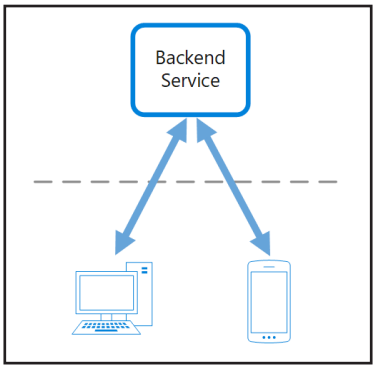
Patron Backends for Frontends (Backends pour Frontends)

Créez des services de serveur principal distincts devant être consommés par des applications ou des interfaces de serveur frontal spécifiques. Ce patron est utile lorsque vous voulez éviter la personnalisation d'un serveur principal unique pour plusieurs interfaces.

Contexte et problème

Une application peut initialement viser l'IU web d'un bureau. En règle générale, un service de serveur principal développé en parallèle fournit les fonctions nécessaires pour cette IU. À mesure que la base d'utilisateurs de l'application se développe, une application mobile est développée et doit interagir avec le même serveur principal. Le service de serveur principal devient un serveur principal polyvalent, répondant aux exigences des deux interfaces (mobile et bureau).

Mais les capacités d'un appareil mobile diffèrent nettement de celle d'un navigateur de bureau, en termes de taille d'écran, de performances et de limites d'affichage. En conséquence, les exigences pour le serveur principal d'une application mobile diffèrent de celles de l'IU web de bureau.

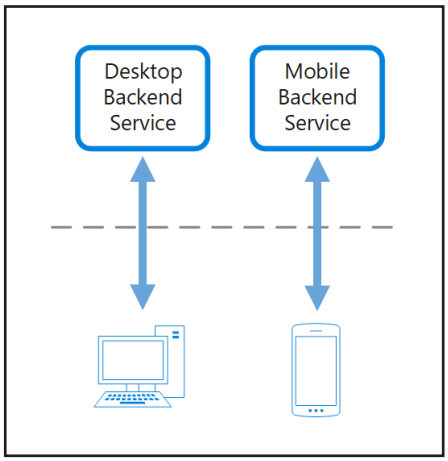


Ces différences entraînent des exigences concurrentes pour le serveur principal. Le serveur principal a besoin de modifications régulières et importantes pour desservir à la fois l'IU web de bureau et l'application mobile. Souvent, des équipes d'interface distinctes travaillent sur chaque serveur frontal, conduisant le serveur principal à devenir un goulot d'étranglement dans le processus de développement. Des exigences de mise à jour contradictoires, et la nécessité de maintenir le service en fonctionnement pour les deux serveurs frontaux, peuvent entraîner la mise en place de beaucoup d'efforts sur une seule ressource déployable.

Du fait que l'activité de développement est axée sur le service de serveur principal, une équipe distincte peut être créée pour gérer et tenir à jour le serveur principal. En fin de compte, cela se traduit par une déconnexion entre les équipes de développement du serveur principal et de l'interface, ce qui alourdit la charge de l'équipe du serveur principal afin d'équilibrer les exigences concurrentes des différentes équipes d'IU. Quand une équipe d'interface a besoin de modifications apportées au serveur principal, ces changements doivent être validés avec d'autres équipes d'interface avant de pouvoir être intégrés dans le serveur principal.

Solution

Créez un serveur principal par interface utilisateur. Ajustez le comportement et les performances de chaque serveur principal pour mieux adapter les besoins de l'environnement du serveur frontal, sans vous soucier d'affecter d'autres expériences de serveur frontal.



Du fait que chaque serveur principal est spécifique à une interface, il peut être optimisé pour cette interface. Ainsi, il sera plus petit, moins complexe et probablement plus rapide qu'un serveur principal générique qui essaie de satisfaire les exigences pour toutes les interfaces. Chaque équipe d'interface dispose d'une autonomie pour contrôler son propre serveur principal et ne s'appuie pas sur une équipe de développement de serveur principal centralisée. Cela donne à l'équipe d'interface la flexibilité de sélection du langage, de la cadence de lancement, de l'établissement des priorités de la charge de travail et de l'intégration de la fonction dans leur serveur principal.

Problèmes et considérations

- Considérez combien de serveurs principaux doivent être déployés.
- Si des interfaces différentes (par exemple, des clients mobiles) vont soumettre les mêmes demandes, examinez s'il est nécessaire d'implémenter un serveur principal pour chaque interface, ou si un serveur principal unique suffira.
- La duplication de code sur l'ensemble des services est très probable lors de l'implémentation de ce patron.
- Les services de serveur principal axés sur le serveur frontal doivent contenir uniquement un comportement et une logique spécifiques à un client. La logique métier générale et autres fonctions globales devraient être gérées ailleurs dans votre application.
- Pensez à comment ce patron peut se refléter dans les responsabilités d'une équipe de développement.
- Envisagez le temps nécessaire à l'implémentation de ce patron. L'effort de développement des nouveaux serveurs principaux occasionnera-t-il une dette technique pendant que vous continuez à soutenir le serveur principal générique existant ?

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Un service de serveur principal polyvalent ou partagé doit être tenu à jour avec une surcharge importante de développement.
- Vous souhaitez optimiser le serveur principal pour les exigences d'interfaces client spécifiques.
- Les personnalisations sont apportées à un serveur principal polyvalent pour accueillir plusieurs interfaces.
- Un autre langage est mieux adapté pour le serveur principal d'une interface utilisateur différente.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Les interfaces soumettent des demandes identiques ou similaires au serveur principal.
- Une seule interface est utilisée pour interagir avec le serveur principal.

Conseils connexes

- Patron Gateway Aggregation (agrégation de passerelles)
- Patron Gateway Offloading (déchargement de passerelle)
- Patron Gateway Routing (routage de passerelle)

Patron Bulkhead (cloison)

Isolez les éléments d'une application dans des pools de sorte que si l'un tombe en panne, les autres continueront à fonctionner.

Ce patron est nommé Bulkhead (cloison) car il ressemble aux partitions sectionnées de la coque d'un navire. Si la coque d'un navire subit des dommages, seule la section endommagée se remplit d'eau, ce qui empêche le navire de couler.

Contexte et problème

Une application basée sur le cloud peut inclure plusieurs services, chaque service ayant un ou plusieurs consommateurs. Une charge excessive ou l'échec d'un service affectera tous les consommateurs du service.

En outre, un consommateur peut envoyer des demandes à plusieurs services en même temps, en utilisant des ressources pour chaque demande. Lorsque le client envoie une requête à un service qui est mal configuré ou ne répond ne pas, les ressources utilisées par la demande du client peuvent ne pas être libérées dans un délai raisonnable. Comme les demandes continuent de parvenir au service, ces ressources peuvent venir à manquer. Par exemple, le pool de connexions du client peut être épuisé. À ce stade, cela a un impact sur les demandes par le consommateur à d'autres services. Au final, le consommateur ne peut plus envoyer de demandes à d'autres services, et pas seulement au service qui à l'origine ne répondait pas.

Le même problème de l'épuisement des ressources affecte les services avec plusieurs consommateurs. Un grand nombre de demandes provenant d'un client peuvent épuiser les ressources disponibles dans le service. Les autres consommateurs ne peuvent plus utiliser le service, provoquant un effet d'échecs en cascade.

Solution

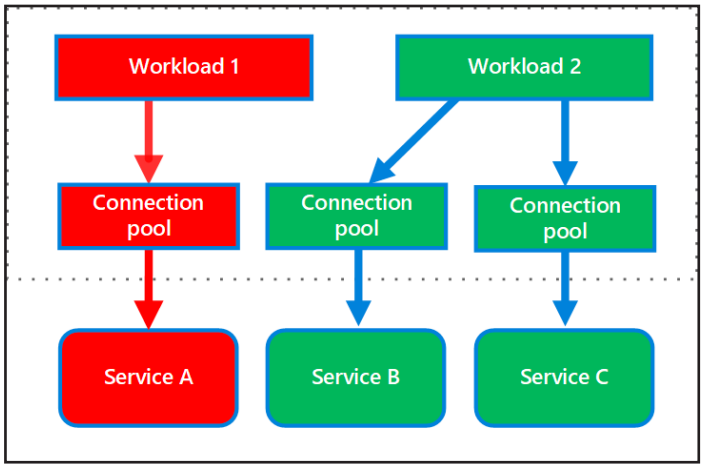
Partitionnez les instances de service en différents groupes, basées sur les exigences de disponibilité et de charge des consommateurs. Cette conception contribue à isoler les échecs et permet de maintenir la fonctionnalité de service pour certains consommateurs, même en cas d'échec.

Un consommateur peut également partitionner des ressources afin de garantir que les ressources utilisées pour appeler un service n'affectent pas les ressources utilisées pour appeler un autre service. Par exemple, un consommateur qui appelle plusieurs services peut se voir affecter un pool de connexions pour chaque service. Si un service commence à échouer, il affecte seulement le pool de connexions affecté pour ce service, ce qui permet au consommateur de continuer à utiliser les autres services.

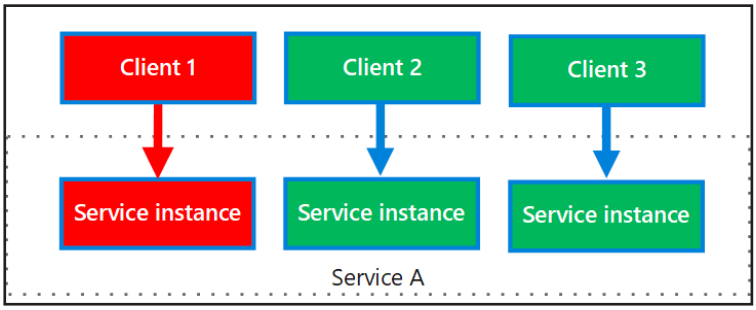
Les avantages de ce patron incluent les suivants :

- Protège les consommateurs et les services contre des échecs en cascade. Un problème affectant un consommateur ou un service peut être isolé au sein de sa propre cloison, empêchant l'échec de l'ensemble de la solution.
- Vous permet de conserver certaines fonctionnalités en cas d'échec d'un service. Les autres services et fonctions de l'application continueront à fonctionner.
- Vous permet de déployer des services qui offrent une qualité de service différente pour l'utilisation d'applications. Un pool de consommateurs hautement prioritaires peut être configuré pour utiliser des services prioritaires.

Le diagramme suivant illustre les cloisons structurées autour des pools de connexions qui appellent des services individuels. Si le Service A échoue ou provoque un autre problème, le pool de connexions est isolé, ainsi seules les charges de travail utilisant le pool de fils de discussion affecté au Service A sont touchées. Les charges de travail qui utilisent les services B et C ne sont pas affectées et peuvent continuer à travailler sans interruption.



Le diagramme suivant illustre plusieurs clients appelant un service unique. Chaque client se voit attribuer une instance de service distincte. Le client 1 a soumis trop de demandes et a surchargé son instance. Du fait que chaque instance de service est isolée des autres, les autres clients peuvent continuer à passer des appels.



Problèmes et considérations

- Définissez des partitions autour des exigences métier et techniques de l'application.
- Lorsque vous partitionnez des services ou des consommateurs en cloisons, envisagez le niveau d'isolement proposé par la technologie ainsi que la surcharge en termes de coûts, de performances et de facilité de gestion.
- Envisagez d'associer des cloisons avec les patrons Retry (Nouvelle tentative), Circuit breaker (Disjoncteur) et Throttling (Limitation) pour fournir une gestion plus sophistiquée des défaillances.
- Lorsque vous partitionnez des consommateurs en cloisons, envisagez d'utiliser des processus, des pools de fils de discussion et des sémaphores. Les projets comme Netflix Hystrix et Polly offrent une infrastructure pour la création de cloisons consommateur
- Lorsque vous partitionnez des services en cloisons, envisagez de les déployer dans des processus, des conteneurs ou des machines virtuelles distincts. Les conteneurs offrent un bon équilibre de l'isolation des ressources avec une surcharge assez faible.
- Les services qui communiquent à l'aide de messages asynchrones peuvent être isolés par le biais de différents ensembles de files d'attente. Chaque file d'attente peut avoir un ensemble dédié d'instances traitant des messages dans la file d'attente, ou un groupe unique d'instances utilisant un algorithme pour enlever la file d'attente et répartir le traitement.
- Déterminez le niveau de granularité pour les cloisons. Par exemple, si vous souhaitez distribuer des locataires sur des partitions, vous pouvez placer chaque locataire dans une partition séparée, ou plusieurs locataires dans une seule partition.
- Surveillez la performance et le SLA de chaque partition.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Isolez les ressources utilisées pour consommer un ensemble de services de serveur principal, surtout si l'application peut fournir un certain niveau de fonctionnalité même lorsque l'un des services ne répond pas.
- Isolez les consommateurs critiques des consommateurs standards.
- Protégez l'application contre tous échecs en cascade.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Une utilisation moins efficace des ressources peut ne pas être acceptable dans le projet.
- Il n'est pas utile d'ajouter de la complexité.

Exemple

Le fichier de configuration Kubernetes suivant crée un conteneur isolé pour exécuter un service unique, avec ses propres limites et ressources processeur et mémoire.

```
apiVersion: v1
kind: Pod
metadata:
  name: drone-management
spec:
  containers:
  - name: drone-management-container
    image: drone-service
    resources:
      requests:
        memory: “64Mi”
        cpu: “250m”
      limits:
        memory: “128Mi”
        cpu: “1”
```

Conseils connexes

- Patron Circuit Breaker (Disjoncteur)
- Conception d’applications résilientes pour Azure
- Patron Retry (Nouvelle tentative)
- Patron Throttling (Limitation)

Patron Cache-Aside (Stockage des données dans le cache)

Chargez des données à la demande dans un cache à partir d’un magasin de données. Cela peut améliorer les performances et contribue également à maintenir la cohérence entre les données contenues dans le cache et celles dans le magasin de données sous-jacent.

Contexte et problème

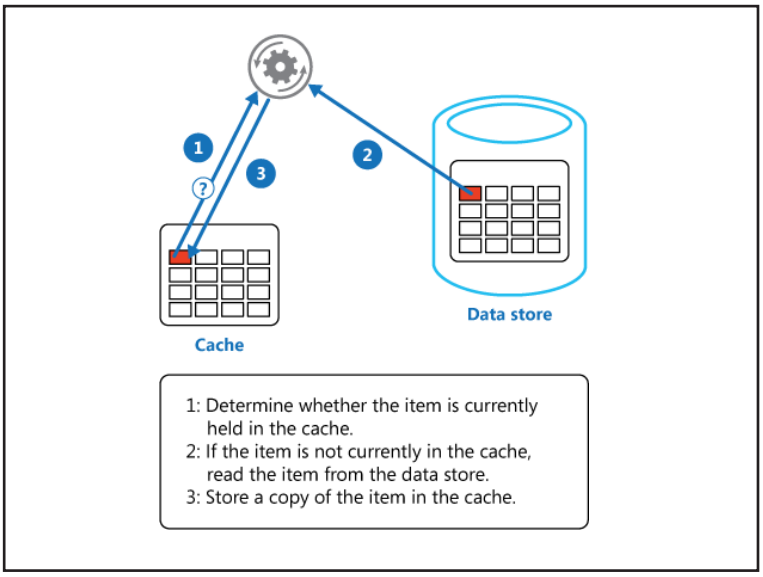
Les applications utilisent un cache pour améliorer l’accès répété à l’information détenue dans un magasin de données. Il est toutefois impossible de s’attendre à ce que les données mises en cache soient toujours tout à fait cohérentes avec les données du magasin de données. Les applications devraient implémenter une stratégie permettant de s’assurer que les données dans le cache sont aussi à jour que possible, mais pouvant également détecter et gérer des situations qui surviennent à la péremption des données dans le cache.

Solution

Plusieurs systèmes de mise en cache commerciaux fournissent des opérations de lecture et de double écriture/d’écriture différée. Dans ces systèmes, une application récupère des données en référençant le cache. Si les données ne sont pas dans le cache, elles sont récupérées depuis le magasin de données et ajoutées au cache. Toute modification de données contenues dans le cache est automatiquement réécrite dans le magasin de données également.

Pour les caches qui ne fournissent pas cette fonctionnalité, il incombe aux applications qui utilisent le cache de tenir les données à jour.

Une application peut émuler la fonctionnalité de mise en cache de lecture en implémentant la stratégie cache-aside. Cette stratégie charge des données dans le cache à la demande. La figure illustre l'utilisation du patron Cache-Aside (Stockage des données dans le cache) pour stocker des données dans le cache.



Si une application met à jour des informations, elle peut suivre la stratégie d'écriture continue en apportant la modification dans le magasin de données et en invalidant l'élément correspondant dans le cache.

Lorsque l'élément est ensuite requis, l'utilisation de la stratégie de cache-aside entraînera la récupération des données actualisées dans le magasin de données et leur nouvelle insertion dans le cache.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Durée de vie des données mises en cache. De nombreux caches implémentent une stratégie d'expiration qui invalide les données et les supprime du cache si elles ne sont pas consultées pendant une période déterminée. Pour que cache-aside soit efficace, assurez-vous que la stratégie d'expiration correspond au patron d'accès pour les applications qui utilisent les données. Ne raccourcissez pas trop la période d'expiration car cela peut conduire les applications à récupérer sans cesse des données du magasin de données et à les ajouter au cache. De même, n'allongez pas la période d'expiration à tel point que les données mises en cache soient susceptibles de devenir obsolètes. N'oubliez pas que la mise en cache est plus efficace pour les données relativement statiques ou des données qui sont lues fréquemment.

Suppression de données. La plupart des caches ont une taille limitée par rapport au magasin de données d'origine des données et ils vont supprimer des données si nécessaire. Les caches adoptent généralement une stratégie du dernier récemment utilisé pour sélectionner les éléments à supprimer, mais cela peut être personnalisable. Configurez la propriété d'expiration globale, les autres propriétés du cache et la propriété d'expiration de chaque élément mis en cache pour vous assurer que le cache soit rentable. Il n'est pas toujours approprié d'appliquer une stratégie d'éviction globale pour chaque élément dans le cache. Par exemple, si la récupération dans le magasin de données d'un élément mis en cache est très onéreuse, il peut être avantageux de maintenir cet élément dans le cache au détriment de d'éléments plus fréquemment consultés mais moins coûteux.

Amorçage du cache. De nombreuses solutions préremplissent le cache avec des données dont une application est susceptible d'avoir besoin dans le cadre du traitement du démarrage. Le patron Cache-Aside (Stockage des données dans le cache) peut encore être utile si certaines de ces données expirent ou sont supprimées.

Cohérence. L'implémentation du patron Cache-Aside (Stockage des données dans le cache) ne garantit pas la cohérence entre le magasin de données et le cache. Un élément dans le magasin de données peut être modifié à tout moment par un processus externe et ce changement peut ne pas être répercuté dans le cache jusqu'au prochain chargement de l'élément. Dans un système qui réplique des données entre des magasins de données, ce problème peut devenir grave si la synchronisation se produit fréquemment.

Mise en cache locale (en mémoire). Un cache pourrait être local pour une instance de l'application et stocké en mémoire. Cache-aside peut être utile dans ce contexte si une application accède à plusieurs reprises aux mêmes données. Toutefois, un cache local est privé et donc différentes instances de l'application pourraient chacune avoir une copie des mêmes données mises en cache. Ces données pourraient rapidement devenir contradictoires entre les caches. C'est pourquoi il peut être nécessaire d'amener à expiration des données détenues dans un cache privé et de les actualiser plus fréquemment. Dans ces scénarios, envisagez d'enquêter sur l'utilisation d'un mécanisme de mise en cache distribuée ou partagée.

Quand utiliser ce patron

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Durée de vie des données mises en cache. De nombreux caches implémentent une stratégie d'expiration qui invalide les données et les supprime du cache si elles ne sont pas consultées pendant une période déterminée. Pour que cache-aside soit efficace, assurez-vous que la stratégie d'expiration correspond au patron d'accès pour les applications qui utilisent les données. Ne raccourcissez pas trop la période d'expiration car cela peut conduire les applications à récupérer sans cesse des données du magasin de données et à les ajouter au cache. De même, n'allongez pas la période d'expiration à tel point que les données mises en cache soient susceptibles de devenir obsolètes. N'oubliez pas que la mise en cache est plus efficace pour les données relativement statiques ou des données qui sont lues fréquemment.

Suppression de données. La plupart des caches ont une taille limitée par rapport au magasin de données d'origine des données et ils vont supprimer des données si nécessaire. Les caches adoptent généralement une stratégie du dernier récemment utilisé pour sélectionner les éléments à supprimer, mais cela peut être personnalisable. Configurez la propriété d'expiration globale et les autres propriétés du cache ainsi que la propriété d'expiration de chaque élément mis en cache pour vous assurer que le cache soit rentable. Il n'est pas toujours approprié d'appliquer une stratégie d'éviction globale pour chaque élément dans le cache. Par exemple, si la récupération dans le magasin de données d'un élément mis en cache est très onéreuse, il peut être avantageux de maintenir cet élément dans le cache au détriment de d'éléments plus fréquemment consultés mais moins coûteux.

Amorçage du cache. De nombreuses solutions préremplissent le cache avec des données dont une application est susceptible d'avoir besoin dans le cadre du traitement du démarrage. Le patron Cache-Aside (Stockage des données dans le cache) peut encore être utile si certaines de ces données expirent ou sont supprimées.

Cohérence. L'implémentation du patron Cache-Aside (Stockage des données dans le cache) ne garantit pas la cohérence entre le magasin de données et le cache. Un élément dans le magasin de données peut être modifié à tout moment par un processus externe et ce changement peut ne pas être répercuté dans le cache jusqu'au prochain chargement de l'élément. Dans un système qui réplique des données entre des magasins de données, ce problème peut devenir grave si la synchronisation se produit fréquemment.

Mise en cache locale (en mémoire). Un cache pourrait être local pour une instance de l'application et stocké en mémoire. Cache-aside peut être utile dans ce contexte si une application accède à plusieurs reprises aux mêmes données. Toutefois, un cache local est privé et donc différentes instances de l'application pourraient chacune avoir une copie des mêmes données mises en cache. Ces données pourraient rapidement devenir contradictoires entre les caches. C'est pourquoi il peut être nécessaire d'amener à expiration des données détenues dans un cache privé et de les actualiser plus fréquemment. Dans ces scénarios, envisagez d'enquêter sur l'utilisation d'un mécanisme de mise en cache distribuée ou partagée.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Un cache ne fournit pas d'opérations natives de lecture et de double écriture.
- La demande de ressources est imprévisible. Ce patron permet aux applications de charger des données à la demande. Il n'émet aucune hypothèse à l'avance quant à savoir de quelles données une application aura besoin.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Le jeu de données mis en cache est statique. Si les données peuvent s'intégrer dans l'espace de cache disponible, amorcez le cache avec les données au démarrage et appliquez une stratégie qui empêche l'expiration des données.
- Pour mettre en cache les informations d'état de session dans une application web hébergée dans une batterie de serveurs Web. Dans cet environnement, vous devriez éviter d'introduire des dépendances basées sur l'affinité de client-serveur.

Exemple

Dans Microsoft Azure, vous pouvez utiliser Azure Redis Cache pour créer un cache distribué qui peut être partagé par plusieurs instances d'une application.

Pour vous connecter à une instance Azure Redis Cache, appelez la méthode Connect statique et faites circuler la chaîne de connexion. La méthode renvoie un ConnectionMultiplexer qui représente la connexion. Une approche pour partager une instance de ConnectionMultiplexer dans votre application consiste à avoir une propriété statique qui retourne une instance connectée, semblable à l'exemple suivant. Cette approche fournit une façon thread-safe d'initialiser uniquement une seule instance connectée.

```
private static ConnectionMultiplexer Connection;

// Redis Connection string info
private static Lazy<ConnectionMultiplexer> lazyConnection = new Lazy<ConnectionMultiplexer>(() =>
{
    string cacheConnection = ConfigurationManager.AppSettings["CacheConnection"].ToString();
    return ConnectionMultiplexer.Connect(cacheConnection);
});

public static ConnectionMultiplexer Connection => lazyConnection.Value;
```

La méthode GetMyEntityAsync dans l'exemple de code suivant illustre une implémentation du patron Cache-Aside (Stockage des données dans le cache) basé sur Azure Redis Cache. Cette méthode récupère un objet dans le cache en utilisant l'approche de lecture.

Un objet est identifié à l'aide d'un ID entier comme clé. La méthode GetMyEntityAsync essaie de récupérer un élément avec cette clé du cache. Si un élément correspondant est trouvé, il est renvoyé. S'il n'y a aucune correspondance dans le cache, la méthode GetMyEntityAsync récupère l'objet dans un magasin de données, l'ajoute au cache, puis le renvoie. Le code qui lit réellement les données du magasin de données ne figure pas ici car il dépend du magasin de données. Veuillez noter que l'élément mis en cache est configuré pour expirer afin de l'empêcher de devenir caduque s'il est mis à jour ailleurs.

```
// Set five minute expiration as a default
private const double DefaultExpirationTimeInMinutes = 5.0;

public async Task<MyEntity> GetMyEntityAsync(int id)
{
  // Define a unique key for this method and its parameters.
  var key = $"MyEntity:{id}";
  var cache = Connection.GetDatabase();

  // Try to get the entity from the cache.
  var json = await cache.StringGetAsync(key).ConfigureAwait(false);
  var value = string.IsNullOrWhiteSpace(json)
                ? default(MyEntity)
                : JsonConvert.DeserializeObject<MyEntity>(json);

  if (value == null) // Cache miss
  {
    // If there's a cache miss, get the entity from the original store and cache it.
    // Code has been omitted because it's data store dependent.
    value = ...;

    // Avoid caching a null value.
    if (value != null)
    {
      // Put the item in the cache with a custom expiration time that
      // depends on how critical it is to have stale data.
      await cache.StringSetAsync(key, JsonConvert.SerializeObject(value)).ConfigureAwait(false);
      await cache.KeyExpireAsync(key, TimeSpan.FromMinutes(DefaultExpirationTimeInMinutes)).
ConfigureAwait(false);
    }
  }

  return value;
}
```

Les exemples utilisent l'API du Cache Redis Azure pour accéder au magasin et récupérer les informations du cache. Pour plus d'informations, consultez Utilisation de Microsoft Azure Redis Cache et Comment créer une application web avec Redis Cache.

La méthode UpdateEntityAsync ci-dessous montre comment invalider un objet dans le cache lorsque la valeur est modifiée par l'application. Le code met à jour le magasin de données d'origine, puis supprime du cache l'élément mis en cache.

```
public async Task UpdateEntityAsync(MyEntity entity)
{
    // Update the object in the original data store.
    await this.store.UpdateEntityAsync(entity).ConfigureAwait(false);

    // Invalidate the current cache object.
    var cache = Connection.GetDatabase();
    var id = entity.Id;
    var key = $"MyEntity:{id}"; // The key for the cached object.
    await cache.KeyDeleteAsync(key).ConfigureAwait(false); // Delete this key from the cache.
}
```

Remarque :

L'ordre des étapes est important. Mettez à jour le magasin de données avant de supprimer l'élément du cache. Si vous commencez par supprimer l'élément mis en cache, il y a une petite fenêtre de temps durant laquelle un client pourrait récupérer l'élément avant la mise à jour du magasin de données. Cela se traduira par une absence de cache (parce que l'élément a été supprimé du cache), entraînant l'extraction de la version précédente de l'élément dans le magasin de données avant d'être rajoutée au cache. Il en résultera des données de cache caduques.

Conseils connexes

Les informations suivantes peuvent être pertinentes lors de l'implémentation de ce patron :

- Conseils de mise en cache. Fournissent des informations supplémentaires sur la manière dont vous pouvez mettre en cache des données dans une solution cloud et les problèmes que vous devriez prendre en considération lorsque vous implémentez un cache.

- Manuel de cohérence de données. Les applications cloud utilisent généralement des données qui se propagent dans des magasins de données. La gestion et la tenue à jour de la cohérence des données dans cet environnement sont un aspect critique du système, en particulier les problèmes de simultanéité et de disponibilité qui peuvent survenir. Ce manuel décrit les problèmes de cohérence entre les données distribuées et résume comment une application peut implémenter une éventuelle cohérence afin de maintenir la disponibilité des données.

Patron Circuit Breaker (Disjoncteur)

Gérez les erreurs dont la récupération pourrait avoir une durée variable lors de la connexion à une ressource ou un service distant. Cela peut améliorer la stabilité et la résilience d'une application.

Contexte et problème

Dans un environnement distribué, les appels aux ressources et services distants peuvent échouer en raison d'erreurs temporaires, telles que des connexions réseau lentes, des délais d'expiration ou bien un excès de sollicitation ou une indisponibilité temporaires des ressources. Ces erreurs se corrigent en général d'elles-mêmes après un court laps de temps et une application cloud robuste devrait être prête à les gérer en utilisant une stratégie telle que le patron Retry (Nouvelle tentative).

Toutefois, il peut exister des situations où les erreurs sont dues à des événements imprévus, et dont la correction pourrait prendre beaucoup plus longtemps. Ces erreurs peuvent s'étendre en gravité d'une perte partielle de la connectivité à la défaillance complète d'un service. Dans ces situations, il peut être inutile pour une application de réessayer sans cesse une opération qui a peu de chance d'aboutir. L'application devrait au contraire accepter rapidement que l'opération a échoué et gérer cet échec en conséquence.

En outre, si un service est très occupé, une défaillance dans une partie du système pourrait conduire à des échecs en cascade. Par exemple, une opération qui appelle un service pourrait être configurée pour implémenter un délai d'expiration et répondre par un message d'erreur si le service ne parvient pas à répondre dans ce délai. Toutefois, cette stratégie pourrait provoquer de nombreuses demandes simultanées pour que la même opération soit bloquée jusqu'à la fin du délai d'expiration. Ces demandes bloquées pourraient détenir des ressources système critiques telles que les suivantes : mémoire, fils de discussion, connexions de base de données et ainsi de suite. Par conséquent, ces ressources pourraient s'épuiser, provoquant l'échec d'autres parties éventuellement indépendantes du système qui ont besoin d'utiliser les mêmes ressources. Dans ces situations, il serait préférable pour l'opération d'échouer immédiatement et de tenter un appel au service uniquement s'il est susceptible d'aboutir. Notez que la définition d'un délai d'expiration plus court pourrait aider à résoudre ce problème, mais le délai d'expiration ne doit pas être si court que l'opération échoue la plupart du temps, même si la demande vers le service pourrait finir par aboutir.

Solution

Le patron Circuit Breaker (Disjoncteur) peut empêcher une application d'essayer à plusieurs reprises d'exécuter une opération qui est susceptible d'échouer, en lui permettant de continuer sans attendre la correction de l'erreur ou en gaspillant des cycles microprocesseur pendant qu'elle détermine que l'erreur est de longue durée. Le patron Circuit Breaker (Disjoncteur) permet également à une application de détecter si l'erreur a été corrigée. Si le problème semble avoir été corrigé, l'application peut essayer d'appeler l'opération.

Le but du patron Circuit Breaker (Disjoncteur) est différent de celui du patron Retry (Nouvelle tentative). Le patron Retry (Nouvelle tentative) permet à une application de relancer une opération dans l'espoir qu'elle va aboutir. Le patron Circuit Breaker (Disjoncteur) empêche une application d'exécuter une opération qui est susceptible d'échouer. Une application peut combiner ces deux patrons en utilisant le patron Retry (Nouvelle tentative) pour appeler une opération via un disjoncteur. Toutefois, la logique de nouvelle tentative devrait être sensible à toutes les exceptions renvoyées par le disjoncteur et abandonner les nouvelles tentatives si le disjoncteur indique qu'une erreur n'est pas temporaire.

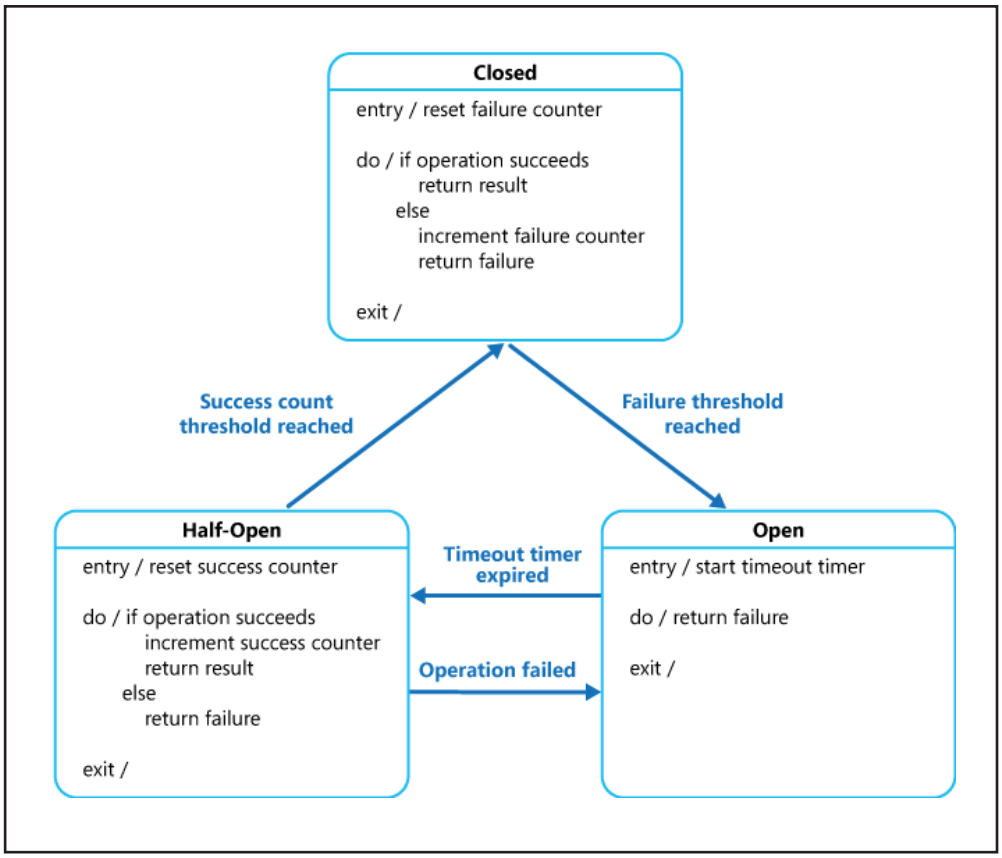
Un disjoncteur agit comme un proxy pour les opérations qui risquent d'échouer. Le proxy doit surveiller le nombre d'échecs récents qui ont eu lieu et utiliser cette information pour décider s'il faut autoriser la poursuite de l'opération ou simplement renvoyer une exception immédiatement.

Le proxy peut être implémenté comme une machine à états avec les états suivants qui imitent la fonctionnalité d'un disjoncteur électrique :

- **Fermé :** la demande de l'application est acheminée vers l'opération. Le proxy conserve le décompte du nombre d'échecs récents et, si l'appel à l'opération n'aboutit pas, le proxy augment cette valeur. Si le nombre d'échecs récents dépasse un seuil déterminé dans un laps de temps donné, le proxy est placé dans l'état **Ouvert**. À ce stade, le proxy démarre un minuteur de délai d'attente et, lorsque ce minuteur expire, le proxy est placé dans l'état **semi-ouvert**.
    - Le minuteur de délai d'attente vise à accorder au système le temps nécessaire pour résoudre le problème qui a provoqué l'échec avant d'autoriser l'application à essayer d'exécuter à nouveau l'opération.

- **Ouvert :** la demande de l'application échoue immédiatement et une exception est renvoyée à l'application.

- **Semi-ouvert :** un nombre limité de demandes de l'application est autorisé à passer directement et à appeler l'opération. Si ces demandes aboutissent, on suppose que l'erreur qui était auparavant à l'origine de la défaillance a été corrigée et le disjoncteur bascule vers l'état **fermé** (réinitialisation du compteur d'échecs). Si une demande échoue, le disjoncteur suppose que l'erreur est toujours présente et il revient donc à l'état **ouvert** et redémarre le minuteur de délai d'attente pour accorder au système une nouvelle période de temps pour récupérer après l'échec.
    - L'état **semi-ouvert** est utile pour empêcher un service de récupération d'être soudainement inondé de demandes. Alors qu'un service récupère, il pourrait être en mesure de soutenir un nombre limité de demandes jusqu'à ce que la récupération soit terminée, mais alors que la reprise est en cours, une saturation de tâches peut provoquer l'expiration du service ou un nouvel échec.



Dans la figure, le compteur d'échecs utilisé par l'état **fermé** est basé sur le temps. Il se réinitialise automatiquement à intervalles réguliers. Cela aide à empêcher le disjoncteur d'entrer dans l'état **ouvert** s'il subit des défaillances occasionnelles. Le seuil de rupture qui déclenche le disjoncteur dans l'état **ouvert** n'est obtenu que lorsqu'un nombre spécifique de défaillances s'est produit pendant un intervalle spécifié. Le compteur utilisé par l'état **semi-ouvert** enregistre le nombre de tentatives réussies pour appeler l'opération. Le disjoncteur revient à l'état **fermé** après un nombre déterminé d'appels consécutifs d'opérations ayant abouti. Si toute invocation échoue, le disjoncteur pénètre dans l'état **ouvert** immédiatement et le compteur de réussite sera réinitialisé la prochaine fois qu'il accède à l'état **semi-ouvert**.

La manière dont les récupération du système sont gérées à l'extérieur, éventuellement en restaurant ou en redémarrant un composant en échec ou en réparant une connexion réseau.

Le patron Circuit Breaker (Disjoncteur) fournit de la stabilité pendant que le système récupère d'une défaillance et minimise l'impact sur les performances. Il peut aider à maintenir le temps de réponse du système en rejetant rapidement une demande pour une opération qui est susceptible d'échouer, au lieu d'attendre que l'opération arrive à expiration ou ne revienne jamais. Si le disjoncteur déclenche un événement chaque fois qu'il change d'état, cette information peut être utilisée pour surveiller la santé de la partie du système protégée par le disjoncteur ou pour alerter un administrateur quand un disjoncteur passe sur l'état **ouvert**.

Le patron est personnalisable et peut être adapté selon le type de l'échec éventuel. Par exemple, vous pouvez appliquer un minuteur de délai d'attente croissant à un disjoncteur. Vous pourriez placer le disjoncteur dans l'état **ouvert** pendant quelques secondes au départ, puis, si la défaillance n'a pas été résolue, augmenter le délai d'expiration de quelques minutes, et ainsi de suite. Dans certains cas, plutôt que l'état **ouvert** renvoyant un échec et déclenchant une exception, il pourrait être utile de renvoyer une valeur par défaut significative pour l'application.

Problèmes et considérations

Prenez en compte les points suivants lorsque vous choisissez le mode d'implémentation de ce patron :

Gestion des exceptions. Une application appelant une opération via un disjoncteur doit être préparée pour gérer les exceptions déclenchées si l'opération n'est pas disponible. La façon dont les exceptions sont gérées sera spécifique à l'application. Par exemple, une application pourrait temporairement dégrader sa fonctionnalité, appeler une autre opération pour tenter d'effectuer la même tâche ou obtenir les mêmes données, ou encore signaler l'exception à l'utilisateur et lui demander de réessayer plus tard.

Types d'exceptions. Une demande peut échouer pour plusieurs raisons, dont certaines pourraient indiquer un type plus sévère de défaillance que d'autres. Par exemple, une requête peut échouer parce qu'un service distant s'est bloqué et que sa récupération prendra plusieurs minutes, ou en raison d'un délai d'expiration dû à une surcharge temporaire du service. Un disjoncteur peut être en mesure d'examiner les types d'exceptions qui se produisent et d'adapter sa stratégie selon la nature de ces exceptions. Par exemple, il pourrait avoir besoin d'un plus grand nombre d'exceptions d'expiration pour déclencher le disjoncteur sur l'état **ouvert** par rapport au nombre d'échecs dus à l'indisponibilité complète du service.

Journalisation. Pour permettre à un administrateur de surveiller l'intégrité de l'opération, un disjoncteur doit enregistrer toutes les demandes ayant échoué (et les demandes pouvant aboutir).

Récupération. Vous devez configurer le disjoncteur pour qu'il s'adapte au patron de récupération probable de l'opération qu'il protège. Par exemple, si le disjoncteur reste dans l'état **ouvert** pendant une longue période, il peut déclencher des exceptions même si la raison de l'échec a été résolue. De même, un disjoncteur pourrait fluctuer et réduire les temps de réponse d'applications s'il passe trop rapidement de l'état **ouvert** à l'état **semi-ouvert**.

Test des opérations ayant échoué. Dans l'état **ouvert**, plutôt que d'utiliser un minuteur pour déterminer le moment de passer à l'état **semi-ouvert**, un disjoncteur peut plutôt effectuer un test ping périodique de la ressource ou du service distant pour déterminer s'il ou si elle est à nouveau disponible. Ce test ping peut prendre la forme d'une tentative d'appel d'une opération qui avait échoué par le passé, ou il pourrait utiliser une opération spéciale fournie par le service distant spécifiquement pour tester l'intégrité du service, tel que décrit par le patron Health Endpoint Monitoring (Point de terminaison pour la surveillance de fonctionnement).

Commande manuelle. Dans un système où le temps de récupération pour une fonction ayant échoué est extrêmement variable, il est utile de fournir une option de réinitialisation manuelle qui permet à un administrateur de fermer un disjoncteur (et de réinitialiser le compteur d'échecs). De même, un administrateur peut forcer un disjoncteur dans l'état **ouvert** (et redémarrer le minuteur du délai d'attente) si l'opération protégée par le disjoncteur est temporairement indisponible.

Simultanéité. Un grand nombre d'instances simultanées d'une application pourraient accéder au même disjoncteur. L'implémentation ne doit pas bloquer les demandes simultanées ni ajouter une surcharge excessive à chaque appel à une opération.

Différenciation de la ressource. Soyez prudent lorsque vous utilisez un disjoncteur unique pour un type de ressource s'il est possible qu'il y ait plusieurs fournisseurs indépendants sous-jacents. Par exemple, dans un magasin de données qui contient plusieurs partitions, l'une d'elles peut être entièrement accessible tandis qu'une autre rencontre un problème temporaire. Si les réponses d'erreurs dans ces scénarios sont fusionnées, une application peut essayer d'accéder à certaines partitions même lorsque l'échec est fort probable, alors que l'accès aux autres partitions peut être bloqué, même si une réussite est probable.

Disjoncteur accéléré. Une réponse de défaillance peut parfois contenir suffisamment d'informations pour permettre au disjoncteur de se déclencher immédiatement et de rester déclenché pour un minimum de temps. Par exemple, la réponse d'erreur provenant d'une ressource partagée qui est surchargée peut indiquer qu'une nouvelle tentative immédiate n'est pas recommandée et que l'application devrait plutôt réessayer dans quelques minutes.

Remarques :

Un service peut retourner HTTP 429 (trop de demandes) s'il limite le client, ou HTTP 503 (Service indisponible) si le service n'est pas disponible actuellement. La réponse peut inclure des informations supplémentaires, telles que la durée prévue du retard.

Nouveau lancement de demandes ayant échoué. Dans l'état **ouvert**, plutôt qu'un simple échec rapide, un disjoncteur pourrait également enregistrer les détails de chaque demande dans un journal et s'organiser pour que ces demandes soient relancées lorsqu'une ressource ou un service distant est disponible.

Délais d'expiration inappropriés sur des services externes. Un disjoncteur peut ne pas être en mesure de protéger pleinement les demandes provenant d'opérations qui échouent dans des services externes qui sont configurés avec un délai d'attente long. Si le délai d'expiration est trop long, un thread exécutant un disjoncteur pourrait être bloqué pendant une longue période avant que le disjoncteur indique que l'opération a échoué. Pendant ce temps, de nombreuses autres instances d'application pourraient également essayer d'appeler le service par l'intermédiaire du disjoncteur et fixer un grand nombre de fils de discussion avant que tous n'échouent.

Quand utiliser ce patron

Utilisez ce patron :

- Pour empêcher une demande d'essayer d'appeler un service distant ou d'accéder à une ressource partagée si cette opération est hautement susceptible d'échouer.

Ce patron n'est pas recommandé :

- Pour la gestion de l'accès à des ressources privées locales dans une application, telles que la structure de données en mémoire. Dans cet environnement, l'utilisation d'un disjoncteur surchargerait votre système.
- Pour remplacer la gestion des exceptions dans la logique métier de vos applications.

Exemple

Dans une application web, plusieurs des pages sont remplies avec des données extraites d'un service externe. Si le système met en œuvre une mise en cache minimum, la plupart des correspondances vers ces pages provoqueront un aller-retour vers le service. Des connexions entre l'application web et le service pourraient être configurées avec un délai d'expiration (en général de 60 secondes) et si le service ne répond pas dans ce délai, la logique de chaque page web supposera que le service n'est pas disponible et déclenchera une exception.

Toutefois, si le service échoue et le système est très occupé, les utilisateurs pourraient être contraints d'attendre jusqu'à 60 secondes avant qu'une exception se produise. Finalement, les ressources telles que la mémoire, les connexions et les fils de discussion pourraient être épuisées, empêchant d'autres utilisateurs de se connecter au système, même s'ils n'accèdent pas à des pages qui extraient des données du service.
Une mise à l'échelle du système en ajoutant des serveurs web et en implémentant un équilibrage de charge peut retarder l'épuisement des ressources, mais elle ne résoudra pas le problème parce que les demandes utilisateur resteront sans réponse et tous les serveurs web pourraient encore au final venir à manquer de ressources.

Inclure dans un wrapper la logique qui se connecte au service et récupère les données dans un disjoncteur pourrait contribuer à résoudre ce problème et à gérer la défaillance du service plus correctement. Les demandes d'utilisateur échoueront encore, mais elles le feront plus rapidement et les ressources ne seront pas bloquées.

La classe CircuitBreaker conserve les informations d'état sur un disjoncteur dans un objet qui implémente l'interface ICircuitBreakerStateStore indiquée dans le code suivant.

```
interface ICircuitBreakerStateStore
{
  CircuitBreakerStateEnum State { get ; }

  Exception LastException { get; }

  DateTime LastStateChangedDateUtc { get; }

  void Trip(Exception ex);

  void Reset();

  void HalfOpen();

  bool IsClosed { get; }
}
```

La propriété d'état indique l'état actuel du disjoncteur, et sera ouvert, semi-ouvert ou fermé comme défini par l'énumération CircuitBreakerStateEnum. La propriété IsClosed devrait être vraie si le disjoncteur est fermé, mais fausse s'il est ouvert ou semi-ouvert. La méthode de voyage passe l'état du disjoncteur sur ouvert et enregistre l'exception ayant provoqué le changement d'état, ainsi que la date et l'heure auxquelles l'exception s'est produite. Les propriétés LastException et LastStateChangedDateUtc retournent cette information. La méthode de réinitialisation ferme le disjoncteur et la méthode HalfOpen définit le disjoncteur sur semi-ouvert.

La classe InMemoryCircuitBreakerStateStore dans l'exemple contient une implémentation de l'interface ICircuitBreakerStateStore. La classe CircuitBreaker crée une instance de cette classe pour conserver l'état du disjoncteur.
La méthode ExecuteAction dans la classe CircuitBreaker encapsule une opération, spécifiée comme un délégué Action. Si le disjoncteur est fermé, ExecuteAction appelle le délégué Action. Si l'opération échoue, un gestionnaire d'exceptions appelle TrackException, qui définit l'état du disjoncteur sur ouvert. L'exemple de code suivant met en évidence ce flux.

La méthode ExecuteAction dans la classe CircuitBreaker encapsule une opération, spécifiée comme un délégué Action. Si le disjoncteur est fermé, ExecuteAction appelle le délégué Action. Si l'opération échoue, un gestionnaire d'exceptions appelle TrackException, qui définit l'état du disjoncteur sur ouvert. L'exemple de code suivant met en évidence ce flux.

```
public class CircuitBreaker
{
  private readonly ICircuitBreakerStateStore stateStore =
    CircuitBreakerStateStoreFactory.GetCircuitBreakerStateStore();

  private readonly object halfOpenSyncObject = new object ();
  ...
  public bool IsClosed { get { return stateStore.IsClosed; } }

  public bool IsOpen { get { return !IsClosed; } }

  public void ExecuteAction(Action action)
  {
    ...
    if (IsOpen)
    {
      // Le disjoncteur est ouvert.
      ... (voir l'exemple de code ci-dessous pour plus de détails)
```

```
      }

      // Le disjoncteur est fermé, l'action est exécutée.
      try
      {
        action();
      }
      catch (Exception ex)
      {
        // If an exception still occurs here, simply
        // retrip the breaker immediately.
        this.TrackException(ex);

        // Throw the exception so that the caller can tell
        // the type of exception that was thrown.
        throw;
      }
    }

    private void TrackException(Exception ex)
    {
      // For simplicity in this example, open the circuit breaker on the first exception.
      // In reality this would be more complex. A certain type of exception, such as one
      // that indicates a service is offline, might trip the circuit breaker immediately.
      // Alternatively it might count exceptions locally or across multiple instances and
      // use this value over time, or the exception/success ratio based on the exception
      // types, to open the circuit breaker.
      this.stateStore.Trip(ex);
    }
  }
```

L'exemple suivant montre le code (omis dans l'exemple précédent) qui est exécuté si le disjoncteur n'est pas fermé. Il commence par vérifier si le disjoncteur est ouvert depuis plus longtemps que la durée spécifiée dans le champ local OpenToHalfOpenWaitTime de la classe CircuitBreaker. Si tel est le cas, la méthode ExecuteAction définit le disjoncteur comme demi-ouvert, puis tente d'effectuer l'opération spécifiée par le délégué Action.

Si l'opération réussit, le disjoncteur est réinitialisé à l'état fermé. Si l'opération échoue, le disjoncteur est ramené à l'état ouvert et l'heure à laquelle l'exception s'est produite est mise à jour pour que le disjoncteur attende un temps supplémentaire avant d'essayer d'effectuer à nouveau l'opération.

Si le disjoncteur n'est ouvert que depuis peu (moins de la valeur de OpenToHalfOpenWaitTime), la méthode ExecuteAction lève simplement une exception CircuitBreakerOpenException et renvoie l'erreur qui a provoqué le passage du disjoncteur à l'état ouvert.

De plus, elle utilise un verrou pour empêcher le disjoncteur d'essayer d'effectuer des appels simultanés à l'opération lorsqu'il est demi-ouvert. Une tentative simultanée d'appel de l'opération serait traitée comme si le disjoncteur était ouvert et elle échouerait en levant une exception, comme cela est décrit ultérieurement.

```
...
    if (IsOpen)
    {
      // Le disjoncteur est ouvert. Check if the Open timeout has expired.
      // If it has, set the state to HalfOpen. Another approach might be to
      // check for the HalfOpen state that had be set by some other operation.
      if (stateStore.LastStateChangedDateUtc + OpenToHalfOpenWaitTime < DateTime.UtcNow)
      {
        // The Open timeout has expired. Allow one operation to execute. Note that, in
        // this example, the circuit breaker is set to HalfOpen after being
        // in the Open state for some period of time. An alternative would be to set
        // this using some other approach such as a timer, test method, manually, and
        // so on, and check the state here to determine how to handle execution
        // of the action.
        // Limit the number of threads to be executed when the breaker is HalfOpen.
        // An alternative would be to use a more complex approach to determine which
        // threads or how many are allowed to execute, or to execute a simple test
        // method instead.
        bool lockTaken = false;
        try
        {
          Monitor.TryEnter(halfOpenSyncObject, ref lockTaken)
          if (lockTaken)
          {
            // Set the circuit breaker state to HalfOpen.
            stateStore.HalfOpen();

            // Attempt the operation.
            action();

            // If this action succeeds, reset the state and allow other operations.
            // In reality, instead of immediately returning to the Closed state, a counter
            // here would record the number of successful operations and return the
            // circuit breaker to the Closed state only after a specified number succeed.
            this.stateStore.Reset();
            return;
          }
          catch (Exception ex)
          {
            // If there's still an exception, trip the breaker again immediately.
            this.stateStore.Trip(ex);

            // Throw the exception so that the caller knows which exception occurred.
            throw;
          }
          finally
          {
            if (lockTaken)
            {
              Monitor.Exit(halfOpenSyncObject);
            }
          }
        }
      }
      // The Open timeout hasn't yet expired. Throw a CircuitBreakerOpen exception to
      // inform the caller that the call was not actually attempted,
      // and return the most recent exception received.
      throw new CircuitBreakerOpenException(stateStore.LastException);
    }
    ...
```

Pour utiliser un objet CircuitBreaker pour protéger une opération, une application crée une instance de la classe CircuitBreaker et appelle la méthode ExecuteAction en spécifiant comme paramètre l'opération à effectuer. L'application doit être préparée à intercepter l'exception CircuitBreakerOpenException si l'opération échoue car le disjoncteur est ouvert. Le code suivant en illustre un exemple :

```
var breaker = new CircuitBreaker();

try
{
  breaker.ExecuteAction(() =>
  {
    // Operation protected by the circuit breaker.
    ...
  });
}
catch (CircuitBreakerOpenException ex)
{
  // Perform some different action when the breaker is open.
  // Last exception details are in the inner exception.
  ...
}
catch (Exception ex)
{
  ...
}
```

Patrons et informations connexes

Les patrons suivants peuvent également être utiles lors de l'implémentation de ce patron :

- Patron Retry (Nouvelle tentative). Décrit comment une application peut gérer des échecs temporaires anticipés lorsqu'elle essaie de se connecter à un service ou à une ressource réseau, en réessayant de façon transparente d'effectuer une opération qui a échoué.
- Patron Health Endpoint Monitoring (Point de terminaison pour la surveillance de fonctionnement). Un disjoncteur peut être en mesure de tester l'intégrité d'un service en envoyant une demande à un point de terminaison exposé par ce service. Le service doit renvoyer des informations indiquant son statut.

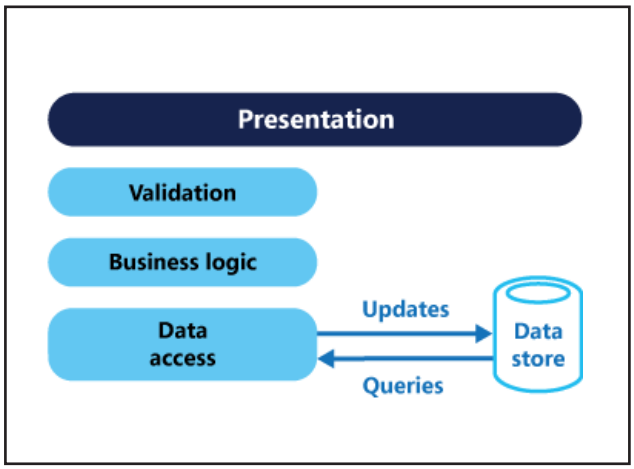
Patron CQRS – Command and Query Responsibility Segregation (Séparation des responsabilités commande / requête)

Séparez les opérations qui lisent des données depuis des opérations mettant à jour des données à l'aide d'interfaces distinctes. Cela peut optimiser les performances, l'évolutivité et la sécurité, ainsi que prendre en charge l'évolution du système au fil du temps, en augmentant la flexibilité. De plus, cela empêche les commandes de mise à jour de générer des conflits de fusion au niveau du domaine.

Contexte et problème

Dans les systèmes de gestion de données traditionnels, les commandes (mises à jour des données) et les requêtes (demandes de données) sont exécutées sur le même ensemble d'entités, dans un référentiel de données unique. Ces entités peuvent être un sous-ensemble des lignes d'une ou plusieurs tables d'une base de données relationnelles, telle qu'une base de données SQL Server.

Généralement, dans ces systèmes, toutes les opérations de création, de lecture, de mise à jour et de suppression (dites opérations CRUD [create, read, update et delete]) sont appliquées à une même représentation de l'entité. Par exemple, un objet DTO (Data Transfer Object) représentant un client est extrait du magasin de données par la couche DAL (Data Access Layer) et affiché à l'écran. Un utilisateur met à jour certains champs de l'objet DTO (éventuellement via une liaison de données), puis la couche DAL réenregistre l'objet DTO dans le magasin de données. Le même objet DTO est utilisé pour les opérations de lecture et d'écriture. La figure ci-dessous illustre une architecture CRUD traditionnelle.



Les conceptions CRUD traditionnelles sont efficaces quand seule une logique métier limitée est appliquée aux opérations de données. Les mécanismes d'échafaudage fournis par les outils de développement peuvent créer très rapidement un code d'accès aux données, qui peut ensuite être personnalisé selon les besoins.

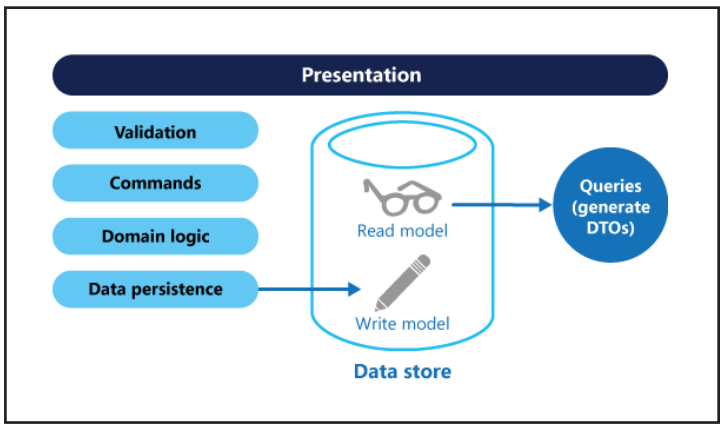
Toutefois, l'approche CRUD traditionnelle présente certains inconvénients :

- Elle dénote souvent des incohérences entre les représentations de lecture et d'écriture des données, telles que des colonnes supplémentaires ou des propriétés devant être mises à jour même si elles ne sont pas requises dans le cadre d'une opération.
- Elle risque de provoquer des contentions de données lorsque des enregistrements sont verrouillés dans le magasin de données, dans un domaine collaboratif, alors que plusieurs acteurs traitent en parallèle le même ensemble de données. Des conflits de mise à jour sont également possibles en raison de mises à jour simultanées, lorsque le verrouillage optimiste est utilisé. Ces risques augmentent parallèlement à la complexité et au débit du système. De plus, l'approche traditionnelle peut avoir un effet négatif sur les performances en raison de la charge qu'elle applique au magasin de données et à la couche d'accès aux données, et de la complexité des requêtes requises pour extraire les informations.
- Elle peut complexifier la gestion de la sécurité et des autorisations parce que chaque entité est soumise à des opérations de lecture et d'écriture, ce qui peut exposer les données dans un contexte inapproprié.

Pour mieux comprendre les limites de l'approche CRUD, consultez CRUD, uniquement quand vous pouvez vous le permettre.

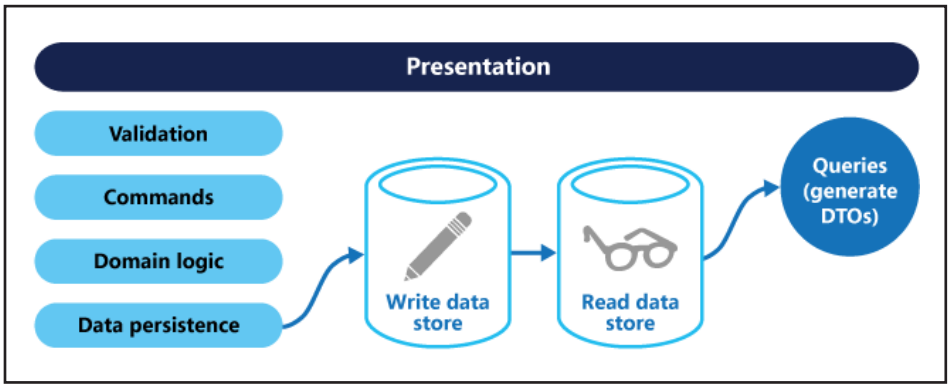
Solution

Le patron CQRS (Command and Query Responsibility Segregation) est un patron qui sépare les opérations qui lisent les données (requêtes) des opérations qui mettent à jour les données (commandes) en utilisant des interfaces distinctes. Cela signifie que les modèles de données utilisés pour effectuer les requêtes et les mises à jour sont différents. Les modèles peuvent ensuite être isolés, comme le montre la figure suivante, mais ce n'est pas une nécessité absolue.



Par rapport au modèle de données unique utilisé dans les systèmes CRUD, l'utilisation de patrons de requête et de mise à jour distincts pour les données dans les systèmes CQRS simplifie la conception et l'implémentation. Toutefois, contrairement aux conceptions CRUD, l'inconvénient réside dans le fait que le code CQRS ne peut pas être généré automatiquement à l'aide de mécanismes d'échafaudage.

Le modèle de requête (pour la lecture des données) et le patron de mise à jour (pour l'écriture des données) peuvent accéder au même magasin physique, peut-être en utilisant des vues SQL ou en générant des projections à la volée. Toutefois, il est courant de séparer les données dans différents magasins physiques, afin d'optimiser les performances, l'évolutivité et la sécurité, comme le montre la figure suivante.



Le magasin de lecture peut être un réplica en lecture seule du magasin d'écriture, ou les magasins de lecture et d'écriture peuvent avoir globalement une structure différente. L'utilisation de plusieurs réplicas en lecture seule du magasin de lecture peut augmenter considérablement les performances des requêtes et la réactivité de l'interface utilisateur de l'application, en particulier dans les scénarios distribués où des réplicas en lecture seule sont situés près des instances de l'application. Certains systèmes de base de données (SQL Server) fournissent des fonctionnalités supplémentaires telles que les réplicas de basculement pour optimiser la disponibilité.

La séparation des magasins de lecture et d'écriture permet également de dimensionner chacun d'eux de façon appropriée à la charge. Par exemple, les magasins de lecture rencontrent généralement une charge beaucoup plus élevée que les magasins d'écriture.

Lorsque le modèle de requête/lecture contient des données dénormalisées (voir le patron Materialized View [Vue matérialisée]), les performances sont optimisées lors de la lecture des données pour chacune des vues dans une application ou lors de la requête de données dans le système.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- La division du magasin de données en magasins physiques distincts pour les opérations de lecture et d'écriture peut augmenter les performances et la sécurité d'un système, mais elle peut augmenter la complexité en termes de résilience et de cohérence à terme. Le magasin de modèles de lecture doit être mis à jour pour refléter les modifications apportées au magasin de modèles d'écriture, et il peut être difficile de détecter qu'un utilisateur a émis une demande basée sur des données lues obsolètes, ce qui signifie que l'opération ne peut pas être terminée.
- Pour obtenir une description de la cohérence à terme, consultez Introduction à la cohérence des données.
- Envisagez d'appliquer le patron CQRS à des sections limitées de votre système où il sera particulièrement efficace.
- Une approche standard du déploiement de la cohérence à terme consiste à utiliser le patron Event Sourcing conjointement à CQRS, afin que le modèle d'écriture soit un flux d'ajout uniquement d'événements entraîné par l'exécution des commandes. Ces événements permettent de mettre à jour les vues matérialisées qui agissent en tant que modèle de lecture. Pour plus d'informations, consultez Event Sourcing et CQRS.

Quand utiliser ce patron

Utilisez ce patron dans les situations suivantes :

- Domaines collaboratifs où plusieurs opérations sont effectuées en parallèle sur les mêmes données. Le patron CQRS permet de définir des commandes avec une granularité suffisante pour minimiser les conflits de fusion au niveau du domaine (tous les conflits qui surviennent peuvent être fusionnés par la commande), même en mettant à jour ce qui semble être le même type de données.
- Interfaces utilisateur basées sur les tâches où les utilisateurs sont guidés via un processus complexe constitué d'une série d'étapes ou avec des modèles de domaine complexes. Également utile pour les équipes déjà familières des techniques de création orientée domaine. Le modèle d'écriture a une pile complète de traitement de commandes qui prend en charge la logique métier, la validation d'entrée et le contrôle métier afin de garantir une cohérence globale constante pour chacun des agrégats (chaque cluster d'objets associés considéré comme une unité de modification des données) dans le patron d'écriture. Le modèle de lecture n'a pas de pile de logique métier ni de contrôle métier, et il renvoie simplement un objet DTO à utiliser dans un patron d'affichage. Le modèle de lecture est cohérent à terme avec le patron d'écriture.
- Scénarios dans lesquels la performance des lectures de données doit être ajustée séparément de la performance des écritures de données, notamment lorsque le rapport lectures/écritures est très élevé et que la mise à l'échelle horizontale est nécessaire. Par exemple, dans de nombreux systèmes, le nombre d'opérations de lecture est de nombreuses fois supérieur au nombre d'opérations d'écriture. Pour tenir compte de cela, envisagez une montée en charge du modèle de lecture tout en exécutant le patron d'écriture sur une seule instance ou sur quelques instances seulement. Un nombre réduit d'instances de modèle d'écriture contribue également à minimiser l'apparition de conflits de fusion.
- Scénarios dans lesquels une équipe de développeurs peut se concentrer sur le modèle de domaine complexe qui fait partie du patron d'écriture, alors qu'une autre équipe peut se concentrer sur le patron de lecture et les interfaces utilisateur.
- Scénarios dans lesquels le système est censé évoluer au fil du temps et peut contenir plusieurs versions du modèle, ou dans lesquels les règles métier changent régulièrement.
- Intégration avec d'autres systèmes, notamment en association avec l'Event Sourcing, dans le cadre duquel la défaillance temporaire d'un sous-système ne doit pas affecter la disponibilité des autres.

Ce patron n'est pas recommandé dans les situations suivantes :

- Lorsque les règles métier ou du domaine sont simples.

- Lorsqu'une interface utilisateur de style CRUD simple et les opérations d'accès aux données connexes sont suffisantes.
- Pour une mise en œuvre sur l'ensemble du système. Il existe des composants spécifiques d'un scénario global de gestion des données où le patron CQRS peut être utile, mais il peut ajouter une complexité notoire et inutile lorsqu'il n'est pas requis.

Event Sourcing et CQRS

Le patron CQRS est souvent utilisé avec le patron Event Sourcing (Matérialisation d'événements). Les systèmes basés sur le patron CQRS utilisent des modèles de données de lecture et d'écriture distincts, tous adaptés à des tâches pertinentes et souvent situés dans des magasins physiquement séparés. Lorsqu'il est utilisé avec le patron Event Sourcing (Matérialisation d'événements), le magasin des événements est le modèle d'écriture et il constitue la source officielle d'informations. Le modèle de lecture d'un système basé sur CQRS fournit des vues matérialisées des données, généralement sous la forme de vues hautement dénormalisées. Ces vues sont adaptées aux interfaces et aux exigences d'affichage de l'application, ce qui contribue à optimiser les performances d'affichage et de requête.

L'utilisation du flux d'événements en tant que magasin d'écriture plutôt que les données réelles à un moment donné évite les conflits de mise à jour sur un agrégat individuel et optimise les performances et l'évolutivité. Les événements peuvent être utilisés pour générer de façon asynchrone des vues matérialisées des données qui sont utilisées pour remplir le magasin de lecture.

Comme le magasin d'événements est la source officielle d'informations, il est possible de supprimer les vues matérialisées et de relire tous les événements passés pour créer une nouvelle représentation de l'état actuel lorsque le système évolue, ou lorsque le modèle de lecture doit changer. Les vues matérialisées constituent un cache durable en lecture seule des données.

Lorsque vous associez le patron CQRS au patron Event Sourcing (Matérialisation d'événements), prenez en compte les éléments suivants :

- Comme avec n'importe quel système dans lequel les magasins de lecture et d'écriture sont séparés, les systèmes basés sur ce patron sont seulement cohérents à terme. Il existe un certain retard entre l'événement qui est généré et la mise à jour du magasin de données.
- Ce patron augmente la complexité car il impose de créer du code pour initier et gérer les événements, ainsi qu'assembler ou mettre à jour les vues appropriées ou les objets requis par les requêtes ou un modèle de lecture. Lorsqu'il est utilisé avec le patron Event Sourcing (Matérialisation d'événements), la complexité du patron CQRS peut compliquer la mise en œuvre et nécessiter une approche différente de la conception des systèmes. Toutefois, l'Event Sourcing peut simplifier la modélisation du domaine et la reconstruction de vues ou la création de nouvelles vues en préservant l'objectif des modifications apportées aux données.
- La création de vues matérialisées en vue de leur utilisation dans le modèle de lecture ou dans des projections des données en relisant et en gérant les événements pour des entités spécifiques ou des collections d'entités peut exiger une durée de traitement et des ressources importantes. Cela est particulièrement vrai si elle nécessite le cumul ou l'analyse des valeurs sur de longues périodes, car cela peut exiger l'examen de tous les événements associés. Résolvez ce problème en mettant en œuvre des captures instantanées des données à intervalles fixes, comme par exemple un décompte global du nombre d'exécutions d'une action spécifique qui ont eu lieu ou l'état actuel d'une entité.

Exemple

Le code suivant montre quelques extraits issus d'un exemple d'implémentation CQRS qui utilise des définitions différentes pour les modèles de lecture et d'écriture. Les interfaces de modèle n'imposent aucunes fonctionnalités des magasins de données sous-jacents. Ils peuvent évoluer et être affinés indépendamment car ces interfaces sont séparées. Le code suivant montre la définition du modèle de lecture.

```
// Interface de requête
namespace ReadModel
{
  public interface ProductsDao
  {
    ProductDisplay FindById(int productId);
    ICollection<ProductDisplay> FindByName(string name);
    ICollection<ProductInventory> FindOutOfStockProducts();
    ICollection<ProductDisplay> FindRelatedProducts(int productId);
  }

  public class ProductDisplay
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal UnitPrice { get; set; }
    public bool IsOutOfStock { get; set; }
    public double UserRating { get; set; }
  }

  public class ProductInventory
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public int CurrentStock { get; set; }
  }
}
```

Le système permet aux utilisateurs d'évaluer les produits. Le code d'application fait cela en utilisant la commande RateProduct illustrée dans le code suivant.

```
public interface ICommand
{
  Guid Id { get; }
}

public class RateProduct : ICommand
{
  public RateProduct()
  {
    this.Id = Guid.NewGuid();
  }
  public Guid Id { get; set; }
  public int ProductId { get; set; }
  public int Rating { get; set; }
  public int UserId {get; set; }
}
```

Le système utilise la classe `ProductsCommandHandler` pour gérer les commandes envoyées par l'application. En général, les clients envoient des commandes au domaine via un système de messagerie tel qu'une file d'attente. Le gestionnaire de commandes accepte ces commandes et appelle les méthodes de l'interface du domaine. La granularité de chaque commande est conçue pour réduire les risques de demandes en conflit. Le code suivant montre une représentation de la classe ProductsCommandHandler.

```
public class ProductsCommandHandler :
    ICommandHandler<AddNewProduct>,
    ICommandHandler<RateProduct>,
    ICommandHandler<AddToInventory>,
    ICommandHandler<ConfirmItemShipped>,
    ICommandHandler<UpdateStockFromInventoryRecount>
{
  private readonly IRepository<Product> repository;

  public ProductsCommandHandler (IRepository<Product> repository)
  {
    this.repository = repository;
  }

  void Handle (AddNewProduct command)
  {
    ...
  }

  void Handle (RateProduct command)
  {
    var product = repository.Find(command.ProductId);
    if (product != null)
    {
      product.RateProduct(command.UserId, command.Rating);
      repository.Save(product);
    }
  }

  void Handle (AddToInventory command)
  {
    ...
  }

  void Handle (ConfirmItemsShipped command)
  {
    ...
  }

  void Handle (UpdateStockFromInventoryRecount command)
  {
    ...
  }
}
```

Le code suivant montre l'interface IProductsDomain issue du modèle d'écriture.

```
public interface IProductsDomain
{
  void AddNewProduct(int id, string name, string description, decimal price);
  void RateProduct(int userId, int rating);
  void AddToInventory(int productId, int quantity);
  void ConfirmItemsShipped(int productId, int quantity);
  void UpdateStockFromInventoryRecount(int productId, int updatedQuantity);
}
```

Notez également que l'interface IProductsDomain contient des méthodes qui ont un sens dans ce domaine. En général, dans un environnement CRUD, ces méthodes auraient des noms génériques, tels que Save ou Update, et auraient un objet DTO comme seul argument. L'approche CQRS peut être conçue pour répondre aux besoins des systèmes de gestion d'entreprise et des stocks de cette organisation.

Patrons et informations connexes

Les informations et les patrons suivants sont utiles lors de l'implémentation de ce patron :

- Pour comparer le patron CQRS à d'autres styles architecturaux, consultez Styles architecturaux et Style d'architecture CQRS.
- Manuel de cohérence de données. Explique les problèmes qui se posent généralement en raison de la cohérence à terme entre les magasins de données de lecture et d'écriture lors de l'utilisation du patron CQRS, et la manière de résoudre ces problèmes.
- Conseils sur le partitionnement des données. Décrit comment les magasins de données de lecture et d'écriture utilisés dans le patron CQRS peuvent être divisés en partitions gérables et accessibles séparément afin d'améliorer l'évolutivité, de réduire les contentions et d'optimiser les performances.
- Patron Event Sourcing (Matérialisation d'événements). Décrit plus en détail comment le patron Event Sourcing peut être utilisé avec le patron CQRS pour simplifier les tâches dans les domaines complexes tout en améliorant les performances, l'évolutivité et la réactivité. Décrit également la façon d'assurer la cohérence des données transactionnelles tout en conservant des pistes d'audit et un historique complets, qui peuvent permettre de compenser les actions.
- Patron Materialized View (Vue matérialisée). Le modèle de lecture d'une implémentation CQRS peut contenir des vues matérialisées des données du patron d'écriture, ou le patron de lecture peut être utilisé pour générer des vues matérialisées.
- Guide des patrons et pratiques Exploration de CQRS. En particulier, la rubrique Présentation du patron CQRS (Command Query Responsibility Segregation) explore le patron et ses cas d'utilisation, et la rubrique Épilogue : Leçons apprises vous aide à comprendre certaines questions qui se posent lors de l'utilisation de ce patron.
- Publication CQRS par Martin Fowler, qui explique les bases du patron et fournit des liens vers d'autres ressources utiles.
- Publications de Greg Young, qui approfondissent de nombreux aspects du patron CQRS.

Patron Compensating Transaction (Transaction de compensation)

Annulez le travail effectué par une série d'étapes, qui définissent ensemble une opération cohérente à terme, si une ou plusieurs des étapes échouent. Des opérations qui suivent le modèle de cohérence à terme figurent généralement dans des applications hébergées sur le cloud qui implémentent des workflows et des processus métier complexes.

Contexte et problème

Les applications qui s'exécutent dans le cloud modifient souvent les données. Ces données peuvent être réparties sur diverses sources de données conservées dans des emplacements géographiques variés. Pour éviter les contentions et améliorer la performance dans un environnement distribué, une application ne doit pas tenter de fournir une forte cohérence transactionnelle. Au contraire, elle doit implémenter la cohérence à terme. Dans ce modèle, une opération métier standard se compose d'une série d'étapes distinctes. Ces étapes sont effectuées, mais la vue d'ensemble de l'état du système peut être incohérent. Toutefois, quand l'opération se termine et que toutes les étapes ont été exécutées, le système doit redevenir cohérent. La rubrique Introduction à la cohérence des données fournit des informations sur les raisons pour lesquelles il est difficile de mettre à l'échelle les transactions distribuées, ainsi que les principes du modèle de cohérence à terme.

Un défi dans le modèle de cohérence à terme tient à la manière de traiter une étape qui a échoué. Dans ce cas, il peut être nécessaire d'annuler tout le travail effectué par les étapes précédentes de l'opération. Toutefois, les données ne peuvent pas simplement être restaurées parce que d'autres instances simultanées de l'application peuvent les avoir changées. Même dans les cas où les données n'ont pas été changées par une instance concurrente, l'annulation d'une étape n'est peut-être pas une simple question de restauration de l'état d'origine. Il peut être nécessaire d'appliquer diverses règles spécifiques au métier (voir le site web de voyage décrit dans la section Exemple).

Si une opération qui implémente la cohérence à terme recouvre plusieurs magasins de données hétérogènes, l'annulation des étapes de l'opération nécessite la visite de chaque magasin de données, tour à tour. Le travail effectué dans chaque magasin de données doit être annulé de façon fiable pour éviter que le système demeure incohérent.

Toutes les données affectées par une opération qui implémente la cohérence à terme ne peuvent pas être conservées dans une base de données. Dans un environnement d'architecture orientée services (SOA), une opération pourrait appeler une action dans un service et entraîner une modification de l'état détenu par ce service. Pour annuler cette opération, cette modification d'état doit également être annulée. Cela peut impliquer d'appeler à nouveau le service et d'effectuer une autre action qui inverse les effets de la première.

Solution

La solution consiste à implémenter une transaction de compensation. Les étapes d'une transaction de compensation doivent annuler les effets de la procédure dans l'opération d'origine. Une transaction de compensation peut ne pas permettre de remplacer simplement l'état actuel par l'état dans lequel le système était au début de l'opération, parce que cette approche pourrait écraser les modifications apportées par d'autres instances concurrentes d'une application. À la place, ce doit être un processus intelligent qui tient compte de tous les travaux effectués par les instances concurrentes. Ce processus est habituellement spécifique à l'application et repose sur la nature du travail effectué par l'opération d'origine.

Une approche courante consiste à utiliser un flux de travail pour implémenter une opération cohérente à terme qui requiert une compensation. Lorsque l'opération d'origine se poursuit, le système enregistre des informations sur chaque étape et la manière dont le travail effectué par cette étape peut être annulé. Si l'opération échoue à un point quelconque, le flux de travail remonte les étapes terminées et effectue le travail qui inverse chaque étape. Notez qu'une transaction de compensation peut ne pas annuler le travail dans l'ordre inverse exact de l'opération d'origine, et qu'il peut s'avérer possible d'effectuer certaines étapes d'annulation en parallèle.

Cette approche est similaire à la stratégie des sagas commentée dans le blog de Clemens Vasters.

Une transaction de compensation est également une opération cohérente à terme et elle pourrait aussi échouer. Le système doit être en mesure de reprendre la transaction de compensation au point de défaillance et de continuer. Il peut être nécessaire de répéter une étape qui a échoué, si bien que les étapes d'une transaction de compensation doivent être définies comme des commandes idempotentes. Pour plus d'informations, consultez Patrons d'idempotence sur le blog de Jonathan Oliver.

Dans certains cas, il peut s'avérer impossible de remonter une étape qui a échoué autrement que par une intervention manuelle. Dans ces situations, le système doit déclencher une alerte et fournir autant d'informations que possible sur la raison de l'échec.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Il peut s'avérer difficile de déterminer quand une étape a échoué dans une opération qui implémente la cohérence à terme. Une étape peut ne pas échouer immédiatement, mais se bloquer. Il peut être nécessaire d'implémenter une certaine forme de délai d'expiration.

La logique de compensation ne se généralise pas facilement. Une transaction de compensation est spécifique à l'application. Elle repose sur le fait que l'application doit avoir suffisamment d'informations pour pouvoir annuler les effets de chaque étape dans une opération qui a échoué.

Vous devez définir les étapes d'une transaction de compensation comme des commandes idempotentes. Cela permet de répéter les étapes si la transaction de compensation échoue elle-même.

L'infrastructure qui gère les étapes de l'opération d'origine, et la transaction de compensation, doivent être résilientes. Elle ne doit pas perdre les informations requises pour compenser une étape défaillante et elle doit être en mesure de surveiller avec fiabilité les progrès de la logique de compensation.

Une transaction de compensation ne rétablit pas nécessairement les données du système à l'état dans lequel elles se trouvaient au début de l'opération d'origine. À la place, elle compense le travail effectué dans les étapes réalisées avec succès avant que l'opération échoue.

Ces étapes dans la transaction de compensation ne doivent pas nécessairement être dans l'ordre inverse exact des étapes de l'opération d'origine. Par exemple, un magasin de données peut être plus sensible aux incohérences qu'un autre, si bien que les étapes qui annulent les modifications apportées à ce magasin doivent figurer en premier dans la transaction de compensation.

Le fait de placer un verrou à court terme basé sur un délai d'expiration sur chaque ressource qui est tenue de terminer une opération, et le fait d'obtenir ces ressources à l'avance peuvent aider à augmenter la probabilité de réussite de l'activité globale. Le travail doit être effectué uniquement après que toutes les ressources ont été acquises. Toutes les actions doivent être finalisées avant l'expiration des verrous.

Envisagez d'utiliser une logique de nouvelles tentatives plus indulgente que d'habitude pour minimiser les échecs qui déclenchent une transaction de compensation. Si une étape échoue dans une opération qui implémente la cohérence à terme, essayez de gérer l'échec comme une exception passagère et répétez l'étape. Arrêtez l'opération et initiez une transaction de compensation seulement si une étape échoue à plusieurs reprises ou de façon irrémédiable.

Bon nombre des défis de l'implémentation d'une transaction de compensation sont les mêmes que ceux afférant à l'implémentation de la cohérence à terme. Pour plus d'informations, consultez la section Considérations relatives à l'implémentation de la cohérence à terme dans la rubrique Introduction à la cohérence des données.

Quand utiliser ce patron

Utilisez ce patron uniquement pour des opérations qui doivent être annulées si elles échouent. Si possible, concevez des solutions pour éviter la complexité d'avoir à exiger des transactions de compensation.

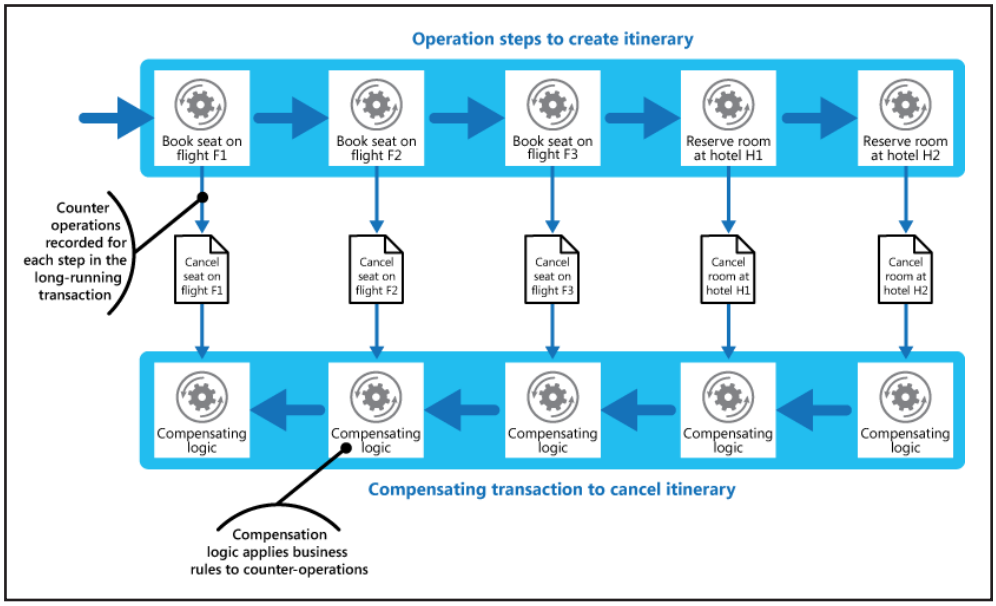
Exemple

Un site de voyage permet à ses clients de réserver des itinéraires. Un itinéraire unique peut comporter une série de vols et d'hôtels. Un client en voyage depuis Seattle jusqu'à Londres puis Paris pourrait effectuer les étapes suivantes lors de la création de son itinéraire de voyage :

1. Réserver un siège sur le vol F1 de Seattle à Londres.
2. Réserver un siège sur le vol F2 de Londres à Paris.
3. Réserver un siège sur le vol F3 de Paris à Londres.
4. Réserver une chambre à l'hôtel H1 à Londres.
5. Réserver une chambre à l'hôtel H2 à Paris.

Ces étapes constituent une opération cohérente à terme, même si chaque étape est une action distincte. Par conséquent, outre la réalisation de ces étapes, le système doit également enregistrer les opérations contraires nécessaires pour annuler chaque étape au cas où le client déciderait d'annuler cet itinéraire. Les étapes nécessaires pour effectuer les opérations contraires peuvent alors s'exécuter comme une transaction de compensation.

Notez que les étapes de la transaction de compensation ne sont pas tenues d'être l'exact opposé des étapes d'origine et la logique inhérente à chaque étape de la transaction de compensation doit prendre en compte toutes les règles spécifiques au métier. Par exemple, l'annulation de la réservation d'un siège sur un vol peut ne pas donner droit au remboursement complet de la somme payée par le client. La figure ci-dessous illustre la création d'une transaction de compensation pour annuler une transaction de longue durée visant à réserver un itinéraire de voyage.



Selon la manière dont vous avez conçu la logique de compensation pour chaque étape, il peut être envisageable d'effectuer en parallèle les étapes de la transaction de compensation.

Dans de nombreuses solutions métier, l'échec d'une seule étape n'exige pas toujours la restauration du système à l'aide d'une transaction de compensation. Par exemple, si, après avoir réservé les vols F1, F2 et F3 dans le scénario du site web de voyage, le client ne parvient pas à réserver une chambre à l'hôtel H1, il est préférable de proposer au client une chambre dans un autre hôtel de la ville plutôt qu'annuler les vols. Le client peut toujours décider d'annuler (auquel cas la transaction de compensation s'exécute et annule les réservations effectuées sur les vols F1, F2 et F3), mais cette décision doit être prise par le client plutôt que par le système.

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Manuel de cohérence de données. Le patron Compensating Transaction (Transaction de compensation) est souvent utilisé pour annuler des opérations qui implémentent le modèle de cohérence à terme. Cette introduction fournit des informations sur les avantages et les inconvénients de la cohérence à terme.
- Patron Scheduler-Agent-Supervisor (Planificateur-agent-superviseur). Décrit comment implémenter des systèmes résilients qui effectuent des opérations métier qui utilisent des ressources et des services distribués. Parfois, il peut être nécessaire d'annuler le travail effectué par une opération en utilisant une transaction de compensation.
- Patron Retry (Nouvelle tentative). Les transactions de compensation peuvent avoir un coût élevé et il est parfois possible de minimiser leur utilisation en implémentant une stratégie efficace visant à réessayer les opérations qui ont échoué en suivant le patron Retry (Nouvelle tentative).

Patron Competing Consumers (Consommateurs concurrents)

Permettez à plusieurs consommateurs simultanés de traiter des messages reçus sur le même canal de messagerie. Cela permet à un système de traiter plusieurs messages simultanément pour optimiser le débit, afin d'améliorer l'évolutivité et la disponibilité, et d'équilibrer la charge de travail.

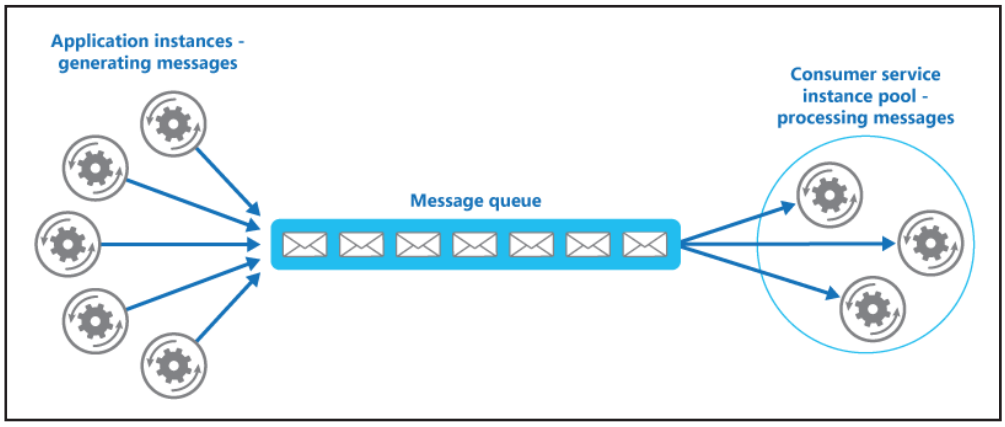
Contexte et problème

Une application s'exécutant dans le cloud doit traiter de nombreuses demandes. Au lieu de traiter chaque demande de façon synchrone, une technique courante consiste à ce que l'application les transmette via un système de messagerie à un autre service (service consommateur) qui les traitera de façon asynchrone. Cette stratégie permet de garantir que la logique métier incluse dans l'application n'est pas bloquée pendant que les demandes sont en cours de traitement.

Le nombre de demandes peut varier considérablement au fil du temps pour de nombreuses raisons. Une augmentation soudaine de l'activité des utilisateurs ou des demandes agrégées provenant de plusieurs locataires peut provoquer une charge de travail inopinée. Aux heures de pointe, un système peut avoir à traiter plusieurs centaines de demandes par seconde, alors qu'à d'autres moments, ce nombre peut être très faible. En outre, la nature du travail exécuté pour traiter ces demandes peut être très variable. Du fait de l'utilisation d'une seule instance du service consommateur, cette instance peut être inondée de demandes ou le système de messagerie peut être surchargé par un afflux de messages en provenance de l'application. Pour gérer cette charge de travail variable, le système peut exécuter plusieurs instances du service consommateur. Toutefois, il convient de coordonner les consommateurs afin de garantir que chaque message soit remis à un seul consommateur. La charge de travail doit également faire l'objet d'un équilibrage entre les consommateurs pour empêcher une instance de devenir un goulot d'étranglement.

Solution

Utilisez une file d'attente de messages pour implémenter le canal de communication entre l'application et les instances du service consommateur. L'application publie des demandes sous la forme de messages dans la file d'attente, et les instances du service consommateur reçoivent ces messages à partir de la file d'attente et les traitent. Cette approche permet au même pool d'instances du service consommateur de gérer les messages provenant de n'importe quelle instance de l'application. La figure suivante illustre l'utilisation d'une file d'attente de messages pour distribuer le travail aux instances d'un service.



Cette solution présente les avantages suivants :

- Elle fournit un système à nivellement de charge, capable de traiter de grandes variations du volume des demandes envoyées par les instances de l'application. La file d'attente sert de tampon entre les instances de l'application et les instances du service consommateur. Cela aide à minimiser l'impact sur la disponibilité et la réactivité de l'application et des instances de service, comme cela est décrit dans l'article Patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente). La gestion d'un message exigeant un traitement de longue durée n'empêche pas le traitement simultané d'autres messages par d'autres instances du service consommateur.
- Elle renforce la fiabilité. Si un producteur communique directement avec un consommateur au lieu d'utiliser ce patron, mais ne surveille pas ce consommateur, il est fort probable que des messages se perdent ou ne soient pas traités en cas d'échec du consommateur. Dans ce patron, les messages ne sont pas envoyés à une instance de service spécifique. Une instance de service ayant échoué ne bloque pas un producteur, et les messages peuvent être traités par n'importe quelle instance de service active.
- Elle ne nécessite pas une coordination complexe entre les consommateurs, ni entre le producteur et les instances de consommateur. La file d'attente de message garantit que chaque message sera remis au moins une fois.
- Elle est dimensionnable. Le système peut dynamiquement augmenter ou diminuer le nombre d'instances du service consommateur en fonction des fluctuations du volume de messages.
- Elle peut améliorer la résilience si la file d'attente de messages fournit des opérations de lecture transactionnelles. Si une instance du service consommateur lit et traite le message dans le cadre d'une opération transactionnelle, et que l'instance du service consommateur échoue, ce patron peut garantir que le message sera renvoyé à la file d'attente afin d'être sélectionné et traité par une autre instance du service consommateur.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- **Classement des messages.** L'ordre dans lequel les instances du service consommateur reçoivent les messages n'est pas garanti et il ne reflète pas nécessairement l'ordre dans lequel les messages ont été créés. Concevez le système pour garantir un traitement idempotent des messages. En effet, cela contribue à éliminer toute dépendance vis-à-vis de l'ordre dans lequel les messages sont traités. Pour plus d'informations, consultez Patrons d'idempotence dans le blog de Jonathan Oliver.

Les files d'attente de Microsoft Azure Service Bus peuvent garantir le classement premier entré, premier sorti des messages à l'aide de sessions de messagerie. Pour plus d'informations, consultez Patrons de messagerie utilisant des sessions.

- **Conception de services en vue de leur résilience.** Si le système est conçu pour détecter et redémarrer les instances de service ayant échoué, il peut s'avérer nécessaire d'implémenter le traitement effectué par les instances du service en tant qu'opérations idempotentes pour minimiser les effets de la récupération et du traitement répétés d'un message.
- **Détection des messages incohérents.** Un message incorrect ou une tâche qui requiert l'accès à des ressources indisponibles peut provoquer l'échec d'une instance de service. Le système doit empêcher le renvoi de tels messages dans la file d'attente et doit à la place capturer et stocker ailleurs les détails de ces messages pour qu'ils puissent être analysés si nécessaire.
- **Gestion des résultats.** L'instance de service qui traite un message est totalement découplée de la logique d'application qui génère le message, et elles ne peuvent peut-être pas communiquer directement. Si l'instance de service génère des résultats qui doivent être soumis de nouveau à la logique d'application, ces informations doivent être stockées dans un emplacement accessible aux deux. Afin d'empêcher la logique d'application de récupérer des données incomplètes, le système doit indiquer la fin du traitement.

  Si vous utilisez Azure, un processus de travail peut transmettre les résultats en retour à la logique d'application en utilisant une file d'attente de réponse dédiée aux messages. La logique d'application doit être en mesure d'établir une corrélation entre ces résultats et le message d'origine. Ce scénario est décrit plus en détail dans la rubrique Introduction à la messagerie asynchrone.
- **Mise à l'échelle du système de messagerie.** Dans une solution à grande échelle, une seule file d'attente de messages peut être submergée par le nombre de messages et devenir un goulot d'étranglement dans le système. Dans ce cas, envisagez le partitionnement du système de messagerie pour envoyer les messages de producteurs spécifiques vers une file d'attente particulière, ou utilisez l'équilibrage de la charge pour distribuer les messages entre plusieurs files d'attente de messages.
- **Garantie de la fiabilité du système de messagerie.** Un système de messagerie fiable est nécessaire pour garantir qu'un message ne se perdra pas une fois que l'application l'aura placé en file d'attente. Ceci est essentiel pour garantir que tous les messages seront remis au moins une fois.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- La charge de travail d'une application est divisée en tâches qui peuvent s'exécuter de manière asynchrone.
- Les tâches sont indépendantes et peuvent s'exécuter en parallèle.
- Le volume de travail est très variable et exige une solution évolutive.
- La solution doit assurer une haute disponibilité et doit être résiliente en cas d'échec du traitement d'une tâche.

Ce patron peut ne pas être approprié quand :

- Il n'est pas aisé de séparer la charge de travail de l'application en tâches discrètes ou les tâches présentent un degré élevé de dépendance.
- Les tâches doivent être effectuées de façon synchrone et la logique de l'application doit attendre la fin d'une tâche avant de continuer.
- Les tâches doivent être effectuées dans un ordre spécifique.

Certains systèmes de messagerie prennent en charge des sessions qui permettent à un producteur de grouper les messages et de s'assurer qu'ils sont tous traités par un même consommateur. Ce mécanisme peut être utilisé avec des messages prioritaires (s'ils sont pris en charge) pour implémenter une forme de classement des messages qui permet de remettre à un consommateur individuel les messages ordonnés d'un producteur.

Exemple

Azure fournit des files d'attente de stockage et des files d'attente Service Bus qui peuvent agir comme un mécanisme pour implémenter ce patron. La logique d'application peut publier les messages dans une file d'attente, et les consommateurs implémentés sous forme de tâches dans un ou plusieurs rôles peuvent récupérer les messages de cette file d'attente et les traiter. Pour assurer la résilience, une file d'attente de Service Bus permet à un consommateur d'utiliser le mode PeekLock lorsqu'il récupère un message à partir de la file d'attente. Ce mode ne supprime pas réellement le message, mais le cache simplement aux autres consommateurs. Le consommateur d'origine peut supprimer le message quand il finit de le traiter. Si le consommateur échoue, le verrouillage expire et le message redevient visible et peut être récupéré par un autre consommateur.

Pour obtenir des informations détaillées sur l'utilisation des files d'attente d'Azure Service Bus, consultez Files d'attente, rubriques et abonnements Service Bus. Pour obtenir des informations sur l'utilisation des files d'attente de stockage Azure, consultez Prise en main du stockage de files d'attente Azure à l'aide de .NET.

Le code ci-dessous, issu de la classe QueueManager dans la solution CompetingConsumers disponible sur GitHub, montre comment créer une file d'attente en utilisant une instance QueueClient dans le gestionnaire d'événements Start, dans un rôle web ou de travail.

```
private string queueName = ...;
private string connectionString = ...;
...

public async Task Start()
{
  // Check if the queue already exists.
  var manager = NamespaceManager.CreateFromConnectionString(this.connectionString);
  if (!manager.QueueExists(this.queueName))
  {
    var queueDescription = new QueueDescription(this.queueName);

    // Set the maximum delivery count for messages in the queue. A message
    // is automatically dead-lettered after this number of deliveries. The
    // default value for dead letter count is 10.
    queueDescription.MaxDeliveryCount = 3;

    await manager.CreateQueueAsync(queueDescription);
  }
  ...

  // Create the queue client. By default the PeekLock method is used.
  this.client = QueueClient.CreateFromConnectionString(
    this.connectionString, this.queueName);
}
```

L'extrait de code suivant montre comment une application peut créer et envoyer un lot de messages à la file d'attente.

```
public async Task SendMessagesAsync()
{
  // Simulate sending a batch of messages to the queue.
  var messages = new List<BrokeredMessage>();

  for (int i = 0; i < 10; i++)
  {
    var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
    messages.Add(message);
  }
  await this.client.SendBatchAsync(messages);
}
```

Le code suivant montre comment une instance de service de consommateur peut recevoir des messages de la file d'attente en suivant une approche basée sur les événements. Le paramètre processMessageTask de la méthode ReceiveMessages est un délégué qui référence le code à exécuter quand un message est reçu. Ce code est exécuté de façon asynchrone.

```
private ManualResetEvent pauseProcessingEvent;
...

public void ReceiveMessages(Func<BrokeredMessage, Task> processMessageTask)
{
  // Set up the options for the message pump.
  var options = new OnMessageOptions();

  // When AutoComplete is disabled it's necessary to manually
  // complete or abandon the messages and handle any errors.
  options.AutoComplete = false;
  options.MaxConcurrentCalls = 10;
  options.ExceptionReceived += this.OptionsOnExceptionReceived;

  // Use of the Service Bus OnMessage message pump.
  // The OnMessage method must be called once, otherwise an exception will occur.
  this.client.OnMessageAsync(
    async (msg) =>
    {
      // Will block the current thread if Stop is called.
      this.pauseProcessingEvent.WaitOne();

      // Execute processing task here.
      await processMessageTask(msg);
    },
    options);
}
...

private void OptionsOnExceptionReceived(object sender,
  ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
  ...
}
```

Notez que des fonctionnalités de mise à l'échelle automatique, telles que celles disponibles dans Azure, peuvent être utilisées pour démarrer et arrêter des instances de rôle lorsque la longueur de la file d'attente varie. Pour plus d'informations, consultez Conseils sur la mise à l'échelle automatique. En outre, il n'est pas nécessaire de maintenir une correspondance un-à-un entre les instances de rôle et les processus de travail . Une instance de rôle unique peut implémenter plusieurs processus de travail. Pour plus d'informations, consultez Patron Compute Resource Consolidation (Consolidation des ressources de calcul).

Patrons et informations connexes

Les informations et les patrons suivants peuvent être pertinents lors de l'implémentation de ce patron :

- Introduction à la messagerie asynchrone. Les files d'attente constituent un mécanisme de communication asynchrone. Si un service consommateur a besoin d'envoyer une réponse à une application, il peut être nécessaire d'implémenter une forme de messagerie de réponses. La rubrique Introduction à la messagerie asynchrone fournit des informations sur la manière d'implémenter une messagerie de demande/réponse à l'aide de files d'attente de messages.
- Conseils sur la mise à l'échelle automatique. Il est parfois possible de démarrer et d'arrêter des instances d'un service consommateur étant donné que la longueur de la file d'attente dans laquelle les applications publient des messages varie. La mise à l'échelle automatique peut aider à maintenir le débit pendant les périodes de pics de traitement.
- Patron Compute Resource Consolidation (Consolidation des ressources de calcul). Il est parfois possible de consolider plusieurs instances d'un service consommateur en un seul processus, afin de réduire les coûts et les charges de gestion. Le patron Compute Resource Consolidation (Consolidation des ressources de calcul) décrit les avantages et les inconvénients de l'adoption de cette approche.
- Patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente). L'introduction d'une file d'attente de messages peut augmenter la résilience du système, en permettant aux instances de service de traiter des volumes très variables de demandes en provenance des instances de l'application. La file d'attente de messages agit comme un tampon, qui nivèle la charge. Le patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente) décrit ce scénario plus en détail.

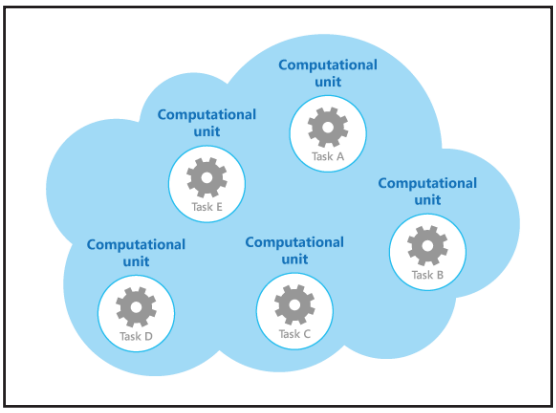
  Ce patron a un exemple d'application qui lui est associé.

Patron Compute Resource Consolidation (Consolidation des ressources de calcul)

Consolidez plusieurs tâches ou opérations en une seule unité de calcul. Cela peut augmenter l'utilisation des ressources de calcul et réduire les coûts et les charges de gestion associés au traitement des calculs dans les applications hébergées dans le cloud.

Contexte et problème

Une application cloud implémente souvent diverses opérations. Dans certaines solutions, il est judicieux de suivre le principe de conception de la séparation initiale des préoccupations et de diviser ces opérations en unités de calcul distinctes, hébergées et déployées individuellement (par exemple, en tant qu'applications web App Service distinctes, machines virtuelles distinctes ou rôles de service cloud distincts). Toutefois, bien que cette stratégie favorise la simplification de la conception logique de la solution, le déploiement d'un grand nombre d'unités de calcul dans le cadre de la même application peut augmenter les coûts d'hébergement au moment de l'exécution et rendre la gestion du système plus complexe.



À titre d'exemple, la figure montre la structure simplifiée d'une solution hébergée dans le cloud qui est implémentée à l'aide de plusieurs unités de calcul. Chaque unité de calcul s'exécute dans son propre environnement virtuel. Toutes les fonctions ont été implémentées en tant que tâches distinctes (étiquetées de Tâche A à Tâche E) qui s'exécutent dans leur propre unité de calcul.

Chaque unité de calcul consomme des ressources payantes, même lorsqu'elle est inactive ou peu utilisée. Par conséquent, ce n'est pas toujours la solution la plus économique.

Dans Azure, cette préoccupation s'applique aux rôles inclus dans un service cloud, dans App Services et dans les machines virtuelles. Ces éléments s'exécutent dans leur propre environnement virtuel. L'exécution d'une collection de rôles distincts, de sites web ou de machines virtuelles qui sont conçus pour effectuer un ensemble d'opérations bien définies, mais qui ont besoin de communiquer et de coopérer dans le cadre d'une solution unique, peut être une utilisation inefficace des ressources.

Solution

Pour favoriser la réduction des coûts, accroître l'utilisation, augmenter la vitesse de communication et réduire l'effort de gestion, il est possible de consolider plusieurs tâches ou opérations en une seule unité de calcul.

Les tâches peuvent être regroupées selon des critères basés sur les fonctionnalités offertes par l'environnement et les coûts associés à ces fonctionnalités. Une approche courante consiste à rechercher les tâches qui ont un profil similaire en matière d'évolutivité, de durée de vie et d'exigences de traitement. Leur regroupement permet de les dimensionner en tant qu'unité. L'élasticité fournie par de nombreux environnements cloud permet de démarrer et d'arrêter des instances supplémentaires d'une unité de calcul pour être en adéquation avec la charge de travail. Par exemple, Azure fournit la mise à l'échelle automatique que vous pouvez appliquer aux rôles dans un service cloud, App Services et les machines virtuelles. Pour plus d'informations, consultez Conseils sur la mise à l'échelle automatique.

Comme contre-exemple illustrant comment l'évolutivité peut servir à déterminer quelles opérations ne doivent pas être regroupées, considérez les deux tâches suivantes :

- La tâche 1 tente d'obtenir les rares messages insensibles au temps qui ont été envoyés vers une file d'attente.
- La tâche 2 traite les pics de débit du trafic réseau.

La seconde tâche requiert une élasticité qui peut impliquer le démarrage et l'arrêt d'un grand nombre d'instances de l'unité de calcul. L'application de la même mise à l'échelle à la première tâche entraînerait simplement davantage de tâches à l'écoute de messages rares sur la même file d'attente et constituerait un gaspillage de ressources.

Dans de nombreux environnements cloud, il est possible de spécifier les ressources à la disposition d'une unité de calcul par un nombre de cœurs de processeur, la mémoire, l'espace disque, etc. Généralement, plus de ressources sont spécifiées, plus le coût est élevé. Pour réduire les coûts, il est important d'optimiser le travail effectué par une unité de calcul onéreuse et de ne pas laisser une telle unité inactive de façon prolongée.

Si des tâches exigent une importante puissance d'UC lors de courts pics d'activité, envisagez de regrouper ces tâches dans une unité de calcul individuelle fournissant la puissance nécessaire. Toutefois, il est important d'équilibrer ce besoin de maintenir occupées les ressources coûteuses avec les contentions susceptibles de survenir dans le cas de contraintes excessives. Par exemple, des tâches de longue durée exigeant des calculs intensifs ne doivent pas figurer dans une même unité de calcul.

Problèmes et considérations

Tenez compte des points suivants lors de l'implémentation de ce patron :

Évolutivité et élasticité. De nombreuses solutions cloud implémentent l'évolutivité et l'élasticité au niveau de l'unité de calcul en démarrant et en arrêtant des instances d'unités. Évitez de regrouper dans une même unité de calcul des tâches soumises à des exigences d'évolutivité contradictoires.

Durée de vie. L'infrastructure cloud recycle périodiquement l'environnement virtuel qui héberge une unité de calcul. Lorsqu'une unité de calcul contient de nombreuses tâches de longue durée, il peut être nécessaire de configurer l'unité pour l'empêcher d'être recyclée jusqu'à ce que ces tâches soient terminées. Vous pouvez également concevoir les tâches en utilisant une approche de pointage qui leur permette d'arrêter proprement et de continuer au point où elles ont été interrompues lorsque l'unité de calcul est redémarrée.

Cadence de publication. Si l'implémentation ou la configuration d'une tâche change fréquemment, il peut être nécessaire d'arrêter l'unité de calcul hébergeant le code mis à jour, de reconfigurer et de redéployer l'unité, puis de la redémarrer. Ce processus exige également l'arrêt, le redéploiement et le redémarrage de toutes les autres tâches au sein de la même unité de calcul.

Sécurité. Les tâches figurant dans une même unité de calcul peuvent partager le même contexte de sécurité et accéder aux mêmes ressources. Il doit exister un degré de confiance élevé entre les tâches, et vous devez être confiant qu'une tâche n'en corrompra ni n'affectera pas une autre. En outre, l'augmentation du nombre de tâches s'exécutant dans une unité de calcul accroît la surface d'attaque de l'unité. Les tâches ne sont pas plus sécurisées que celle présentant le plus de vulnérabilités.

Tolérance aux pannes. Si une tâche d'une unité de calcul échoue ou se comporte de façon anormale, elle peut affecter les autres tâches qui s'exécutent dans cette unité. Par exemple, si une tâche ne peut pas démarrer correctement, elle peut entraîner l'échec de toute la logique de démarrage de l'unité de calcul et empêcher l'exécution d'autres tâches dans la même unité.

Contention. Évitez d'introduire des contentions entre des tâches en concurrence pour des ressources dans une même unité de calcul. Dans l'idéal, des tâches qui partagent la même unité de calcul doivent présenter des caractéristiques différentes d'utilisation des ressources. Par exemple, deux tâches exigeant des calculs intensifs ne devraient probablement pas résider dans la même unité de calcul, de même que deux tâches qui consomment de grandes quantités de mémoire. Toutefois, il est possible de mélanger une tâche exigeant des calculs intensifs avec une tâche nécessitant une grande quantité de mémoire.

Complexité. La combinaison de plusieurs tâches en une seule unité de calcul augmente la complexité du code dans l'unité, ce qui peut compliquer les procédures de test, de débogage et de maintenance.

Architecture logique stable. Concevez et implémentez le code dans chaque tâche afin qu'il n'ait pas besoin de changer, même si l'environnement physique dans lequel la tâche s'exécute change.

Autres stratégies. La consolidation des ressources de calcul ne représente qu'une seule façon d'aider à réduire les coûts associés à l'exécution simultanée de plusieurs tâches. Elle nécessite une planification et une surveillance minutieuses pour rester une approche efficace. D'autres stratégies peuvent être plus appropriées, selon la nature du travail et l'emplacement où se trouvent les utilisateurs de ces tâches. Par exemple, la décomposition fonctionnelle de la charge de travail (telle qu'elle est décrite dans la rubrique Conseils sur le partitionnement des calculs) pourrait être une meilleure option.

Problèmes et considérations

Utilisez ce patron pour des tâches coûteuses si elles s'exécutent dans leurs propres unités de calcul. Si une tâche est inactive une grande partie du temps, l'exécution de cette tâche dans une unité dédiée peut s'avérer coûteuse.

Ce patron peut ne pas convenir pour des tâches qui effectuent des opérations stratégiques à tolérance de panne ou des tâches qui traitent des données hautement sensibles ou privées et exigent leur propre contexte de sécurité. Ces tâches doivent s'exécuter dans leur propre environnement isolé, dans une unité de calcul distincte.

Exemple

Lorsque vous générez un service cloud sur Azure, il est possible de consolider en un seul rôle le traitement effectué par plusieurs tâches. Généralement, un rôle de travail effectue les tâches de traitement asynchrones ou d'arrière-plan.

Dans certains cas, il est possible d'inclure les tâches de traitement asynchrones ou d'arrière-plan dans le rôle web. Cette technique contribue à réduire les coûts et à simplifier le déploiement, même si elle peut affecter l'évolutivité et la réactivité de l'interface publique fournie par le rôle web. L'article Association de plusieurs rôles de travail Azure en un rôle web Azure contient une description détaillée de l'implémentation de tâches de traitement asynchrones ou d'arrière-plan dans un rôle web.

Le rôle est chargé de démarrer et d'arrêter les tâches. Lorsque le contrôleur de structure Azure charge un rôle, il déclenche l'événement de début du rôle. Vous pouvez remplacer la méthode OnStart de la classe WebRole ou WorkerRole pour gérer cet événement, peut-être pour initialiser les données et d'autres ressources dont dépendent les tâches figurant dans cette méthode.

Quand OnStartmethod se termine, le rôle peut commencer à répondre aux demandes. Vous trouverez plus d'informations et de conseils sur l'utilisation des méthodes OnStart et Run dans un rôle dans la section Processus de démarrage d'application du Guide des patrons et pratiques Migration d'applications dans le cloud.

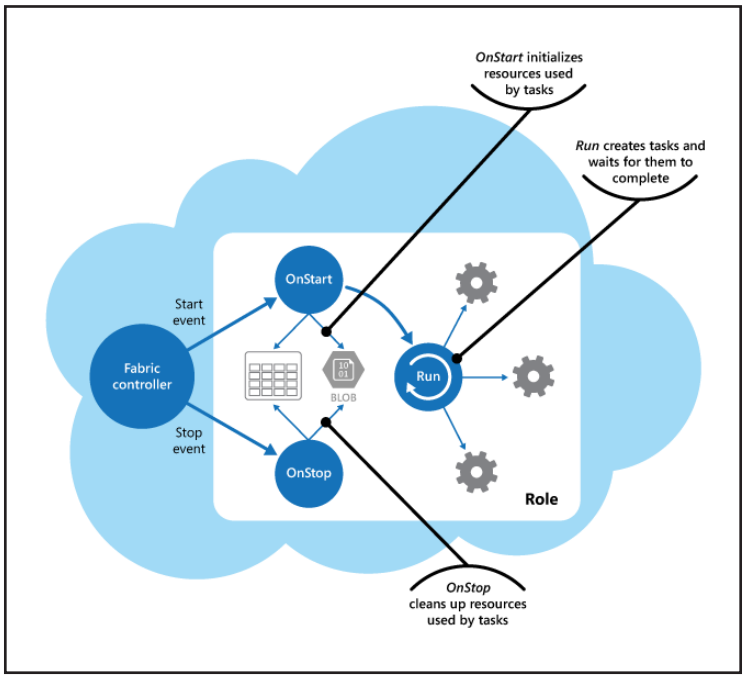
Maintenez le code dans la méthode OnStart aussi concis que possible. Azure n'impose aucune limite sur le temps nécessaire pour terminer cette méthode, mais le rôle ne pourra commencer à répondre aux demandes réseau qui lui sont envoyées qu'une fois cette méthode terminée.

Une fois la méthode OnStart terminée, le rôle exécute la méthode Run. À ce stade, le contrôleur de structure peut commencer à envoyer des demandes au rôle.

Placez le code qui crée effectivement les tâches dans la méthode Run. Notez que la méthode Run définit la durée de vie de l'instance du rôle. Une fois cette méthode terminée, le contrôleur de structure organise l'arrêt du rôle.

Quand un rôle est arrêté ou recyclé, le contrôleur de structure empêche toute autre demande entrante d'être reçue à partir de l'équilibreur de charge et il déclenche l'événement d'arrêt. Vous pouvez capturer cet événement en remplaçant la méthode OnStop du rôle et effectuer tout nettoyage nécessaire avant l'arrêt du rôle.

Toute action effectuée dans la méthode OnStop doit être terminée dans les cinq minutes (ou 30 secondes si vous utilisez l'émulateur Azure sur un ordinateur local). Sinon, le contrôleur de structure Azure considère que le rôle a calé et l'oblige à s'arrêter.
Les tâches sont démarrées par la méthode Run qui attend la fin des tâches. Les tâches implémentent la logique métier du service cloud et peuvent répondre aux messages publiés pour le rôle via l'équilibreur de charge Azure. La figure montre le cycle de vie des tâches et des ressources dans un rôle, dans un service cloud Azure.



Le fichier WorkerRole.cs, dans le projet ComputeResourceConsolidation.Worker, montre un exemple de la manière dont vous pouvez implémenter ce patron dans un service cloud Azure.

Le projet ComputeResourceConsolidation.Worker fait partie de la solution ComputeResourceConsolidation, disponible au téléchargement sur GitHub.

Les méthodes MyWorkerTask1 et MyWorkerTask2 illustrent comment effectuer des tâches différentes dans un même rôle de travail. Le code suivant illustre MyWorkerTask1. Il s'agit d'une tâche simple qui reste inactive pendant 30 secondes, puis renvoie un message de suivi. Elle répète ce processus jusqu'à ce que la tâche soit annulée. Le code dans MyWorkerTask2 est similaire.

```
// A sample worker role task.
private static async Task MyWorkerTask1(CancellationToken ct)
{
  // Fixed interval to wake up and check for work and/or do work.
  var interval = TimeSpan.FromSeconds(30);

  try
  {
    while (!ct.IsCancellationRequested)
    {
      // Wake up and do some background processing if not canceled.
      // TASK PROCESSING CODE HERE
      Trace.TraceInformation("Doing Worker Task 1 Work");

      // Go back to sleep for a period of time unless asked to cancel.
      // Task.Delay will throw an OperationCanceledException when canceled.
      await Task.Delay(interval, ct);
    }
  }
  catch (OperationCanceledException)
  {
    // Expect this exception to be thrown in normal circumstances or check
    // the cancellation token. If the role instances are shutting down, a
    // cancellation request will be signaled.
    Trace.TraceInformation("Stopping service, cancellation requested");

    // Rethrow the exception.
    throw;
  }
}
```

L'exemple de code montre une implémentation courante d'un processus en arrière-plan. Dans une application réelle, vous pouvez suivre cette même structure, mais vous devez placer votre propre logique de traitement dans le corps de la boucle qui attend la demande d'annulation.

Une fois que le rôle de travail a initialisé les ressources qu'il utilise, la méthode Run commence les deux tâches simultanément, comme illustré ici.

```
/// <summary>
/// The cancellation token source use to cooperatively cancel running tasks
/// </summary>
private readonly CancellationTokenSource cts = new CancellationTokenSource();

/// <summary>
/// List of running tasks on the role instance
/// </summary>
private readonly List<Task> tasks = new List<Task>();

// RoleEntry Run() is called after OnStart().
// Returning from Run() will cause a role instance to recycle.
public override void Run()
{
  // Start worker tasks and add to the task list
  tasks.Add(MyWorkerTask1(cts.Token));
  tasks.Add(MyWorkerTask2(cts.Token));

  foreach (var worker in this.workerTasks)
  {
      this.tasks.Add(worker);
  }

  Trace.TraceInformation("Worker host tasks started");
  // The assumption is that all tasks should remain running and not return,
  // similar to role entry Run() behavior.
  try
  {
    Task.WaitAll(tasks.ToArray());
  }
  catch (AggregateException ex)
  {
    Trace.TraceError(ex.Message);

    // If any of the inner exceptions in the aggregate exception
    // are not cancellation exceptions then re-throw the exception.
    ex.Handle(innerEx => (innerEx is OperationCanceledException));
  }

  // If there wasn't a cancellation request, stop all tasks and return from Run()
  // An alternative to canceling and returning when a task exits would be to
  // restart the task.
  if (!cts.IsCancellationRequested)
  {
    Trace.TraceInformation("Task returned without cancellation request");
    Stop(TimeSpan.FromMinutes(5));
  }
}
...
```

Dans cet exemple, la méthode Run attend la fin des tâches. Si une tâche est annulée, la méthode Run suppose que le rôle est en cours d'arrêt et elle attend que les tâches restantes soient annulées avant de se terminer (elle attend un maximum de cinq minutes avant de se terminer). Si une tâche échoue en raison d'une exception attendue, la méthode Run annule cette tâche.

Vous pouvez implémenter des stratégies de surveillance et de gestion des exceptions plus complètes dans la méthode Run, comme le redémarrage de tâches qui ont échoué ou l'insertion d'un code permettant au rôle d'arrêter et de démarrer des tâches individuelles.

La méthode Stop illustrée dans le code suivant est appelée lorsque le contrôleur de structure arrête l'instance de rôle (elle est appelée à partir de la méthode OnStop). Le code arrête chaque tâche correctement en l'annulant. Si une tâche quelconque met plus de cinq minutes à se terminer, le processus d'annulation dans la méthode Stop cesse d'attendre et le rôle est arrêté.

```
// Stop running tasks and wait for tasks to complete before returning
// unless the timeout expires.
private void Stop(TimeSpan timeout)
{
  Trace.TraceInformation("Stop called. Canceling tasks.");
  // Cancel running tasks.
  cts.Cancel();

  Trace.TraceInformation("Waiting for canceled tasks to finish and return");

  // Wait for all the tasks to complete before returning. Note that the
  // emulator currently allows 30 seconds and Azure allows five
  // minutes for processing to complete.
  try
  {
    Task.WaitAll(tasks.ToArray(), timeout);
  }
  catch (AggregateException ex)
  {
    Trace.TraceError(ex.Message);

    // If any of the inner exceptions in the aggregate exception
    // are not cancellation exceptions then rethrow the exception.
    ex.Handle(innerEx => (innerEx is OperationCanceledException));
  }
}
```

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Conseils sur la mise à l'échelle automatique. La mise à l'échelle automatique peut être utilisée pour démarrer et arrêter des instances de service hébergeant des ressources de calcul, en fonction de la demande anticipée de traitement.
- Conseils de partitionnement des calculs. Décrit comment allouer les services et les composants dans un service cloud d'une manière qui contribue à minimiser les coûts d'exploitation tout en préservant l'évolutivité, la performance, la disponibilité et la sécurité du service.
- Ce patron inclut un exemple d'application téléchargeable.

Patron Event Sourcing (Matérialisation d'événements)

Au lieu de stocker uniquement l'état actuel des données dans un domaine, utilisez un magasin « append-only » (prenant en charge uniquement les ajouts) pour enregistrer la série complète des actions effectuées sur ces données. Le magasin agit comme système d'enregistrement et peut être utilisé pour matérialiser les objets du domaine. Cela peut simplifier les tâches dans les domaines complexes, en évitant d'avoir besoin de synchroniser le modèle de données et le domaine métier, tout en améliorant la performance, l'évolutivité et la réactivité. Cela peut également assurer la cohérence des données transactionnelles et permettre de conserver des pistes d'audit et un historique complets, qui peuvent permettre de compenser les actions.

Contexte et problème

La plupart des applications fonctionnent avec des données. L'approche standard est d'avoir une application qui conserve l'état actuel des données en le mettant à jour au fur et à mesure que les utilisateurs utilisent les données. Par exemple, dans le modèle CRUD (create, read, update et delete) traditionnel, un processus de données standard consiste à lire des données dans le magasin, à leur apporter des modifications et à mettre à jour l'état actuel des données avec les nouvelles valeurs (souvent en utilisant des transactions qui verrouillent les données).

L'approche CRUD présente certaines limitations :

- Les systèmes CRUD effectuent des opérations de mise à jour directement sur un magasin de données, ce qui peut ralentir les performances et la réactivité, ainsi que limiter l'évolutivité, en raison des charges de traitement que cela requiert.
- Dans un domaine collaboratif comprenant de nombreux utilisateurs simultanés, les conflits de mise à jour des données sont plus fréquents parce que les opérations de mise à jour interviennent sur un seul élément de données.
- À moins qu'il existe un mécanisme d'audit supplémentaire pour enregistrer les détails de chaque opération dans un journal séparé, l'historique est perdu.

Pour mieux comprendre les limites de l'approche CRUD, consultez CRUD, uniquement quand vous pouvez vous le permettre.

Solution

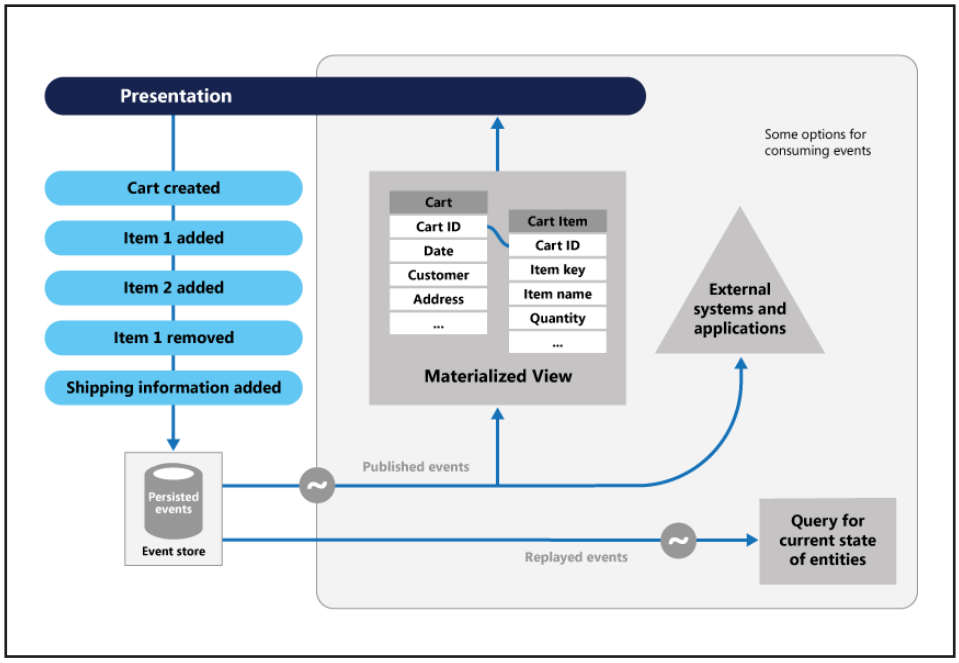
Le patron Event Sourcing (Matérialisation d'événements) définit une approche de la gestion des opérations sur les données qui est pilotée par une séquence d'événements, dont chacun est enregistré dans un magasin « append-only ». Le code de l'application envoie au magasin d'événements une série d'événements qui décrivent impérativement chaque action qui a eu lieu sur les données. Les événements y sont conservés en tant que données persistantes. Chaque événement représente un ensemble de modifications apportées aux données (par exemple, AddedItemToOrder).

Les événements sont conservés comme données persistantes dans un magasin d'événements qui agit comme système d'enregistrement (la source de données faisant autorité) concernant l'état actuel des données. Le magasin d'événements publie généralement ces événements afin que les consommateurs puissent être avertis et puissent les traiter si nécessaire. Les consommateurs peuvent, par exemple, démarrer des tâches qui appliquent les opérations de ces événements à d'autres systèmes, ou effectuer toute autre action associée, nécessaire pour terminer l'opération. Notez que le code d'application qui génère les événements est découplé des systèmes qui s'abonnent aux événements.

Les utilisations standard des événements publiés par le magasin d'événements consistent à maintenir des vues matérialisées des entités, alors que les actions dans l'application les modifient, et en vue de leur l'intégration avec des systèmes externes. Par exemple, un système peut maintenir une vue matérialisée de toutes les commandes client qui sont utilisées pour renseigner des parties de l'interface utilisateur. Lorsque l'application ajoute de nouvelles commandes, ajoute ou supprime des articles dans une commande, ou ajoute les informations d'expédition, les événements qui décrivent ces modifications peuvent être traités et utilisés pour mettre à jour la vue matérialisée.

En outre, les applications peuvent à tout moment accéder à l'historique des événements et l'utiliser pour matérialiser l'état actuel d'une entité par la lecture et la consommation de tous les événements associés à cette entité. Cela peut se produire à la demande pour matérialiser un objet de domaine lors du traitement d'une demande ou via une tâche planifiée de manière à pouvoir stocker l'état de l'entité comme vue matérialisée pour la couche présentation.

La figure présente une vue d'ensemble du patron, notamment certaines options permettant d'utiliser le flux d'événements, comme la création d'une vue matérialisée, l'intégration des événements avec des systèmes et des applications externes et la relecture d'événements pour créer des projections de l'état actuel des entités spécifiques.



Le patron Event Sourcing (Matérialisation d'événements) offre les avantages suivants :

Les événements ne sont pas modifiables et peuvent être stockés à l'aide d'une opération d'ajout uniquement. L'interface utilisateur, le workflow ou le processus qui a déclenché un événement peut continuer, et les tâches qui gèrent les événements peuvent être exécutées en arrière-plan. Si l'on y ajoute l'absence de contention pendant le traitement des transactions, les performances et l'évolutivité des applications sont considérablement améliorées, notamment pour le niveau présentation ou l'interface utilisateur.

Les événements sont de simples objets qui décrivent une action qui s'est produite avec des données connexes requises pour décrire l'action représentée par l'événement. Les événements ne mettent pas à jour directement un magasin de données. Ils sont simplement enregistrés pour être traités au moment opportun. Cela peut simplifier la mise en œuvre et la gestion.

Généralement, les événements ont un sens pour le spécialiste des domaines, tandis que les différences d'impédance au niveau relationnel objet peuvent rendre la compréhension des tables de bases de données plus difficile. Les tables sont des constructions artificielles qui reflètent l'état actuel du système, et non les événements qui ont eu lieu.

Le patron de matérialisation d'événements évite que les mises à jour simultanées ne provoquent des conflits, car il ne rend pas obligatoire la mise à jour des objets directement dans le magasin de données. Toutefois, le modèle de domaine doit être conçu de manière à se protéger contre les demandes susceptibles de provoquer un état incohérent.

Le stockage d'événements de type ajout uniquement fournit une piste d'audit qui peut servir à surveiller les actions prises contre un magasin de données, à régénérer l'état actuel sous la forme de vues matérialisées ou de projections, grâce à la relecture des événements à n'importe quel moment, et à aider à tester et déboguer le système. En outre, l'obligation d'utiliser des événements de compensation pour annuler des modifications permet de fournir un historique des changements annulés, ce qui ne serait pas le cas si le modèle ne stockait que l'état actuel. La liste des événements peut également servir à analyser les performances des applications et à détecter les tendances liées au comportement des utilisateurs ou à obtenir d'autres informations professionnelles utiles.

Le magasin d'événements déclenche des événements et les tâches exécutent des opérations en réponse à ces derniers. Ce découplage entre les tâches et les événements offre une capacité de flexibilité et d'extension. Les tâches connaissent le type d'événement et ses données, mais ignorent l'opération qui a déclenché l'événement. En outre, plusieurs tâches peuvent gérer chaque événement. Cela facilite l'intégration avec d'autres services et systèmes qui sont uniquement à l'écoute des nouveaux événements déclenchés par le magasin d'événements. Cependant, comme le niveau des événements Event Sourcing (matérialisation d'événements) peut être faible, il peut s'avérer nécessaire de générer des événements d'intégration spécifiques.

Le patron de matérialisation d'événements est souvent combiné avec le patron CQRS en réalisant des tâches de gestion de données en réponse aux événements et en matérialisant les vues à partir d'événements stockés.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

La cohérence du système ne sera assurée qu'après la création des vues matérialisées ou la génération de projections de données via la relecture des événements. Il existe un délai plus ou moins long entre l'ajout d'événements par une application dans le magasin d'événements à la suite du traitement d'une demande, les événements en cours de publication et les consommateurs d'événements qui les traitent. Au cours de cette période, de nouveaux événements qui décrivent d'autres modifications apportées aux entités sont peut-être présents dans le magasin d'événements.

> Notes :
>
> Voir la section Introduction à la cohérence des données pour plus d'informations sur la cohérence éventuelle.

Étant donné que le magasin d'événements est la source d'informations permanente, les données d'événement ne doivent jamais être actualisées. La seule façon de mettre à jour une entité dans le but d'annuler une modification consiste à ajouter un événement de compensation dans le magasin d'événements. Si le format (et non les données) des événements persistants est amené à changer, par exemple au cours d'une migration, il peut s'avérer difficile de combiner les événements existants dans le magasin avec la nouvelle version. Il peut être nécessaire d'effectuer une itération sur tous les événements en cours de modification pour les rendre compatibles avec le nouveau format ou d'ajouter de nouveaux événements qui utilisent déjà le nouveau format. Pensez à utiliser une marque de version pour chaque version du schéma d'événement afin de conserver les anciens et les nouveaux formats.

Les applications multithread et plusieurs instances d'applications peuvent stocker des événements dans le magasin. La cohérence des événements dans le magasin d'événements est essentielle, comme l'ordre des événements qui affectent une entité spécifique (l'ordre des changements d'une entité affecte son état actuel). L'ajout d'un horodatage à chaque événement permet d'éviter ces problèmes. Une autre pratique courante consiste à annoter par le biais d'un identificateur incrémentiel chaque événement issu d'une demande. Si deux actions tentent d'ajouter simultanément des événements à la même entité, le magasin d'événements peut rejeter un événement qui correspond à un identificateur d'entité et à un identificateur d'événement existants.

Il n'existe aucune méthode standard, ni mécanisme, par exemple des requêtes SQL, pour lire les événements afin d'obtenir des informations. Seul un flux d'événements peut être extrait à l'aide d'un identificateur d'événement comme critère. L'ID d'événement correspond généralement aux entités individuelles. L'état actuel d'une entité ne peut être déterminé qu'avec la relecture de tous les événements associés, en fonction de l'état d'origine de cette entité.

La longueur de chaque flux d'événements affecte la gestion et la mise à jour du système. Si les flux sont volumineux, pensez à créer des instantanés à intervalles réguliers, par exemple après un certain nombre d'événements. Vous pouvez obtenir l'état actuel de l'entité à partir de l'instantané et de la relecture des événements qui se sont produits à un moment précis. Pour plus d'informations sur la création d'instantanés des données, consultez les pages Instantané et Réplication des instantanés de type maître-subordonné sur le site web Enterprise Application Architecture de Martin Fowler.

Même si le patron de matérialisation d'événements réduit le risque lié aux mises à jour de données conflictuelles, l'application doit toujours être capable de traiter les incohérences qui résultent de la cohérence éventuelle et de l'absence de transactions. Par exemple, un événement qui indique une réduction des stocks peut arriver dans le magasin de données et une commande est passée pour cet article, ce qui entraîne l'obligation de rapprocher ces deux opérations, soit en informant le client, soit en créant une commande différée.

La publication d'événements peut être de type « au moins une fois », et donc les consommateurs d'événements doivent être idempotents. Ils ne doivent pas appliquer de nouveau la mise à jour décrite dans un événement si celui-ci est géré plus d'une fois. Par exemple, si plusieurs instances d'un consommateur gèrent dans une propriété de l'entité un agrégat tel que le volume total des commandes passées, une seule doit permettre d'incrémenter l'agrégat lorsqu'un événement de commande passée est déclenché. Même si ce n'est pas une caractéristique principale du patron de matérialisation d'événements, il s'agit de la décision de mise en œuvre habituelle.

Quand utiliser ce patron

Utilisez ce patron dans les scénarios suivants :

- Vous cherchez à saisir l'intention, le but ou la raison des données. Par exemple, les modifications apportées à une entité client peuvent être capturées sous la forme d'une série de types d'événements spécifiques tels que Déménagement, Compte clôturé ou Décédé.

- Il est essentiel de minimiser ou d'éviter entièrement la présence de mises à jour de données conflictuelles.

- Vous souhaitez enregistrer les événements qui se produisent et être en mesure de les relire pour restaurer l'état d'un système, annuler des modifications ou tenir à jour un registre d'historique et un journal d'audit. Par exemple, lorsqu'une tâche comporte plusieurs étapes, vous pouvez être amené à exécuter des actions pour annuler des mises à jour, puis réexécuter certaines étapes pour restaurer l'état cohérent des données.

- Lorsque l'utilisation d'événements est inhérente au fonctionnement de l'application et nécessite des travaux de développement ou de mise en œuvre supplémentaire.

- Vous avez besoin de dissocier le processus de saisie ou de mise à jour des données des tâches requises pour appliquer ces actions. Il peut s'agir d'améliorer les performances de l'interface utilisateur ou de distribuer les événements à d'autres écouteurs qui déclenchent des actions lorsque les événements se produisent. Par exemple, l'intégration d'un système de paie sur un site web de remboursement des frais, afin que les événements déclenchés par le magasin d'événements, en réponse aux données mises à jour, soient consommés par le site web et le système de paie.

- Vous souhaitez bénéficier d'une certaine souplesse pour modifier le format des modèles matérialisés et des données d'entité en fonction des exigences, ou, conjointement avec le patron CQRS, vous devez adapter un modèle de lecture ou les vues qui exposent les données.

- Vous utilisez ce modèle conjointement avec le modèle CQRS et la cohérence éventuelle est acceptable avec la mise à jour d'un modèle de lecture, ou l'impact sur les performances d'une regénération des entités et des données d'un flux d'événements est acceptable.

Ce patron n'est pas nécessairement utile dans les situations suivantes :

- Domaines petits ou simples, systèmes qui ont peu de logique métier, voir aucune, ou systèmes sans domaine qui fonctionnent bien avec les systèmes de gestion de données traditionnels de création, lecture, mise à jour et suppression.

- Systèmes qui nécessitent de la cohérence et la mise à jour en temps réel des vues des données.

- Systèmes sur lesquels les pistes d'audit, l'historique et les fonctions de restauration et de relecture des actions ne sont pas requis.

- Systèmes sur lesquels la probabilité des mises à jour conflictuelles des données sous-jacentes est très faible. Par exemple, les systèmes qui ajoutent principalement des données au lieu de les mettre à jour.

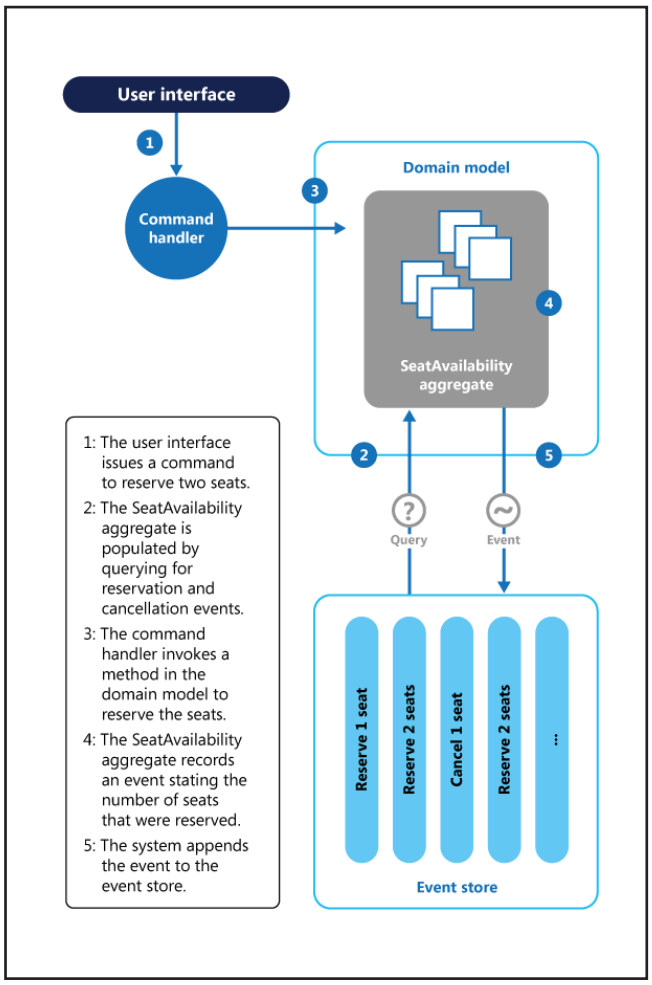
Exemple

Un système de gestion de conférence doit assurer le suivi du nombre de réservations réalisées pour une conférence afin de pouvoir vérifier la disponibilité des places lorsqu'un participant potentiel effectue une réservation. Le système stocke le nombre total de réservations pour une conférence au moins de deux façons :

- Le système stocke les informations sur le nombre total de réservations sous une entité distincte dans une base de données qui contient les informations de réservation. Au fur et à mesure que les réservations sont prises ou annulées, le système augmente ou réduit ce nombre selon le cas. Cette méthode est simple en théorie, mais peut entraîner des problèmes de montée en charge si un grand nombre de participants effectuent une réservation sur une courte période. Par exemple, les derniers jours précédant la clôture de période des réservations.

- Le système peut stocker des informations sur les réservations et les annulations sous forme d'événements dans un magasin d'événements. Il peut alors calculer le nombre de places disponibles en effectuant une relecture de ces événements. Cette méthode est plus évolutive en raison du caractère non modifiable des événements. Le système doit être uniquement capable de lire des données dans le magasin d'événements ou d'y ajouter des données. Les informations sur les réservations et les annulations ne sont jamais modifiées.

Le diagramme suivant illustre la mise en œuvre du sous-système de réservation du système de gestion de conférence à l'aide du patron de matérialisation d'événements.



La séquence des actions de réservation de deux places est la suivante :

1. L'interface utilisateur émet une commande pour réserver des places pour deux participants. La commande est traitée par un gestionnaire de commandes distinct. Un élément de logique dissocié de l'interface utilisateur et responsable du traitement des demandes envoyées comme commandes.
2. Un agrégat contenant des informations sur toutes les réservations de la conférence est construit en interrogeant les événements qui décrivent les réservations et les annulations. Cet agrégat est appelé SeatAvailability et est contenu dans un modèle de domaine qui expose les méthodes d'interrogation et de modification des données dans l'agrégat.

   Certaines optimisations sont à prendre en compte, comme l'utilisation d'instantanés (ainsi, vous n'avez pas besoin d'interroger et de relire la liste complète des événements pour obtenir l'état actuel de l'agrégat) et la gestion d'une copie de l'agrégat mise en cache.
3. Le gestionnaire de commandes appelle une méthode exposée par le modèle de domaine pour effectuer les réservations.
4. L'agrégat SeatAvailability enregistre un événement contenant le nombre de places qui ont été réservées. La prochaine fois que l'agrégat applique les événements, toutes les réservations serviront à calculer le nombre de places restantes.
5. Le système ajoute le nouvel événement à la liste d'événements dans le magasin d'événements.

Si un utilisateur annule une place, le système suit un processus semblable, à la différence près que le gestionnaire de commandes lance une commande qui génère un événement d'annulation de place et l'ajoute au magasin d'événements.

Tout en offrant davantage de possibilités d'évolutivité, l'utilisation d'un magasin d'événements fournit également un historique complet, ou une piste d'audit, des réservations et des annulations d'une conférence. Les événements du magasin sont d'une grande précision. Il est inutile de rendre les agrégats persistants de quelque manière que ce soit parce que le système peut facilement relire les événements et restaurer l'état à un moment précis.

Vous trouverez plus d'informations sur cet exemple sur la page web Présentation du patron Event Sourcing (matérialisation d'événements).

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Patron CQRS - Command and Query Responsibility Segregation (séparation des responsabilités commande / requête). Le magasin d'écriture qui fournit la source permanente d'informations pour une mise en œuvre du patron CQRS repose souvent sur une implémentation du patron Event Sourcing (Matérialisation d'événements). Décrit le mode de séparation entre les opérations qui lisent les données dans une application et les opérations qui mettent à jour les données à l'aide d'interfaces distinctes.
- Patron Materialized View (Vue matérialisée). Le magasin de données utilisé dans un système basé sur le patron de matérialisation d'événements n'est généralement pas bien adapté à l'exécution efficace de requêtes. Une méthodologie courante consiste plutôt à générer des vues de données préremplies à intervalles réguliers ou lors de la mise à jour des données. Montre la manière de procéder.
- Patron Compensating Transaction (Transaction de compensation). Les données contenues dans un magasin de matérialisation d'événements ne sont pas mises à jour. En fait, de nouvelles entrées sont ajoutées, ce qui assure la transition de l'état des entités vers de nouvelles valeurs. Pour annuler une modification, on utilise des entrées de compensation parce qu'il n'est pas possible d'annuler simplement les changements précédents. Décrit la manière d'annuler le travail effectué par une opération précédente.
- Manuel de cohérence de données. Lorsque vous utilisez le patron de matérialisation d'événements avec un magasin de lecture distinct ou des vues matérialisées, l'état des données lues n'est pas immédiatement cohérent ; il le deviendra par la suite. Résume les questions entourant le maintien de la cohérence des données distribuées.
- Conseils sur le partitionnement des données. Souvent, lorsque vous utilisez le patron de matérialisation d'événements, les données sont partitionnées pour améliorer l'évolutivité, réduire les problèmes de contention et optimiser les performances. Décrit la manière de diviser les données en partitions distinctes et les problèmes qui peuvent se poser.
- Article de Greg Young Pourquoi utiliser le patron de matérialisation d'événements ?

Patron External Configuration Store (magasin de configuration externe)

Extrayez les informations de configuration du package de déploiement de l'application vers un emplacement centralisé. Il permet de faciliter la gestion et le contrôle des données de configuration, ainsi que le partage des données de configuration entre les applications et les instances de l'application.

Contexte et problème

La majorité des environnements d'exécution des applications comportent des informations de configuration dans des fichiers déployés avec l'application. Dans certains cas, il est possible de mettre à jour ces fichiers pour modifier le comportement de l'application après le déploiement. Cependant, les modifications apportées à la configuration nécessitent le redéploiement de l'application, ce qui entraîne souvent des interruptions de service inacceptables et d'autres surcharges administratives.

Les fichiers de configuration en local limitent également la configuration à une seule application, alors qu'il serait parfois utile de partager les paramètres de configuration entre plusieurs applications. Citons les chaînes de connexion aux bases de données, les informations de thème de l'interface utilisateur ou les URL des files d'attente et du stockage utilisées par un ensemble associé d'applications.

Il est difficile de gérer les modifications apportées aux configurations locales sur plusieurs instances d'application en cours d'exécution, en particulier dans un scénario d'applications hébergées dans le cloud. Les instances peuvent alors utiliser des paramètres de configuration différents alors que la mise à jour est en cours de déploiement.

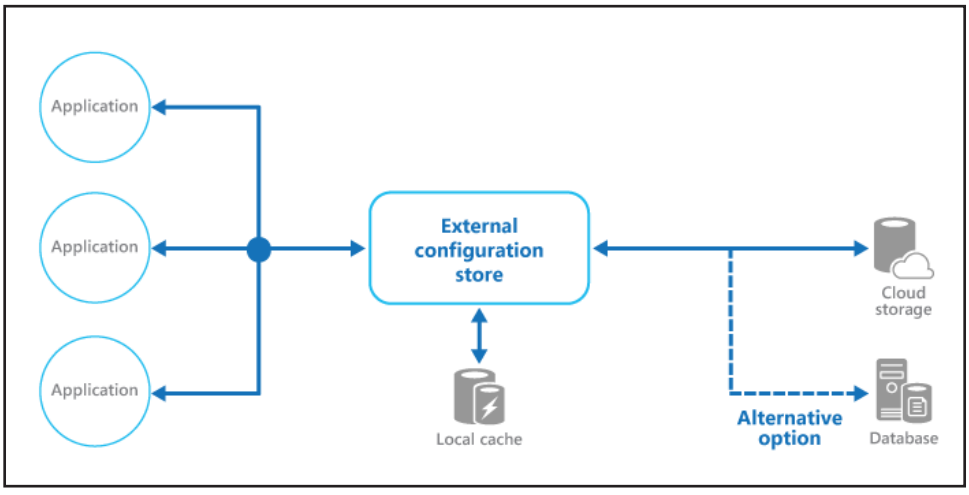
De plus, les mises à jour des applications et des composants peuvent nécessiter la modification des schémas de configuration. Plusieurs systèmes de configuration ne prennent pas en charge plusieurs versions de données de configuration.

Solution

Vous devez stocker les informations de configuration en externe et fournir une interface qui permet de lire et mettre à jour rapidement et efficacement les paramètres de configuration. Le type de magasin externe dépend de l'environnement d'hébergement et d'exécution de l'application. Dans un scénario d'applications hébergées dans le cloud, il s'agit généralement d'un service de stockage dans le cloud, mais cela pourrait être une base de données hébergée ou tout autre système.

Le magasin de sauvegarde que vous choisissez pour les informations de configuration doit posséder une interface qui offre un accès constant et simple d'utilisation. Il doit communiquer les informations dans un format structuré et correctement typé. La mise en œuvre doit également pouvoir autoriser l'accès aux utilisateurs afin de protéger les données de configuration et être suffisamment souple pour permettre le stockage de plusieurs versions de la configuration (par exemple, développement, pré-production ou production, y compris plusieurs versions de chaque).

Plusieurs systèmes de configuration intégrés lisent les données au démarrage de l'application et mettent en mémoire cache les données pour y accéder rapidement et minimiser l'impact sur les performances des applications. En fonction du type de sauvegarde utilisée et du temps de latence de ce magasin, il peut s'avérer utile de mettre en œuvre un système de mise en mémoire cache dans le magasin de configuration externe. Pour plus d'informations, reportez-vous à la section Conseils de mise en cache. La figure présente une vue d'ensemble du patron External Configuration Store (magasin de configuration externe) avec une fonction de mise en cache locale.



Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Choisissez un magasin de stockage qui offre des performances acceptables, une haute disponibilité, de la robustesse et qui peut être sauvegardé dans le cadre des processus de maintenance et d'administration de l'application. Dans une application hébergée dans le cloud, le recours à un stockage dans le cloud est généralement un bon choix pour répondre à ces exigences.

Créez le schéma du magasin de sauvegarde en fournissant une certaine flexibilité aux types d'information qu'il contient. Assurez-vous qu'il couvre tous les aspects des exigences de configuration, notamment les données typées, les ensembles de paramètres, plusieurs versions de paramètres et d'autres fonctionnalités dont les applications ont besoin. L'évolutivité du schéma doit être privilégiée pour prendre en charge des paramètres supplémentaires lorsque les spécifications évoluent.

Évaluez les caractéristiques physiques du magasin de sauvegarde, son impact sur le mode de stockage des informations de configuration et sur les performances. Par exemple, pour le stockage d'un document XML contenant des informations de configuration, l'interface de configuration ou l'application doit analyser le document pour lire les paramètres individuels. Cela rend la mise à jour d'un paramètre plus compliquée, même si la mise en mémoire cache des paramètres peut compenser le ralentissement des performances de lecture.

Examinez comment l'interface de configuration permet d'assurer le contrôle de l'étendue et de l'héritage des paramètres de configuration. Par exemple, il peut être nécessaire d'examiner les paramètres de configuration au niveau de l'organisation, de l'application et de l'ordinateur. Il devra prendre en charge la délégation du contrôle d'accès aux différents champs d'application et l'autorisation ou le rejet des applications individuelles de remplacer les paramètres.

Assurez-vous que l'interface de configuration peut communiquer les données de configuration dans les formats requis, par exemple comme valeurs typées, collections, paires clé/valeur ou conteneurs de propriétés.

Examinez le comportement de l'interface du magasin de configuration lorsque les paramètres contiennent des erreurs ou n'existent pas dans le magasin de sauvegarde. Il conviendra peut-être de restaurer les paramètres par défaut et de consigner les erreurs. En outre, tenez compte de certains aspects tels que le respect de la casse des clés ou des noms des paramètres de configuration, le stockage et le traitement des données binaires et le mode de gestion des valeurs null ou vides.

Étudiez les moyens de protéger les données de configuration pour limiter l'accès aux seuls utilisateurs et applications appropriés. Il s'agit d'une fonctionnalité de l'interface du magasin de configuration, mais il est également nécessaire de s'assurer que les données du magasin de sauvegarde ne sont pas accessibles directement sans autorisation préalable. Assurez-vous d'une séparation stricte entre les autorisations requises pour accéder en lecture et en écriture aux données de configuration. Déterminez également la nécessité de chiffrer tout ou partie des paramètres de configuration et le mode de mise en œuvre dans l'interface du magasin de configuration.

Les configurations stockées de manière centralisée, qui changent le comportement de l'application pendant l'exécution, sont très importantes et doivent être déployées, mises à jour et gérées à l'aide des mêmes mécanismes utilisés pour le déploiement du code de l'application. Par exemple, les changements qui peuvent affecter plusieurs applications doivent être effectués en utilisant une méthodologie de déploiement complète en test et en pré-production pour vous assurer que le changement est adapté à toutes les applications qui utilisent cette configuration. Si un administrateur modifie un paramètre pour mettre à jour une application, cela pourrait nuire aux autres applications qui utilisent le même paramètre.

Si une application met en mémoire cache les informations de configuration, l'application doit être alertée en cas de modification. Il est possible de mettre en œuvre une stratégie d'expiration des données de configuration mises en mémoire cache afin que ces informations soient automatiquement actualisées périodiquement et que toute modification soit détectée (et appliquée).

Quand utiliser ce patron

Ce patron est utile dans les cas suivants :

- Les paramètres de configuration sont partagés entre plusieurs applications et instances d'application, ou une configuration standard doit être appliquée à plusieurs applications et instances d'application.
- Un système de configuration standard ne prend pas en charge tous les paramètres de configuration requis, comme le stockage des images ou les types de données complexes.
- Comme un magasin complémentaire pour certains paramètres des applications permettant aux applications de remplacer tout ou partie des paramètres stockés de manière centralisée.
- Comme moyen de simplifier l'administration d'applications multiples et éventuellement pour surveiller l'utilisation des paramètres de configuration en consignant tout ou partie des types d'accès au magasin de configuration.

Exemple

Dans une application hébergée Microsoft Azure, il s'agit d'un choix typique d'utiliser Azure Storage pour le stockage externe des informations de configuration. Le système est solide et offre des performances élevées et la réplication s'effectue à trois reprises avec fonction de basculement automatique pour fournir un système haute disponibilité. Le stockage Azure Table fournit un magasin de clés/valeurs avec la possibilité d'utiliser un schéma flexible pour les valeurs. Le stockage Azure Blob fournit un conteneur hiérarchique qui peut contenir n'importe quel type de données dans des blobs nommés de manière individuelle.

L'exemple suivant illustre la mise en œuvre d'un magasin de configuration à partir du stockage d'objets blob pour stocker et communiquer les informations de configuration. La classe BlobSettingsStore implémente le stockage d'objets blob pour contenir les informations de configuration et met en œuvre l'interface ISettingsStore illustrée dans l'exemple de code suivant.

```
Public interface ISettingsStore
{
    Task<string> GetVersionAsync();

    Task<Dictionary<string, string>> FindAllAsync();
}
```

Cette interface définit des méthodes d'extraction et de mise à jour des paramètres de configuration contenus dans le magasin de configuration et comprend un numéro de version qui peut servir à détecter une récente modification des paramètres de configuration. La classe BlobSettingsStore utilise la propriété ETag du blob pour implémenter la gestion des versions. La propriété ETag est mise à jour automatiquement chaque fois qu'une opération d'écriture est appliquée à l'objet blob.

Selon la conception de cette solution simple, tous les paramètres de configuration sont transmis sous forme de valeurs de chaîne plutôt que de valeurs typées.

La classe ExternalConfigurationManager fournit un wrapper pour un objet BlobSettingsStore. Une application peut utiliser cette classe pour stocker et récupérer des informations de configuration. Cette classe utilise la bibliothèque Microsoft Reactive Extensions pour exposer toutes les modifications apportées à la configuration grâce à une implémentation de l'interface IObservable. Si un paramètre est modifié en appelant la méthode SetAppSetting, l'événement Changed est déclenché et tous les abonnés à cet événement en sont informés.

Notez que tous les paramètres sont également mis en mémoire cache dans un objet Dictionary à l'intérieur de la classe ExternalConfigurationManager pour accélérer l'accès. La méthode GetSetting utilisée pour récupérer un paramètre de configuration lit les données dans la mémoire cache. Si le paramètre est introuvable dans le cache, il est alors recherché dans l'objet BlobSettingsStore.

La méthode GetSettings appelle la méthode CheckForConfigurationChanges pour vérifier si les informations de configuration du stockage blob ont été modifiées. Cela s'effectue en examinant le numéro de version et en le comparant avec le numéro de version actuel détenu par l'objet ExternalConfigurationManager. Si une ou plusieurs modifications ont été effectuées, l'événement Changed est déclenché et les paramètres de configuration mis en mémoire cache dans l'objet Dictionary sont actualisés. Il s'agit d'une application du patron Cache-Aside (Stockage des données dans le cache).

L'exemple de code suivant illustre le mode d'implémentation de l'événement Changed et des méthodes GetSettings et CheckForConfigurationChanges :

```
public class ExternalConfigurationManager : IDisposable
{
  // An abstraction of the configuration store.
  private readonly ISettingsStore settings;
  private readonly ISubject<KeyValuePair<string, string>> changed;
  ...
  private readonly ReaderWriterLockSlim settingsCacheLock = new ReaderWriterLockSlim();
  private readonly SemaphoreSlim syncCacheSemaphore = new SemaphoreSlim(1);
  ...
  private Dictionary<string, string> settingsCache;
  private string currentVersion;
  ...
  public ExternalConfigurationManager(ISettingsStore settings, ...)
  {
    this.settings = settings;
    ...
  }
  ...
  public IObservable<KeyValuePair<string, string>> Changed => this.changed.AsObservable();
  ...

  public string GetAppSetting(string key)
  {
    ...
    // Try to get the value from the settings cache.
    // If there's a cache miss, get the setting from the settings store and refresh the settings
cache.

    string value;
    try
    {
        this.settingsCacheLock.EnterReadLock();

        this.settingsCache.TryGetValue(key, out value);
    }
    finally
    {
        this.settingsCacheLock.ExitReadLock();
    }

    return value;
  }
  ...
  private void CheckForConfigurationChanges()
  {
    try
    {
        // It is assumed that updates are infrequent.
```

```
        // To avoid race conditions in refreshing the cache, synchronize access to the in-memory
cache.
        await this.syncCacheSemaphore.WaitAsync();

        var latestVersion = await this.settings.GetVersionAsync();

        // If the versions are the same, nothing has changed in the configuration.
        if (this.currentVersion == latestVersion) return;

        // Get the latest settings from the settings store and publish changes.
        var latestSettings = await this.settings.FindAllAsync();

        // Refresh the settings cache.
        try
        {
            this.settingsCacheLock.EnterWriteLock();

            if (this.settingsCache != null)
            {
                //Notify settings changed
                latestSettings.Except(this.settingsCache).ToList().ForEach(kv => this.changed.
OnNext(kv));
            }
            this.settingsCache = latestSettings;
        }
        finally
        {
            this.settingsCacheLock.ExitWriteLock();
        }

        // Update the current version.
        this.currentVersion = latestVersion;
    }
    catch (Exception ex)
    {
        this.changed.OnError(ex);
    }
    finally
    {
        this.syncCacheSemaphore.Release();
    }
  }
}
```

La classe ExternalConfigurationManager fournit également une propriété appelée Environment. Cette propriété prend en charge les différentes configurations d'une application exécutée dans différents environnements tels qu'un environnement de pré-production et de production.

Un objet ExternalConfigurationManager peut également interroger l'objet BlobSettingsStore régulièrement pour détecter tout changement. Dans l'exemple de code suivant, la méthode StartMonitor appelle la méthode CheckForConfigurationChanges à intervalles réguliers pour détecter tout changement et déclencher l'événement Changed, comme décrit précédemment.

```
public class ExternalConfigurationManager : IDisposable
{
  ...
  private readonly ISubject<KeyValuePair<string, string>> changed;
  private Dictionary<string, string> settingsCache;
  private readonly CancellationTokenSource cts = new CancellationTokenSource();
  private Task monitoringTask;
  private readonly TimeSpan interval;

  private readonly SemaphoreSlim timerSemaphore = new SemaphoreSlim(1);
  ...
  public ExternalConfigurationManager(string environment) : this(new
BlobSettingsStore(environment), TimeSpan.FromSeconds(15), environment)
  {
  }

  public ExternalConfigurationManager(ISettingsStore settings, TimeSpan interval, string
environment)
  {
      this.settings = settings;
      this.interval = interval;
      this.CheckForConfigurationChangesAsync().Wait();
      this.changed = new Subject<KeyValuePair<string, string>>();
      this.Environment = environment;
  }
  ...
  /// <summary>
  /// Check to see if the current instance is monitoring for changes
  /// </summary>
  public bool IsMonitoring => this.monitoringTask != null && !this.monitoringTask.IsCompleted;

  /// <summary>
  /// Start the background monitoring for configuration changes in the central store
  /// </summary>
  public void StartMonitor()
  {
      if (this.IsMonitoring)
          return;

      try
      {
          this.timerSemaphore.Wait();

          // Check again to make sure we are not already running.
          if (this.IsMonitoring)
              return;

          // Start running our task loop.
          this.monitoringTask = ConfigChangeMonitor();
      }
      finally
      {
          this.timerSemaphore.Release();
      }
  }

  /// <summary>
  /// Loop that monitors for configuration changes
  /// </summary>
  /// <returns></returns>
  public async Task ConfigChangeMonitor()
  {
      while (!cts.Token.IsCancellationRequested)
      {
```

```
            await this.CheckForConfigurationChangesAsync();
            await Task.Delay(this.interval, cts.Token);
        }
    }

    /// <summary>
    /// Stop monitoring for configuration changes
    /// </summary>
    public void StopMonitor()
    {
        try
        {
            this.timerSemaphore.Wait();

            // Signal the task to stop.
            this.cts.Cancel();

            // Wait for the loop to stop.
            this.monitoringTask.Wait();

            this.monitoringTask = null;
        }
        finally
        {
            this.timerSemaphore.Release();
        }
    }

    public void Dispose()
    {
        this.cts.Cancel();
    }
    ...
}
```

La classe ExternalConfigurationManager est instanciée comme instance singleton de la classe ExternalConfiguration illustrée ci-dessous.

```
public static class ExternalConfiguration
{
    private static readonly Lazy<ExternalConfigurationManager> configuredInstance = new
Lazy<ExternalConfigurationManager>(
        () =>
        {
            var environment = CloudConfigurationManager.GetSetting(“environment”);
            return new ExternalConfigurationManager(environment);
        });

    public static ExternalConfigurationManager Instance => configuredInstance.Value;
}
```

L’exemple de code suivant provient de la classe WorkerRole du projet ExternalConfigurationStore.Cloud. Il montre comment l’application utilise la classe ExternalConfiguration pour lire un paramètre.

```
public override void Run()
{
  // Start monitoring configuration changes.
  ExternalConfiguration.Instance.StartMonitor();

  // Get a setting.
  var setting = ExternalConfiguration.Instance.GetAppSetting("setting1");
  Trace.TraceInformation("Worker Role: Get setting1, value: " + setting);

  this.completeEvent.WaitOne();
}
```

L'exemple de code suivant, également issu de la classe WorkerRole, illustre la manière dont l'application s'abonne aux événements de configuration.

```
public override bool OnStart()
{
  ...
  // Subscribe to the event.
  ExternalConfiguration.Instance.Changed.Subscribe(
     m => Trace.TraceInformation("Configuration has changed. Key:{0} Value:{1}",
          m.Key, m.Value),
     ex => Trace.TraceError("Error detected: " + ex.Message));
  ...
}
```

Patrons et informations connexes

- Un exemple illustrant ce patron est disponible sur GitHub.

Patron Federated Identity (Identité fédérée)

Déléguez l'authentification à un fournisseur d'identité externe. Cela permet de simplifier le développement, de minimiser l'obligation liée à l'administration des utilisateurs et d'améliorer l'expérience utilisateur de l'application.

Contexte et problème

Les utilisateurs doivent généralement utiliser plusieurs applications fournies et hébergées par différentes organisations avec lesquelles ils entretiennent une relation d'affaires. Ces utilisateurs peuvent être amenés à utiliser des informations d'identification spécifiques (et différentes) pour chaque application. Cela peut :

- **Entraîner une expérience utilisateur fragmentée.** Les utilisateurs oublient souvent leurs mots de passe quand ils en possèdent plusieurs.
- **Mettre en évidence des failles de sécurité.** Lorsqu'un utilisateur quitte l'entreprise, son compte doit être immédiatement désactivé. Pourtant, il est facile de l'oublier dans de grandes organisations.
- **Compliquer la gestion des utilisateurs.** Les administrateurs doivent gérer les informations d'identification de tous les utilisateurs et effectuer des tâches supplémentaires, notamment l'envoi de rappels concernant les mots de passe.

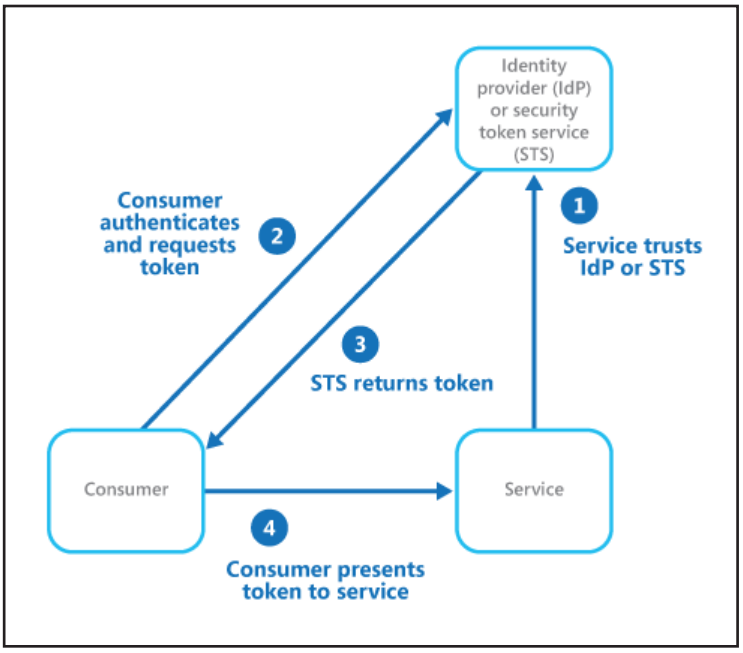
Les utilisateurs préfèrent généralement utiliser les mêmes informations d'identification pour toutes les applications.

Solution

Mettez en place un mécanisme d'authentification basée sur une gestion des identités fédérées. Retirez l'authentification utilisateur dans le code de l'application et déléguez l'authentification à un fournisseur d'identité approuvé. Cela peut simplifier le développement et permettre aux utilisateurs de s'authentifier auprès d'un éventail de fournisseurs d'identité plus large tout en réduisant les tâches d'administration. Cela permet également de dissocier clairement l'authentification de l'autorisation.

Parmi les fournisseurs d'identité approuvés, citons les répertoires d'entreprise, les services de fédération et autres services de jeton de sécurité fournis par des partenaires commerciaux ou les fournisseurs d'identité sociale qui peuvent authentifier les utilisateurs détenant, par exemple, un compte Microsoft, Google, Yahoo! ou Facebook.

La figure illustre le patron de gestion des identités fédérées (Federated Identity) lorsqu'une application cliente doit accéder à un service qui nécessite une authentification. L'authentification est effectuée par un fournisseur d'identité qui travaille en collaboration avec un service de jeton de sécurité. Le fournisseur d'identité émet des jetons de sécurité qui fournissent des informations sur l'utilisateur authentifié. Ces informations, appelées réclamations, incluent l'identité de l'utilisateur, mais aussi d'autres informations comme l'appartenance à des rôles et des droits d'accès plus affinés.



Ce modèle est souvent appelé contrôle d'accès basé sur les revendications. Les applications et les services autorisent l'accès aux fonctionnalités, selon les revendications contenues dans le jeton. Le service qui nécessite une authentification doit faire approuver le fournisseur d'identité. L'application cliente contacte le fournisseur d'identité qui procède à l'authentification. Si l'authentification aboutit, le fournisseur d'identité retourne un jeton contenant les revendications identifiant l'utilisateur auprès du service de jeton de sécurité (notez que le fournisseur d'identité et le service de jeton de sécurité peuvent être le même service). Le service de jeton de sécurité peut transformer et enrichir les revendications dans le jeton en fonction de règles prédéfinies, avant de le renvoyer au client. L'application cliente peut ensuite transmettre ce jeton au service comme preuve de son identité.

D'autres services de jeton de sécurité peuvent se retrouver dans la chaîne d'approbation. Par exemple, dans le scénario décrit plus loin, un service de jeton de sécurité en local approuve un autre service de jeton de sécurité qui est responsable de l'accès à un fournisseur d'identité pour authentifier l'utilisateur. Cette méthode est commune dans les scénarios d'entreprise où l'on retrouve un service de jeton de sécurité sur site et un annuaire.

L'authentification fédérée offre une solution normalisée au problème de confiance des identités entre les différents domaines et peut prendre en charge le mode d'authentification unique. Cela devient de plus en plus courant pour tous les types d'applications, notamment les applications hébergées dans le cloud, en raison de la prise en charge du mode d'authentification unique sans exiger de connexion réseau directe pour identifier les fournisseurs. L'utilisateur n'est pas obligé de saisir ses informations d'identification pour chaque application. La sécurité s'en trouve améliorée parce que cela empêche la création d'informations d'identification pour accéder aux nombreuses applications, et les informations d'identification de l'utilisateur sont tenues secrètes de tous, sauf du fournisseur d'identité d'origine. Les applications n'ont accès qu'aux informations d'identité authentifiées contenues dans le jeton.

La gestion des identités fédérées présente l'avantage principal de transférer la responsabilité de la gestion des identités et des informations d'identification au fournisseur d'identité. L'application ou le service n'a pas besoin de fournir des fonctions de gestion d'identités. En outre, dans une situation d'entreprise, l'annuaire d'entreprise n'a pas besoin de connaître l'utilisateur si le fournisseur d'identité est approuvé. Cela élimine les tâches administratives liées à la gestion de l'identité des utilisateurs dans l'annuaire.

Problèmes et considérations

Suivez les conseils ci-dessous lorsque vous créez des applications implémentant le mode d'authentification fédérée :

- L'authentification peut être un point de défaillance unique. Si vous déployez votre application dans plusieurs centres de données, envisagez le déploiement de la fonction de gestion des identités dans les mêmes centres de données pour garantir la disponibilité et la fiabilité des applications.

- Les outils d'authentification permettent de configurer le contrôle d'accès à partir des revendications de rôle contenues dans le jeton d'authentification. On parle souvent de contrôle d'accès basé sur les rôles (RBAC) qui permet un niveau de contrôle plus affiné sur l'accès aux fonctionnalités et aux ressources.

- Contrairement à un annuaire d'entreprise, le mode d'authentification basée sur les revendications ayant recours aux fournisseurs d'identité sociale ne fournit généralement pas d'informations sur l'utilisateur authentifié, sauf l'adresse e-mail et peut-être le nom. Certains fournisseurs d'identité sociale tels qu'un compte Microsoft, ne fournissent qu'un identificateur unique. Généralement, l'application doit gérer les informations sur les utilisateurs enregistrés et être capable de comparer ces informations à l'identificateur contenu dans les revendications du jeton. En règle générale, cela s'effectue au moment de l'inscription lorsque l'utilisateur accède pour la première fois à l'application, et les informations sont ensuite transmises dans le jeton comme revendications supplémentaires après chaque authentification.

- Si plusieurs fournisseurs d'identité sont configurés pour le service de jeton de sécurité, ce dernier doit identifier le fournisseur d'identité vers lequel l'utilisateur doit être redirigé pour l'authentification. Ce processus est appelé découverte de domaine d'accueil. Le service de jeton de sécurité peut s'en charger automatiquement à partir d'une adresse e-mail ou d'un nom fourni par l'utilisateur, d'un sous-domaine de l'application auquel accède l'utilisateur, de l'étendue de l'adresse IP de l'utilisateur ou du contenu d'un cookie stocké dans le navigateur de l'utilisateur. Si l'utilisateur a entré une adresse e-mail dans le domaine Microsoft, par exemple utilisateur@live.com, le service de jeton de sécurité redirige l'utilisateur vers la page de connexion à un compte Microsoft. Lors des visites ultérieures, le service de jeton de sécurité peut utiliser un cookie pour indiquer que la dernière connexion s'est effectuée avec un compte Microsoft. Si la détection automatique ne peut pas identifier le domaine d'accueil, le service de jeton de sécurité affiche une page de découverte de domaine d'accueil qui répertorie les fournisseurs d'identité approuvés, et l'utilisateur doit alors choisir celui qu'il souhaite utiliser.

Quand utiliser ce patron

Ce patron est utile dans les scénarios suivants :

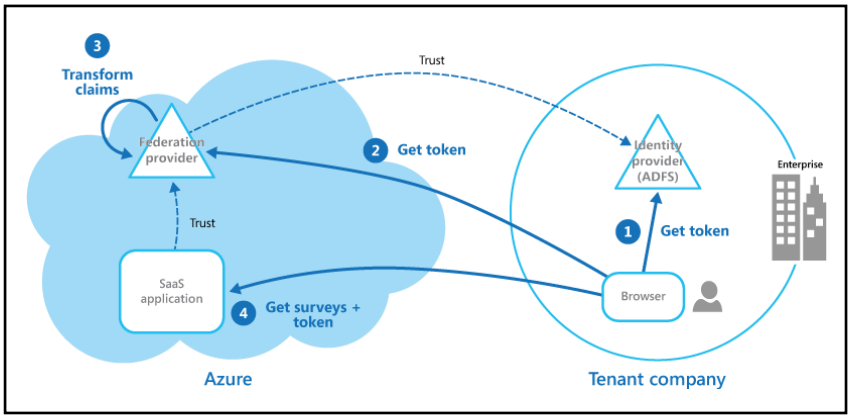
- **Mode d'authentification unique dans l'entreprise.** Dans ce scénario, vous devez authentifier les collaborateurs pour les applications d'entreprise qui sont hébergées dans le cloud à l'extérieur du périmètre de sécurité de l'entreprise, sans les obliger à se connecter chaque fois qu'ils accèdent à l'application. L'expérience utilisateur est identique à celle observée pour les applications en local auprès desquelles les utilisateurs sont authentifiés lorsqu'ils se connectent au réseau d'entreprise et, à partir de là, ont accès à toutes les applications importantes sans avoir besoin de se reconnecter.
- **Gestion des identités fédérées avec plusieurs partenaires.** Dans ce cas de figure, vous devez authentifier les collaborateurs d'une entreprise et les partenaires commerciaux qui ne possèdent pas de compte dans l'annuaire d'entreprise. Cela est fréquent dans les applications B2B (entreprise à entreprise), les applications qui s'intègrent à des services fournis par des tiers et dans les entreprises avec différents systèmes informatiques qui ont fusionné ou qui partagent leurs ressources.
- **Identité fédérée dans des applications SaaS.** Dans ce cas de figure, les éditeurs de logiciels indépendants offrent un service prêt à l'emploi pour plusieurs clients ou locataires. Chaque locataire s'authentifie auprès d'un fournisseur d'identité approprié. Par exemple, les utilisateurs professionnels doivent utiliser leurs informations d'identification d'entreprise, tandis que les consommateurs et les clients du locataire utilisent leurs informations d'identification relatives à leur identité sociale.

Ce patron n'est pas nécessairement utile dans les situations suivantes :

- Tous les utilisateurs de l'application peuvent s'authentifier auprès d'un fournisseur d'identité, et il n'est pas obligatoire de s'authentifier auprès d'un autre fournisseur. Cette situation est courante dans les applications métier qui utilisent un annuaire d'entreprise (accessible dans l'application) pour l'authentification, via un réseau VPN ou (dans le cas d'un hébergement dans le cloud) via une connexion à un réseau virtuel entre l'annuaire local et l'application.
- L'application a été créée au départ avec une méthode d'authentification différente, peut-être avec des magasins d'utilisateurs personnalisés, ou elle ne prend pas en charge les normes de négociation utilisées par les technologies basées sur les revendications. L'intégration d'un mode d'authentification basée sur les revendications et d'un contrôle d'accès dans les applications existantes peut s'avérer compliquée et probablement coûteux.

Exemple

Une organisation héberge un logiciel mutualisée comme application SaaS dans Microsoft Azure. L'application comprend un site web que les locataires peuvent utiliser pour gérer l'application pour leurs utilisateurs. L'application permet aux locataires d'accéder au site web à partir d'une identité fédérée qui est générée par ADFS (Active Directory Federation Services) une fois que l'utilisateur est authentifié via les services Active Directory de cette organisation.



La figure illustre le mode d'authentification des locataires avec leur propre fournisseur d'identité (étape 1), en l'occurrence ADFS. Une fois que l'authentification d'un locataire a abouti, ADFS émet un jeton. Le navigateur client transmet ce jeton au fournisseur de fédération de l'application SaaS, qui approuve les jetons émis par les services ADFS du locataire, afin de récupérer un jeton valide pour le fournisseur de fédération SaaS (étape 2). Si nécessaire, le fournisseur de fédération SaaS transforme les revendications du jeton pour permettre à l'application de les reconnaître (étape 3) avant de renvoyer le nouveau jeton au navigateur client. L'application approuve les jetons émis par le fournisseur de fédération SaaS et utilise les revendications du jeton pour appliquer les règles d'autorisation (étape 4).

Les locataires n'ont pas besoin de mémoriser différents profils d'identification pour accéder à l'application, et un administrateur de l'entreprise du locataire peut configurer dans son propre système ADFS la liste des utilisateurs autorisés à accéder à l'application.

Conseils connexes

- Microsoft Azure Active Directory
- Services de domaine Active Directory
- Services ADFS (Active Directory Federation Services)
- Gestion des identités pour les applications mutualisées dans Microsoft Azure
- Applications mutualisées dans Azure

Patron Gatekeeper (Contrôleur d'accès)

Protégez les applications et les services à l'aide d'une instance de l'hôte dédiée qui sert d'intermédiaire entre les clients et l'application ou le service, valide et assainit les requêtes et transmet des requêtes et des données entre eux. Cela permet d'ajouter un niveau de sécurité supplémentaire et de limiter l'angle d'attaque du système.

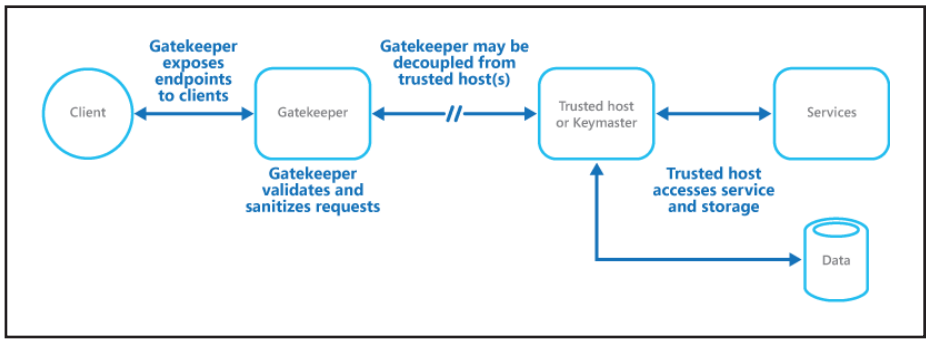
Contexte et problème

Les applications présentent leurs fonctionnalités aux clients par l'acceptation et le traitement de requêtes. Dans un scénario d'hébergement dans le cloud, les applications exposent les points de terminaison auxquels les clients se connectent et généralement cela comporte le code utilisé pour gérer les requêtes des clients. Ce code exécute l'authentification et la validation, tout ou partie du traitement des requêtes et peut accéder au stockage et à d'autres services pour le compte du client.

Si un utilisateur malveillant tente de compromettre le système et accède à l'environnement d'hébergement de l'application, les mécanismes de sécurité utilisés, notamment les informations d'identification et les clés de stockage, ainsi que les services et les données auxquels il accède, sont vulnérables. Par conséquent, l'utilisateur malveillant peut accéder de manière libre aux informations sensibles et à d'autres services.

Solution

Pour minimiser le risque d'accès des clients aux informations sensibles et aux services, vous devez séparer les hôtes ou les tâches qui exposent les points de terminaison publics du code qui traite les demandes et accède au stockage. Vous y parviendrez en utilisant une façade ou une tâche dédiée qui interagit avec les clients, puis transmet la demande (peut-être via une interface découplée) aux hôtes ou aux tâches qui doivent gérer la requête. La figure suivante donne un aperçu général de ce patron.



Le patron Gatekeeper (Contrôleur d'accès) peut être utilisé pour protéger simplement le stockage ou peut servir de solution de façade plus complète pour protéger toutes les fonctions de l'application. Les facteurs importants sont les suivants :

- **Validation contrôlée.** Le contrôleur d'accès valide toutes les requêtes et rejette celles qui ne répondent pas aux critères de validation.
- **Risque et exposition limités.** Le contrôleur d'accès n'a pas accès aux informations d'identification ou aux clés utilisées par l'hôte approuvé pour accéder au stockage et aux services. Si le contrôleur d'accès est compromis, l'utilisateur malveillant ne peut pas accéder à ces informations d'identification ou clés.
- **Sécurité appropriée.** Le contrôleur d'accès est exécuté en mode de privilège limité, tandis que le reste de l'application est exécutée en mode de confiance totale pour l'accès au stockage et aux services. Si le contrôleur d'accès est compromis, il ne peut pas accéder directement aux services ou aux données de l'application.

Ce patron agit comme un pare-feu dans une topographie de réseau classique. Il permet au contrôleur d'accès d'examiner les requêtes et de décider de transmettre la requête à l'hôte approuvé (parfois appelé maître des clés) qui exécute les tâches requises. Pour prendre cette décision, le contrôleur d'accès doit généralement valider et nettoyer le contenu de la requête avant de la transmettre à l'hôte approuvé.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- Assurez-vous que les hôtes approuvés auxquels le contrôleur d'accès transmet les requêtes n'exposent que des points de terminaison internes ou protégés, et qui se connectent uniquement au contrôleur d'accès. Les hôtes approuvés ne doivent pas exposer des points de terminaison externes ou des interfaces.
- Le contrôleur d'accès doit être exécuté en mode de privilège limité. Généralement, cela signifie que le contrôleur d'accès et l'hôte approuvé sont exécutés dans des services hébergés distincts ou sur des machines virtuelles.
- Le contrôleur d'accès ne doit pas exécuter des opérations liées à l'application ou aux services, ou accéder à des données. Sa fonction est limitée à la validation et au nettoyage des requêtes. Les hôtes approuvés peuvent être amenés à procéder à des opérations supplémentaires de validation des requêtes, mais la validation principale doit être effectuée par le contrôleur d'accès.
- Utilisez un canal de communication sécurisé (HTTPS, SSL ou TLS) entre le contrôleur d'accès et les hôtes approuvés ou les tâches, dans la mesure du possible. Toutefois, certains environnements d'hébergement ne prennent pas en charge le protocole HTTPS sur des points de terminaison internes.
  L'ajout de cette couche supplémentaire à l'application dans le but d'implémenter le patron Gatekeeper (Contrôleur d'accès) aura vraisemblablement un impact sur les performances du système en raison des opérations de traitement supplémentaires et de la communication réseau requises.

- L'instance du contrôleur d'accès peut être un point de défaillance unique. Pour minimiser l'impact d'une défaillance, vous pouvez envisager de déployer des instances supplémentaires et d'utiliser une fonction de mise à l'échelle automatique afin de garantir la capacité disponible.

Quand utiliser ce patron

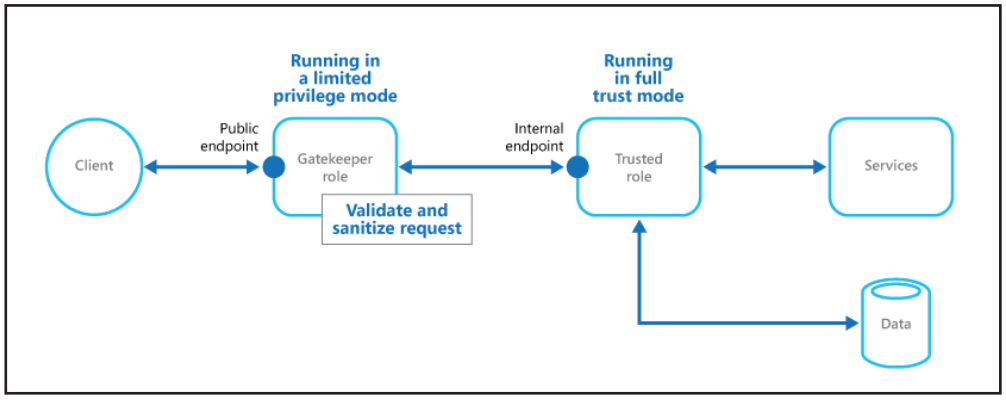
Ce patron est utile dans les cas suivants :

- Les applications qui traitent des informations sensibles exposent des services qui doivent présenter un niveau de protection élevé contre les attaques malveillantes ou exécutent des opérations critiques qui ne doivent pas être interrompues.

- Les applications distribuées pour lesquelles il est nécessaire de procéder à la validation des requêtes séparément des tâches principales, ou de centraliser cette validation pour simplifier la maintenance et l'administration.

Exemple

Dans un scénario d'hébergement dans le cloud, ce patron peut être implémenté en séparant le rôle du contrôleur d'accès ou la machine virtuelle des rôles et services de confiance dans une application. Pour cela, utilisez un point de terminaison interne, une file d'attente ou un emplacement de stockage comme fonction de communication intermédiaire. La figure suivante illustre l'utilisation d'un point de terminaison interne.



Patrons connexes

Le patron Valet Key (Clé à accès restreint) peut être aussi intéressant pour l'implémentation du patron Gatekeeper (Contrôleur d'accès). Pour les communications entre le contrôleur d'accès et les rôles approuvés, il est conseillé de renforcer la sécurité à l'aide de clés ou de jetons qui limitent les autorisations d'accès aux ressources.

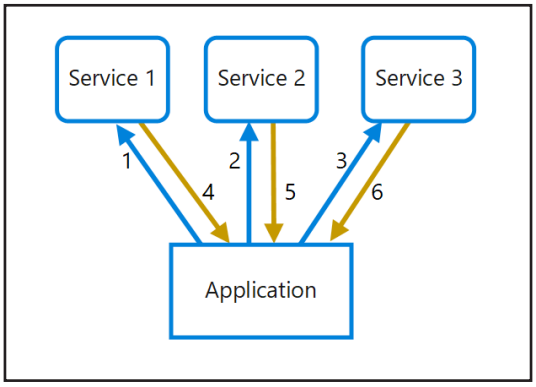
Patron Gateway Aggregation (agrégation de passerelles)

Utilisez une passerelle pour agréger plusieurs requêtes individuelles en une seule requête. Ce patron est utile lorsqu'un client doit effectuer plusieurs appels vers différents systèmes principaux pour réaliser une opération.

Contexte et problème

Pour effectuer une seule tâche, un client peut être amené à effectuer plusieurs appels vers différents services principaux. Une application qui fait appel à de nombreux services pour effectuer une tâche doit affecter des ressources à chaque requête. Lorsqu'une nouvelle fonctionnalité ou un nouveau service est ajouté à l'application, des requêtes supplémentaires sont nécessaires, ce qui augmente le nombre de ressources nécessaires et d'appels réseau. Ces échanges entre un client et un système principal peuvent nuire à la performance et à la taille de l'application. Les architectures de microservices ont répandu ce problème, car les applications développées autour de nombreux petits services multiplient tout naturellement les appels interservices.

Dans le diagramme suivant, le client envoie des requêtes à chaque service (1,2,3). Chaque service traite la requête et renvoie la réponse à l'application (4,5,6). Sur un réseau cellulaire avec une latence généralement élevée, le recours à des requêtes individuelles de cette manière n'est pas efficace et peut même interrompre la connectivité ou rendre les requêtes incomplètes. Alors que chaque requête peut être effectuée en parallèle, l'application doit envoyer, attendre et traiter les données de chaque requête, tout cela sur des connexions distinctes, ce qui augmente le risque de défaillance.

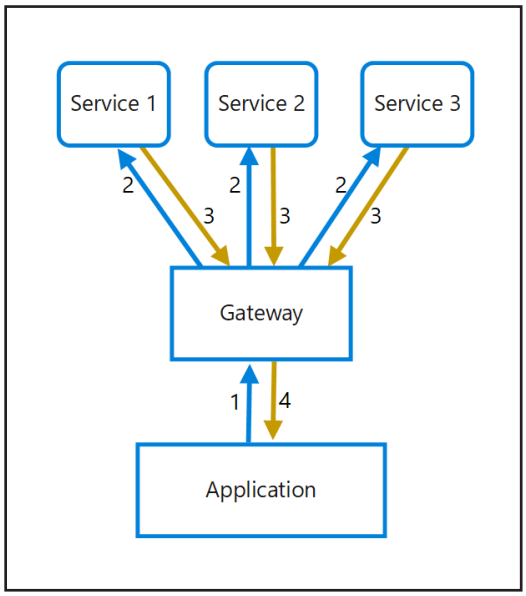


Solution

Utilisez une passerelle pour réduire les échanges entre le client et les services. La passerelle reçoit les requêtes du client, les transmet aux différents systèmes principaux, puis regroupe les résultats et les renvoie au client demandeur.

Ce patron permet de réduire le nombre de requêtes de l'application envoyées aux services principaux et d'améliorer les performances des applications sur les réseaux à latence élevée.

Dans le diagramme suivant, l'application envoie une requête à la passerelle (1). La requête contient un ensemble de requêtes supplémentaires. La passerelle les décompose et traite chaque requête en l'envoyant au service concerné (2). Chaque service renvoie une réponse à la passerelle (3). La passerelle combine les réponses de chaque service et envoie la réponse à l'application (4). L'application effectue une seule requête et reçoit une seule réponse de la passerelle.



Problèmes et considérations

- La passerelle ne doit pas introduire la fonction de couplage de services dans les services principaux.
- La passerelle doit se trouver à proximité des services principaux pour réduire autant que possible les problèmes de latence.
- Le service de passerelle peut introduire un point de défaillance unique. Assurez-vous que la conception de la passerelle répond aux exigences de disponibilité de votre application. La passerelle peut introduire un goulot d'étranglement. Assurez-vous que les performances de la passerelle sont suffisantes pour gérer la charge et qu'elle peut être redimensionnée pour faire face à la croissance prévue.
- Effectuez un test de charge sur la passerelle pour vous assurer de ne pas entraîner une cascade de défaillances des services.
- Mettez en œuvre une conception robuste, en utilisant des patrons tels que Bulkhead (cloison), Circuit Breaker (disjoncteur), Retry (réessayer) et des délais d'expiration.
- Si un ou plusieurs appels de service durent trop longtemps, il peut être judicieux d'appliquer une expiration de délai et de renvoyer un jeu partiel de données. Réfléchissez à la manière dont votre application peut traiter ce scénario.
- Utilisez des E/S asynchrones pour vous assurer qu'un retard au niveau du système principal ne nuit pas aux performances de l'application.
- Mettez en œuvre un système de suivi distribué à l'aide d'ID de corrélation pour suivre chaque appel individuel.
- Surveillez les indicateurs de mesure des requêtes et la taille des réponses.
- Pensez à renvoyer les données en cache dans le cadre d'une stratégie de basculement pour gérer les défaillances.
- Au lieu d'intégrer l'agrégation dans la passerelle, étudiez la possibilité de placer un service d'agrégation derrière la passerelle. L'agrégation des requêtes répond à des critères de ressources différents des autres services de la passerelle et peut avoir un impact sur la fonction de routage et de déchargement de la passerelle.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Un client doit être capable de communiquer avec plusieurs services principaux pour réaliser une opération.
- Le client peut utiliser les réseaux à latence élevée tels que les réseaux cellulaires.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Vous souhaitez réduire le nombre d'appels entre un client et un seul service lors de plusieurs opérations. Dans ce scénario, il est peut-être préférable d'ajouter au service une opération de traitement par lots.
- Le client ou l'application est situé à proximité des services principaux et la latence n'est pas un facteur important.

Exemple

L'exemple suivant illustre la création d'un service NGINX d'agrégation de passerelle simple avec Lua.

```
worker_processes  4;

events {
  worker_connections 1024;
}

http {
  server {
    listen 80;

    location = /batch {
      content_by_lua '
        ngx.req.read_body()

        -- read json body content
        local cjson = require "cjson"
        local batch = cjson.decode(ngx.req.get_body_data())["batch"]

        -- create capture_multi table
        local requests = {}
        for i, item in ipairs(batch) do
          table.insert(requests, {item.relative_url, { method = ngx.HTTP_GET}})
        end

        -- execute batch requests in parallel
        local results = {}
        local resps = { ngx.location.capture_multi(requests) }
        for i, res in ipairs(resps) do
          table.insert(results, {status = res.status, body = cjson.decode(res.body), header =
res.header})
        end

        ngx.say(cjson.encode({results = results}))
      ';
    }

    location = /service1 {
      default_type application/json;
      echo '{"attr1":"val1"}';
    }

    location = /service2 {
      default_type application/json;
      echo '{"attr2":"val2"}';
    }
  }
}
```

Conseils connexes

- Patron Backends for Frontends (Backends pour Frontends)
- Patron Gateway Offloading (déchargement de passerelle)
- Patron Gateway Routing (routage de passerelle)

Patron Gateway Offloading (déchargement de passerelle)

Déchargez la fonctionnalité de service partagé ou spécialisé vers un proxy de passerelle. Ce patron permet de simplifier le développement d'applications en déplaçant dans la passerelle les fonctionnalités de services partagés telles que l'utilisation de certificats SSL, provenant d'autres parties de l'application.

Contexte et problème

Certaines fonctions sont couramment utilisées avec plusieurs services et nécessitent des opérations de configuration, de gestion et de maintenance. Un service partagé ou spécialisé, distribué avec chaque déploiement d'application, augmente les tâches administratives et la probabilité d'erreurs de déploiement. Les mises à jour d'une fonction partagée doivent être déployées à l'ensemble des services qui partagent cette fonction.

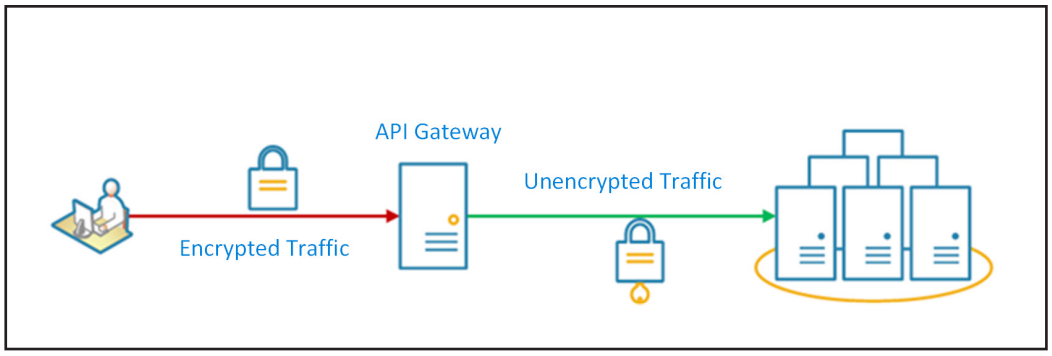
Pour gérer correctement les problèmes de sécurité (validation de jeton, chiffrement, gestion des certificats SSL) et autres tâches complexes, les membres de l'équipe doivent être hautement qualifiés. Par exemple, un certificat requis par une application doit être configuré et déployé sur toutes les instances de l'application. À chaque nouveau déploiement, le certificat doit être traité pour s'assurer qu'il n'expire pas. Tout certificat commun qui arrive à expiration doit être mis à jour, testé et vérifié sur chaque déploiement d'application.

D'autres services communs, comme l'authentification, l'autorisation, la consignation, la surveillance ou la limitation peuvent être difficiles à implémenter et à gérer avec un nombre de déploiements élevé. Il peut être préférable de consolider ce type de fonctionnalité afin de réduire les tâches administratives et les risques d'erreurs.

Solution

Déchargez certaines fonctions dans une passerelle API, notamment pour les questions transversales, comme la gestion des certificats, l'authentification, le processus de terminaison SSL, la surveillance, la traduction de protocole ou la limitation. Déchargez certaines fonctions dans une passerelle API, notamment pour les questions transversales, comme la gestion des certificats, l'authentification, le processus de terminaison SSL, la surveillance, la traduction de protocole ou la limitation.

Le diagramme suivant illustre une passerelle API qui bloque les connexions SSL entrantes. Elle réclame des données pour le compte du demandeur initial à partir de n'importe quel serveur HTTP situé en amont de la passerelle API.



Avantages de ce patron :

- Simplifie le développement de services en supprimant la nécessité de distribuer et de gérer les ressources de prise en charge telles que les certificats du serveur web et la configuration des sites web sécurisés. Une configuration simplifiée se traduit par une facilité de gestion et d'évolutivité qui simplifie les mises à jour des services.

- Permet à des équipes dédiées d'implémenter des fonctions qui nécessitent des compétences spécialisées telles que la sécurité. Votre équipe principale peut ainsi se concentrer sur la fonctionnalité de l'application et laisser le soin aux experts concernés de traiter ces questions spécialisées mais partagées.

- Offre une certaine cohérence des requêtes, de la journalisation des réponses et de la surveillance. Même si un service n'est pas correctement activé, la passerelle peut être configurée pour assurer un niveau minimal de surveillance et d'enregistrement.

Problèmes et considérations

- Assurez-vous que la passerelle API présente une haute disponibilité et résiste aux défaillances. Évitez les points de défaillance uniques en exécutant plusieurs instances de votre passerelle API.

- Assurez-vous que la passerelle répond aux critères de capacité et de mise à l'échelle de votre application et points de terminaison. Évitez que la passerelle ne devienne un goulot d'étranglement pour l'application et assurez-vous qu'elle est suffisamment extensible.

- Ne déchargez que les fonctions utilisées par l'ensemble de l'application, notamment la sécurité ou le transfert des données.

- La logique métier ne doit jamais être déchargée sur la passerelle API.

- Si vous devez effectuer le suivi des transactions, pensez à générer des ID de corrélation à des fins de journalisation.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Un déploiement d'application partage la même préoccupation, celle relative aux certificats SSL ou au chiffrement.

- Fonctionnalité commune aux déploiements d'applications dont les ressources requises peuvent varier, par exemple les ressources de mémoire, la capacité de stockage ou les connexions réseau.

- Vous souhaitez confier la responsabilité des questions de sécurité des réseaux, de limitation ou autres problèmes de limites des réseaux, à une équipe plus spécialisée.

Ce patron ne conviendra pas nécessairement s'il introduit le couplage entre les services.

Exemple

En utilisant Nginx comme application de déchargement SSL, la configuration suivante bloque une connexion SSL entrante et distribue les demandes de connexion à l'un des trois serveurs HTTP situés en amont.

```
upstream iis {
        server  10.3.0.10    max_fails=3    fail_timeout=15s;
        server  10.3.0.20    max_fails=3    fail_timeout=15s;
        server  10.3.0.30    max_fails=3    fail_timeout=15s;
}

server {
        listen 443;
        ssl on;
        ssl_certificate /etc/nginx/ssl/domain.cer;
        ssl_certificate_key /etc/nginx/ssl/domain.key;

        location / {
                set $targ iis;
                proxy_pass http://$targ;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header X-Forwarded-Proto https;
proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header Host $host;
        }
}
```

Conseils connexes

- Patron Backends for Frontends (Backends pour Frontends)
- Patron Gateway Aggregation (agrégation de passerelles)
- Patron Gateway Routing (routage de passerelle)

Patron Gateway Routing (routage de passerelle)

Acheminez les requêtes vers plusieurs services à l'aide d'un point de terminaison unique. Ce patron est utile pour exposer plusieurs services sur un seul point de terminaison et définir l'acheminement vers le service approprié en fonction de la requête.

Contexte et problème

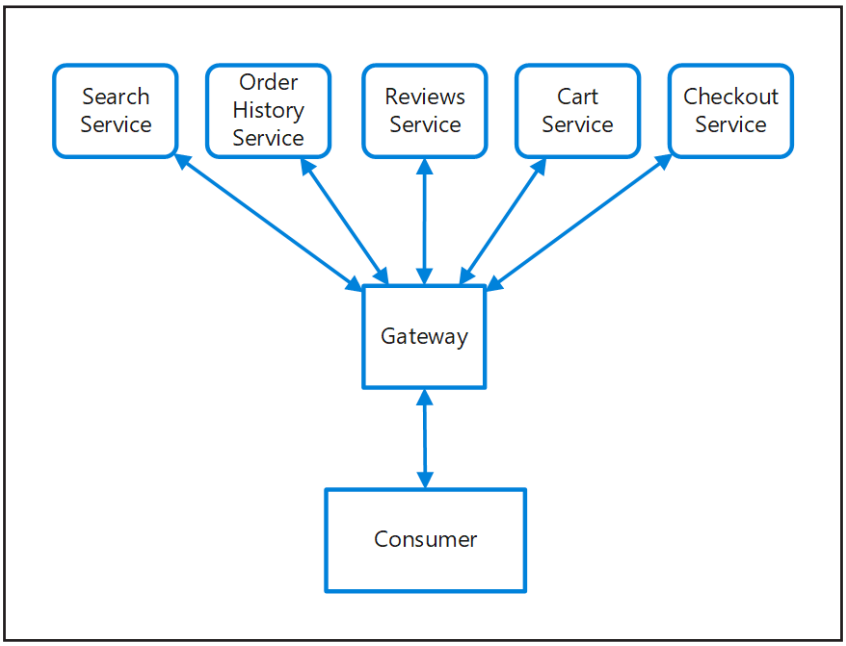
Si un client a besoin de consommer de multiples services, la mise en place d'un point de terminaison distinct pour chaque service et la gestion par ce client de chaque point de terminaison peuvent s'avérer difficiles. Par exemple, une application de commerce électronique peut proposer des services tels que la recherche, les commentaires, le panier, la validation des achats et l'historique des commandes. Chaque service possède une API différente qui impose une interaction avec le client, et le client doit identifier chaque point de terminaison pour se connecter aux services. En cas de modification ou de mise à jour, le client doit également être mis à jour. Si vous réorganisez un service en deux services distincts ou plus, le code doit changer à la fois dans le service et le client.

Solution

Placez une passerelle devant un ensemble d'applications, de services ou de déploiements. Utilisez la couche d'application 7 pour acheminer la requête vers les instances appropriées.

Avec ce patron, l'application cliente ne doit pouvoir identifier qu'un seul point de terminaison et communiquer avec lui. Si un service est consolidé ou décomposé, le client ne doit pas nécessairement être mis à jour. Il peut continuer à soumettre des requêtes à la passerelle et seul le routage change.

Une passerelle permet également d'extraire les services principaux des clients, ce qui permet de conserver la simplicité des appels clients tout en autorisant des changements au niveau des services principaux derrière la passerelle. Les appels clients peuvent être acheminés vers les services requis qui doivent gérer le comportement du client prévu, ce qui permet d'ajouter, de fractionner et de réorganiser les services derrière la passerelle sans modifier le client.



Ce patron peut également vous assister dans le cadre du déploiement, en vous laissant gérer la manière dont les mises à jour sont déployées aux utilisateurs. Lorsqu'une nouvelle version du service est déployée, elle peut l'être en parallèle avec la version existante. Le routage permet de gérer la version du service qui est présentée aux clients, en vous donnant la possibilité d'utiliser différentes stratégies de gestion des versions, incrémentale, parallèle, ou des déploiements de mises à jour dans leur totalité. Si un problème est identifié après le déploiement du nouveau service, le déploiement peut être annulé rapidement en modifiant la configuration au niveau de la passerelle, sans affecter les clients.

Problèmes et considérations

- Le service de passerelle peut introduire un point de défaillance unique. Assurez-vous qu'il est répond correctement à vos critères de disponibilité. Envisagez d'utiliser des fonctions de résilience et de tolérance aux pannes lors de l'implémentation.

- Le service de passerelle peut introduire un goulot d'étranglement. Assurez-vous que les performances de la passerelle sont adéquates pour gérer la charge et qu'elle peut facilement être redimensionnée pour répondre à vos objectifs de croissance.

- Effectuez un test de charge sur la passerelle pour vous assurer de ne pas entraîner une cascade de défaillances des services.
- Le routage de la passerelle est de niveau 7. Il peut être basé sur l'adresse IP, le port, l'en-tête ou l'URL.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Un client a besoin de consommer plusieurs services accessibles derrière une passerelle.
- Vous souhaitez simplifier les applications clientes à l'aide d'un seul point de terminaison.
- Vous devez acheminer des requêtes depuis des points de terminaison adressables en externe vers des points de terminaison virtuels internes, comme l'exposition des ports sur une machine virtuelle sur des adresses IP virtuelles en cluster.

Ce patron n'est pas nécessairement adapté pour une situation dans laquelle une simple application n'utilise qu'un ou deux services.

Exemple

En utilisant Nginx comme routeur, l'exemple suivant est un simple fichier de configuration d'un serveur qui achemine les requêtes d'applications depuis des répertoires virtuels divers vers différentes machines sur le serveur principal.

```
server {
    listen 80;
    server_name domain.com;

    location /app1 {
        proxy_pass http://10.0.3.10:80;
    }

    location /app2 {
        proxy_pass http://10.0.3.20:80;
    }

    location /app3 {
        proxy_pass http://10.0.3.30:80;
    }
}
```

Conseils connexes

- Patron Backends for Frontends (Backends pour Frontends)
- Patron Gateway Aggregation (agrégation de passerelles)
- Patron Gateway Offloading (déchargement de passerelle)

Patron Health Endpoint Monitoring (Point de terminaison pour la surveillance de fonctionnement)

Implémentez des contrôles fonctionnels dans une application auxquels des outils externes ont accès via des points de terminaison exposés à intervalles réguliers. Cela permet de vérifier que les applications et les services fonctionnent correctement.

Contexte et problème

La surveillance des applications web et des services principaux est une bonne pratique et souvent une exigence opérationnelle pour s'assurer qu'ils sont disponibles et fonctionnent correctement. Toutefois, il est plus difficile de surveiller des services exécutés dans le cloud, qu'en local. Par exemple, vous ne contrôlez pas entièrement l'environnement d'hébergement, et en général les services dépendent d'autres services fournis par les fournisseurs de plates-formes.

Plusieurs facteurs ont un impact sur les applications hébergées dans le cloud telles que la latence du réseau, les performances et la disponibilité des systèmes de calcul et de stockage sous-jacents, ainsi que la bande passante réseau entre ces systèmes. Le service peut échouer totalement ou partiellement en raison d'un de ces facteurs. Par conséquent, vous devez vérifier à intervalles réguliers que le service fonctionne correctement pour garantir le niveau de disponibilité requis, ce qui peut d'ailleurs être inscrit dans votre contrat de niveau de service (SLA).

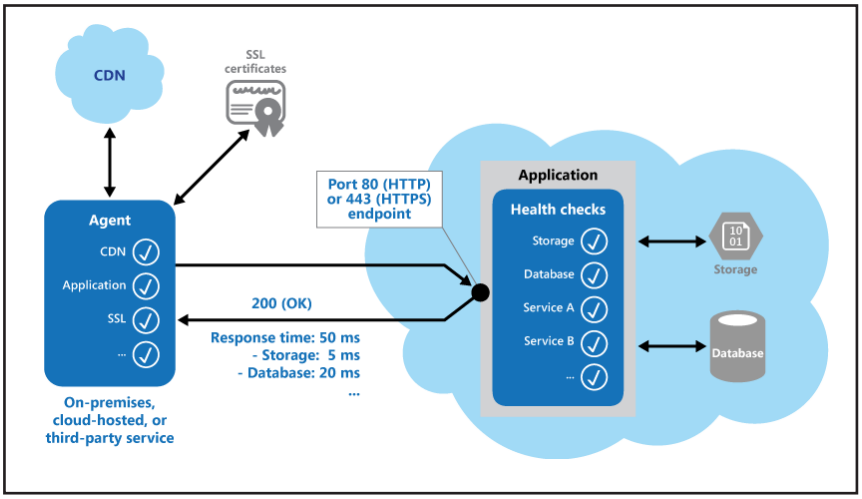
Solution

Mettez en œuvre la surveillance de l'intégrité en envoyant des requêtes à un point de terminaison sur l'application. L'application doit effectuer les vérifications nécessaires et renvoyer une indication de son statut.

Une vérification de la surveillance de l'intégrité combine en général deux facteurs :

- Les vérifications (le cas échéant) réalisées par l'application ou le service en réponse à la requête du point de terminaison de la vérification de l'intégrité.
- L'analyse des résultats par l'outil ou l'infrastructure qui effectue la vérification de l'intégrité.

Le code de réponse indique l'état de l'application et, éventuellement, des composants ou services qu'elle utilise. Le contrôle du temps de latence ou de réponse est effectué par l'outil ou l'infrastructure de surveillance. Cette figure fournit un aperçu du patron.



Parmi les autres contrôles qui peuvent être effectués par la code de la fonction du contrôle d'intégrité de l'application, citons les éléments suivants :

- Vérification du stockage dans le cloud ou du temps de disponibilité et de réponse d'une base de données.
- Vérification d'autres ressources ou services situés dans l'application ou en dehors, mais qui sont utilisés par l'application.

Des services et des outils sont disponibles pour surveiller les applications web en soumettant une requête à un ensemble configurable de points de terminaison, puis en comparant les résultats à un ensemble de règles configurables. Il est relativement facile de créer un point de terminaison de service dont le seul but est d'effectuer certains tests fonctionnels sur le système.

Parmi les vérifications classiques qui peuvent être effectuées par les outils de surveillance, citons les points suivants :

- Validation du code de réponse. Par exemple, une réponse HTTP de 200 (OK) indique que l'application a répondu sans erreur. Le système de surveillance peut également vérifier d'autres codes de réponse pour obtenir des résultats plus complets.
- Contrôle du contenu de la réponse pour détecter des erreurs, même si un code de statut de 200 (OK) est renvoyé. Cela permet de détecter des erreurs qui n'affectent qu'une section de la page web retournée ou de la réponse du service. Par exemple, la vérification du titre d'une page ou la recherche d'un groupe de mots spécifique indiquant que la page correcte a été retournée.
- Calcul du temps de réponse, qui correspond à une combinaison du temps de latence du réseau et de la durée d'exécution de la requête par l'application. Une valeur croissante peut indiquer un problème potentiel au niveau de l'application ou du réseau.
- Vérification des ressources ou des services situés à l'extérieur de l'application tels qu'un réseau de distribution de contenu utilisé par l'application pour fournir le contenu à partir de mémoires caches globales.
- Vérification de l'expiration des certificats SSL.
- Calcul du temps de réponse d'une recherche DNS pour l'URL de l'application pour mesurer la latence et les défaillances DNS.
- Validation de l'URL retournée par la recherche DNS pour garantir des entrées correctes. Cela permet d'éviter la redirection malveillante des requêtes via une attaque réussie sur le serveur DNS.

Il est également utile, dans la mesure du possible, de lancer ces vérifications sur différents sites locaux ou hébergés pour calculer et comparer les temps de réponse. L'idéal serait de surveiller les applications à partir d'emplacements situés à proximité des clients pour obtenir une vision précise des performances de chaque emplacement. En plus d'offrir un système de vérification plus robuste, les résultats peuvent vous aider à déterminer l'emplacement du déploiement pour l'application, et si le déploiement peut être exécuté dans plusieurs centres de données.

Des tests doivent également être exécutés pour toutes les instances de services que les clients utilisent afin de s'assurer que l'application fonctionne correctement pour tous les clients. Par exemple, si le stockage des clients est réparti sur plusieurs comptes de stockage, le processus de surveillance doit les vérifier tous.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Comment valider la réponse. Par exemple, est-ce que le seul code de statut 200 (OK) est suffisant pour s'assurer que l'application fonctionne correctement ? Même si cela constitue un instrument de mesure brut de la disponibilité des applications, et qu'il s'agit de l'implémentation minimale de ce patron, il offre peu d'informations sur les opérations, les tendances et les éventuels problèmes à venir au niveau de l'application.

Assurez-vous que l'application renvoie bien le code 200 (OK) uniquement lorsque la ressource cible est trouvée et traitée. Dans certaines situations, par exemple lorsque vous utilisez une page maître pour héberger la page web cible, le serveur renvoie un code de statut 200 (OK) au lieu du code d'erreur 404 (introuvable), même si la page cible du contenu est introuvable.

Nombre de points de terminaison à exposer pour une application. Une approche consiste à exposer au moins un point de terminaison pour les services de base que l'application utilise et un autre pour les services de priorité inférieure. Cela permet ainsi d'attribuer différents niveaux d'importance aux différents résultats d'analyse. Envisagez également d'exposer plusieurs points de terminaison, par exemple un pour chaque service de base, afin d'accroître la granularité de surveillance. Par exemple, un contrôle d'intégrité peut vérifier la base de données, le stockage et un service de géocodage externe utilisé par une application, qui nécessitent tous un niveau différent de durée de fonctionnement et de temps de réponse. L'application peut encore être intègre si le service de géocodage, ou une autre tâche en arrière-plan, est indisponible pendant quelques minutes.

Utiliser ou non le même point de terminaison pour la surveillance que pour l'accès général, mais avec un chemin spécifique conçu pour les contrôles d'intégrité, par exemple, /HealthCheck/{GUID}/ sur le point de terminaison d'accès général. Cela permet l'exécution de certains tests fonctionnels dans l'application par les outils d'analyse, tels que l'ajout de l'enregistrement d'un nouvel utilisateur, sa connexion et la prise d'une commande test, tout en vérifiant également que le point de terminaison d'accès général est disponible.

Type d'informations à collecter dans le service en réponse aux demandes de surveillance, et comment renvoyer ces informations. La plupart des structures et outils existants examinent uniquement le code de statut HTTP renvoyé par le point de terminaison. Pour renvoyer et valider des informations supplémentaires, vous pouvez être amené à créer un service ou un utilitaire de surveillance personnalisé.

Quantité d'informations à collecter. La réalisation d'un traitement excessif au cours de la vérification peut surcharger l'application et affecter les autres utilisateurs. Le temps nécessaire peut dépasser le délai d'attente du système de surveillance, qui marque alors l'application comme non disponible. La plupart des applications incluent des instruments tels que des gestionnaires d'erreurs et des compteurs de performances. Ils consignent les performances et des informations détaillées sur les erreurs. Ils peuvent s'avérer suffisants et éviter d'avoir à renvoyer des informations supplémentaires d'un contrôle d'intégrité.

Mise en cache de l'état du point de terminaison. Il peut être onéreux d'effectuer des contrôles d'intégrité trop fréquents. Si l'état d'intégrité est signalé via un tableau de bord, par exemple, vous ne voulez pas que chaque demande issue du tableau de bord déclenche un contrôle d'intégrité. Vérifiez plutôt régulièrement l'intégrité du système et mettez en cache l'état obtenu. Exposez un point de terminaison qui renvoie l'état mis en cache.

Comment configurer la sécurité pour les points de terminaison de surveillance, afin de les protéger d'un accès public qui pourrait exposer l'application à des attaques malveillantes, menacer de dévoiler des informations sensibles ou attirer des attaques par déni de service. En général, cela doit être fait dans la configuration de l'application pour pouvoir être mis à jour facilement sans avoir à redémarrer l'application. Envisagez d'utiliser une ou plusieurs des techniques suivantes :

- Sécurisez le point de terminaison en exigeant une authentification. Pour ce faire, vous pouvez utiliser une clé de sécurité d'authentification dans l'en-tête de demande ou transmettre des informations d'identification avec la demande, sous réserve que le service ou l'outil de surveillance prenne en charge l'authentification.
    - Utilisez un point de terminaison obscur ou masqué. Par exemple, exposez le point de terminaison sur une adresse IP différente de celle utilisée par l'URL de l'application par défaut, configurez le point de terminaison sur un port HTTP standard et/ou utilisez un chemin complexe vers la page de test. Vous pouvez généralement spécifier les ports et les adresses de points de terminaison supplémentaires dans la configuration de l'application, et ajouter des entrées pour ces points de terminaison sur le serveur DNS, si nécessaire, pour éviter d'avoir à spécifier directement l'adresse IP.

  - Exposez une méthode sur un point de terminaison qui accepte un paramètre tel qu'une valeur de clé ou une valeur de mode de fonctionnement. Selon la valeur fournie pour ce paramètre, quand une demande est reçue, le code peut effectuer un test spécifique ou un ensemble de tests, ou renvoyer une erreur 404 (Introuvable) si la valeur du paramètre n'est pas reconnue. Les valeurs de paramètre reconnues peuvent être définies dans la configuration de l'application.
  - Les attaques par déni de service sont susceptibles d'avoir un impact moindre sur un point de terminaison distinct qui exécute des tests fonctionnels de base, sans compromettre le fonctionnement de l'application. Dans l'idéal, évitez d'utiliser un test susceptible d'exposer des informations sensibles. Si vous devez renvoyer des informations qui pourraient être utiles à un attaquant, réfléchissez à la manière de protéger le point de terminaison et les données contre tout accès non autorisé. Dans ce cas, il ne suffit pas de compter simplement sur l'obscurité. Vous devez également envisager d'utiliser une connexion HTTPS et de chiffrer toutes les données sensibles, même si cela augmente la charge appliquée au serveur.
- Comment accéder à un point de terminaison qui est sécurisé via une authentification. Tous les outils et toutes les structures ne peuvent pas être configurés de manière à inclure des informations d'identification avec la demande de contrôle d'intégrité. Par exemple, les fonctionnalités de contrôle d'intégrité intégrées à Microsoft Azure ne peuvent pas fournir d'informations d'authentification. Pingdom, Panopta, NewRelic et Statuscake représentent des alternatives tierces.
- Comment veiller à ce que l'agent de surveillance fonctionne correctement. Une approche consiste à exposer un point de terminaison qui renvoie simplement une valeur à partir de la configuration de l'application ou une valeur aléatoire qui peut être utilisée pour tester l'agent.

Veillez également à ce que le système de surveillance effectue des vérifications sur lui-même, telles qu'un test automatique et un test intégré, pour éviter qu'il fournisse des faux positifs.

Quand utiliser ce patron

Ce patron est utile dans les cas suivants :

- Surveillance de sites et d'applications web pour vérifier la disponibilité.
- Surveillance de sites et d'applications web pour vérifier leur bon fonctionnement.
- Surveillance de services intermédiaires ou partagés pour détecter et isoler un échec qui pourrait perturber d'autres applications.
- Apport d'un complément aux instruments existants dans l'application, tels que les compteurs de performances et les gestionnaires d'erreurs. Les contrôles d'intégrité ne remplacent pas la journalisation et l'audit exigés dans l'application. Les instruments peuvent fournir des informations précieuses pour une structure existante qui permettent de surveiller les compteurs et les journaux d'erreurs afin de détecter les échecs et d'autres problèmes. Toutefois, ils ne peuvent pas fournir d'informations si l'application n'est pas disponible.

Exemple

Les exemples de code suivants, tirés de la classe HealthCheckController (un exemple illustrant ce patron est disponible sur GitHub), illustrent l'exposition d'un point de terminaison pour effectuer une série de contrôles d'intégrité.

La méthode CoreServices, illustrée ci-dessous en langage C#, effectue une série de contrôles sur les services utilisés dans l'application. Si tous les tests s'exécutent sans erreur, la méthode retourne un code de statut 200 (OK). Si l'un des tests lève une exception, la méthode renvoie un code de statut 500 (Erreur interne). La méthode peut éventuellement renvoyer des informations supplémentaires en cas d'erreur, si la structure ou l'outil de surveillance est en mesure de les utiliser.

```
public ActionResult CoreServices()
{
  try
  {
    // Run a simple check to ensure the database is available.
    DataStore.Instance.CoreHealthCheck();

    // Run a simple check on our external service.
    MyExternalService.Instance.CoreHealthCheck();
  }
  catch (Exception ex)
  {
    Trace.TraceError("Exception in basic health check: {0}", ex.Message);

    // This can optionally return different status codes based on the exception.
    // Optionally it could return more details about the exception.
    // The additional information could be used by administrators who access the
    // endpoint with a browser, or using a ping utility that can display the
    // additional information.
    return new HttpStatusCodeResult((int)HttpStatusCode.InternalServerError);
  }
  return new HttpStatusCodeResult((int)HttpStatusCode.OK);
}
```

La méthode ObscurePath montre comment vous pouvez lire un chemin à partir de la configuration de l'application et l'utiliser comme point de terminaison pour les tests. Cet exemple, en C#, montre également comment vous pouvez accepter un ID comme paramètre et l'utiliser pour vérifier les demandes valides.

```
public ActionResult ObscurePath(string id)
{
  // The id could be used as a simple way to obscure or hide the endpoint.
  // The id to match could be retrieved from configuration and, if matched,
  // perform a specific set of tests and return the result. If not matched it
  // could return a 404 (Not Found) status.

  // The obscure path can be set through configuration to hide the endpoint.
  var hiddenPathKey = CloudConfigurationManager.GetSetting("Test.ObscurePath");

  // If the value passed does not match that in configuration, return 404 (Not Found).
  if (!string.Equals(id, hiddenPathKey))
  {
    return new HttpStatusCodeResult((int)HttpStatusCode.NotFound);
  }

  // Else continue and run the tests...
  // Return results from the core services test.
  return this.CoreServices();
}
```

La méthode TestResponseFromConfig montre comment vous pouvez exposer un point de terminaison qui effectue une vérification pour une valeur de paramètre de configuration spécifiée.

```
public ActionResult TestResponseFromConfig()
{
  // Health check that returns a response code set in configuration for testing.
  var returnStatusCodeSetting = CloudConfigurationManager.GetSetting(
                                                        "Test.ReturnStatusCode");

  int returnStatusCode;

  if (!int.TryParse(returnStatusCodeSetting, out returnStatusCode))
  {
    returnStatusCode = (int)HttpStatusCode.OK;
  }

  return new HttpStatusCodeResult(returnStatusCode);
}
```

Surveillance des points de terminaison dans les applications hébergées sur Azure

Certaines options de surveillance des points de terminaison dans les applications Azure sont :

- Utiliser les fonctionnalités intégrées de surveillance d'Azure.
- Utiliser une structure ou un service tiers tel que Microsoft System Center Operations Manager.
- Créer un utilitaire personnalisé ou un service qui s'exécute sur votre propre serveur ou sur un serveur hébergé.

Même si Azure fournit un ensemble assez complet d'options de surveillance, vous pouvez utiliser des outils et services supplémentaires pour fournir des informations supplémentaires. Les services de gestion Azure fournissent un mécanisme de surveillance intégré pour les règles d'alerte. La section des alertes de la page des services de gestion dans le portail Azure vous permet de configurer jusqu'à dix règles d'alerte par abonnement pour vos services. Ces règles spécifient une condition et une valeur seuil pour un service, telle que la charge CPU ou le nombre de demandes ou d'erreurs par seconde, et le service peut envoyer automatiquement des notifications par e-mail aux adresses que vous définissez dans chaque règle.

Les conditions que vous pouvez surveiller varient selon le mécanisme d'hébergement que vous choisissez pour votre application (par exemple, des sites web, des services cloud, des machines virtuelles ou des services mobiles), mais tous ces éléments incluent la capacité à créer une règle d'alerte qui utilise un point de terminaison web que vous spécifiez dans les paramètres de votre service. Ce point de terminaison doit répondre dans un délai raisonnable pour que le système d'alerte puisse détecter que l'application fonctionne correctement.

Lisez plus d'informations sur la création de notifications d'alerte.

Si vous hébergez votre application dans des machines virtuelles ou des rôles de travail ou des rôles web de services cloud Azure, vous pouvez tirer parti du service intégré d'Azure appelé Traffic Manager. Traffic Manager est un service de routage et d'équilibrage de la charge qui peut distribuer les demandes à des instances spécifiques de votre application hébergée dans les services cloud basés sur une plage de règles et de paramètres.

Outre les demandes de routage, Traffic Manager envoie une requête ping pour obtenir une URL, un port et un chemin relatif que vous spécifiez sur une base régulière, afin de déterminer quelles instances de l'application définie dans ses règles sont actives et répondent aux demandes. S'il détecte un code de statut 200 (OK), il marque l'application comme disponible. Pour tout autre code de statut, Traffic Manager marque l'application comme étant hors connexion. Vous pouvez afficher le statut dans la console de Traffic Manager et configurer la règle pour rediriger les demandes vers d'autres instances de l'application qui répondent.

Toutefois, Traffic Manager n'attend que dix secondes pour recevoir une réponse de l'URL de surveillance. Par conséquent, vous devez vous assurer que votre code de contrôle d'intégrité s'exécute dans ce délai, en autorisant la latence réseau pour l'aller-retour à partir de Traffic Manager vers votre application et inversement.

Lisez plus d'informations sur l'utilisation de Traffic Manager pour surveiller vos applications. Traffic Manager est également abordé dans Conseils de déploiement de plusieurs centres de données.

Conseils connexes

Les documents suivants peuvent être utiles lors de l'implémentation de ce patron :

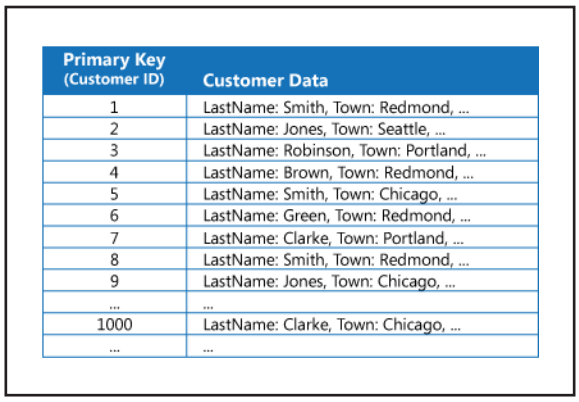
- Conseils pour l'instrumentation et la télémétrie. Le contrôle de l'intégrité des services et des composants se fait généralement au moyen de la détection, mais il est également utile de disposer d'informations pour surveiller les performances des applications et détecter les événements qui se produisent au moment de l'exécution. Ces données peuvent être transmises en retour à des outils de surveillance, en tant qu'informations d'analyse du fonctionnement. La rubrique Conseils d'instrumentation et de télémétrie explore la collecte des informations de diagnostic à distance obtenues à l'aide des instruments figurant dans les applications.
- Réception de notifications d'alerte.
- Ce patron inclut un exemple d'application téléchargeable.

Patron Index Table (Tableau indexé)

Créez des index sur les champs dans des magasins de données qui sont souvent référencés par requêtes. Ce patron peut améliorer les performances des requêtes en permettant aux applications de localiser plus rapidement les données à récupérer auprès d'un magasin de données.

Contexte et problème

De nombreux magasins de données organisent les données pour collecter les entités à l'aide de la clé primaire. Une application peut utiliser cette clé pour localiser et récupérer des données. La figure ci-dessous montre un exemple de magasin de données détenant des informations client. La clé primaire est l'ID client. La figure ci-dessous montre les informations client, classées par clé primaire (ID client).



Primary Key (Customer ID)	Customer Data
1	LastName: Smith, Town: Redmond, ...
2	LastName: Jones, Town: Seattle, ...
3	LastName: Robinson, Town: Portland, ...
4	LastName: Brown, Town: Redmond, ...
5	LastName: Smith, Town: Chicago, ...
6	LastName: Green, Town: Redmond, ...
7	LastName: Clarke, Town: Portland, ...
8	LastName: Smith, Town: Redmond, ...
9	LastName: Jones, Town: Chicago, ...
...	...
1000	LastName: Clarke, Town: Chicago, ...
...	...

La clé primaire a de la valeur pour les requêtes qui extraient des données selon la valeur de cette clé, mais une application peut ne pas être en mesure d'utiliser la clé primaire si elle a besoin de récupérer des données basées sur un autre champ. Dans l'exemple des clients, une application ne peut pas utiliser la clé primaire de l'ID client pour récupérer les clients si elle interroge les données uniquement en référençant la valeur d'un autre attribut, par exemple de la ville dans laquelle se trouve le client. Pour effectuer une requête comme celle-ci, l'application peut avoir à extraire et à examiner chaque enregistrement client, ce qui peut ralentir le processus.

De nombreux systèmes de gestion de base de données relationnelle prennent en charge les index secondaires. Un index secondaire est une structure de données distincte qui est organisée par un ou plusieurs champs clés (secondaires) non primaires, et elle indique où les données sont stockées pour chaque valeur indexée. Les éléments d'un index secondaire sont généralement classés par la valeur des clés secondaires pour permettre une recherche rapide des données. La maintenance de ces index est généralement effectuée automatiquement par le système de gestion de base de données.

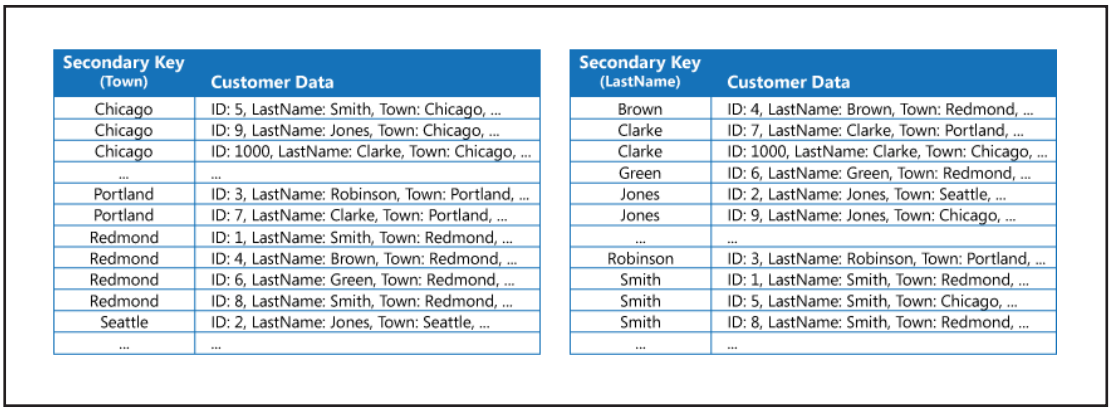
Vous pouvez créer autant d'index secondaires que nécessaire pour prendre en charge les différentes requêtes que votre application effectue. Par exemple, dans une table des clients d'une base de données relationnelle où l'ID client est la clé primaire, il est avantageux d'ajouter un index secondaire sur le champ Ville, si l'application recherche souvent des clients en fonction de la ville où ils résident.

Toutefois, même si les index secondaires sont courants dans les systèmes relationnels, la plupart des magasins de données NoSQL utilisés par les applications cloud ne fournissent pas de fonctionnalité équivalente.

Solution

Si le magasin de données ne prend pas en charge les index secondaires, vous pouvez les émuler manuellement en créant vos propres tables d'index. Une table d'index organise les données en fonction d'une clé spécifiée. Trois stratégies sont couramment utilisées pour structurer une table d'index, selon le nombre d'index secondaires qui sont nécessaires et la nature des requêtes qu'effectue une application.

La première stratégie consiste à dupliquer les données dans chaque table d'index mais à les organiser en fonction de clés différentes (dénormalisation complète). La figure suivante illustre les tables d'index qui organisent les mêmes données client par ville (Town) et nom de famille (LastName).

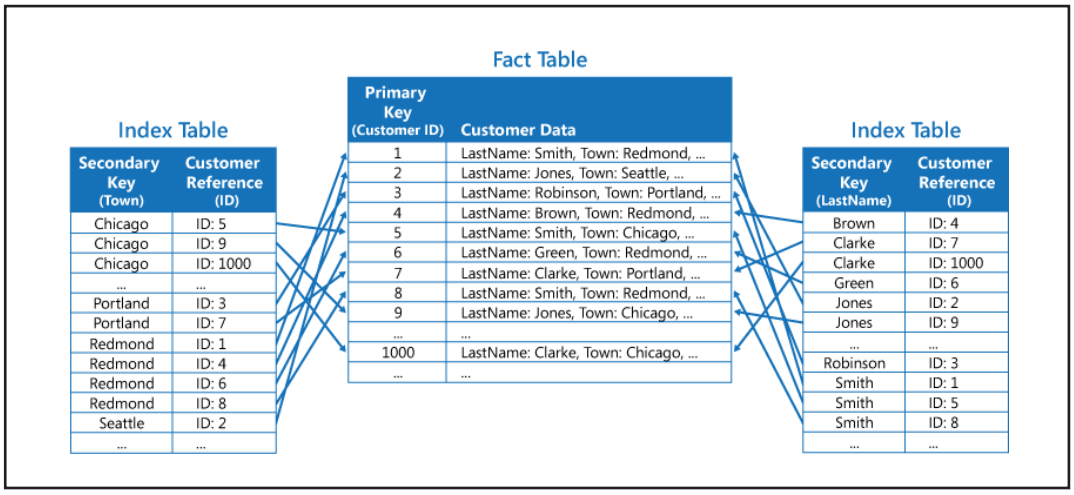


Secondary Key (Town)	Customer Data
Chicago	ID: 5, LastName: Smith, Town: Chicago, ...
Chicago	ID: 9, LastName: Jones, Town: Chicago, ...
Chicago	ID: 1000, LastName: Clarke, Town: Chicago, ...
...	...
Portland	ID: 3, LastName: Robinson, Town: Portland, ...
Portland	ID: 7, LastName: Clarke, Town: Portland, ...
Redmond	ID: 1, LastName: Smith, Town: Redmond, ...
Redmond	ID: 4, LastName: Brown, Town: Redmond, ...
Redmond	ID: 6, LastName: Green, Town: Redmond, ...
Redmond	ID: 8, LastName: Smith, Town: Redmond, ...
Seattle	ID: 2, LastName: Jones, Town: Seattle, ...
...	...

Secondary Key (LastName)	Customer Data
Brown	ID: 4, LastName: Brown, Town: Redmond, ...
Clarke	ID: 7, LastName: Clarke, Town: Portland, ...
Clarke	ID: 1000, LastName: Clarke, Town: Chicago, ...
Green	ID: 6, LastName: Green, Town: Redmond, ...
Jones	ID: 2, LastName: Jones, Town: Seattle, ...
Jones	ID: 9, LastName: Jones, Town: Chicago, ...
...	...
Robinson	ID: 3, LastName: Robinson, Town: Portland, ...
Smith	ID: 1, LastName: Smith, Town: Redmond, ...
Smith	ID: 5, LastName: Smith, Town: Chicago, ...
Smith	ID: 8, LastName: Smith, Town: Redmond, ...
...	...

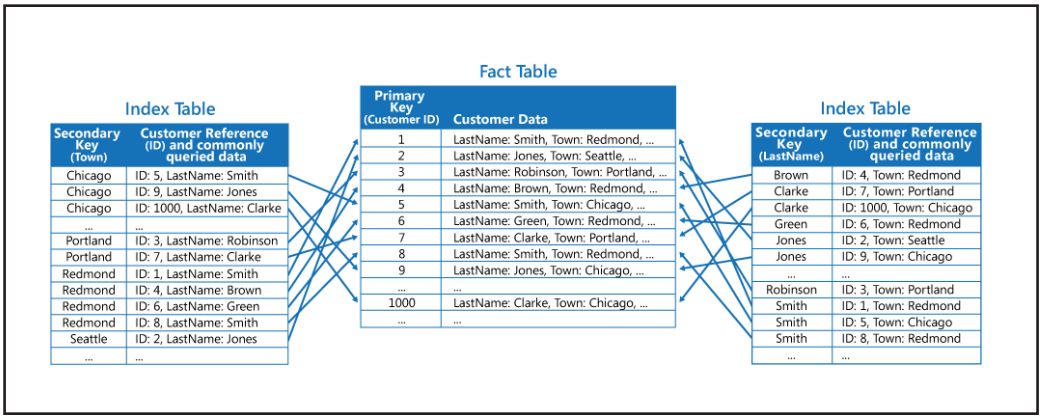
Cette stratégie est appropriée si les données sont relativement statiques par rapport au nombre de fois où elles sont interrogées à l'aide de chaque clé. Si les données sont plus dynamiques, les charges de traitement de la maintenance de chaque table d'index deviennent trop grandes pour que cette approche soit utile. En outre, si le volume de données est très important, l'espace requis pour stocker les données dupliquées est important.

La deuxième stratégie consiste à créer des tables d'index normalisées, organisées selon différentes clés, et à référencer les données d'origine en utilisant la clé primaire plutôt qu'en les dupliquant, comme illustré dans la figure suivante. Les données d'origine sont appelées « table de faits ».



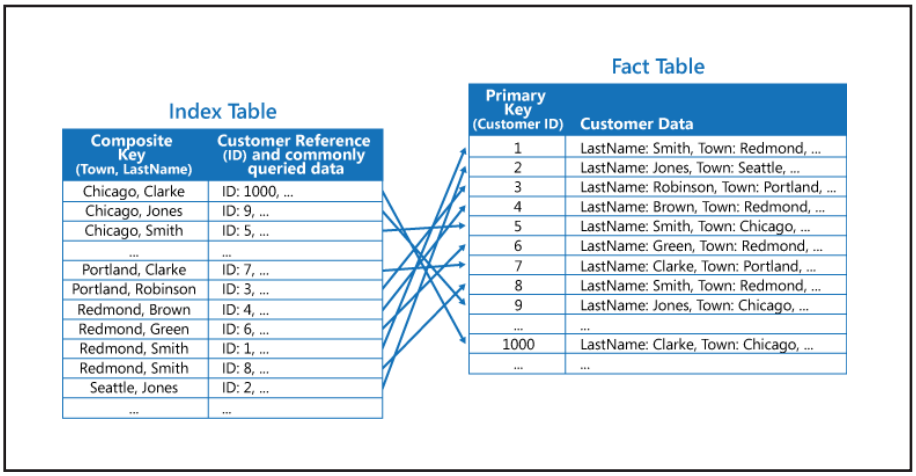
Cette technique économise de l'espace et réduit les charges de la maintenance des données dupliquées. L'inconvénient est qu'une application doit effectuer deux opérations de recherche pour trouver des données à l'aide d'une clé secondaire. Elle doit trouver la clé primaire des données dans la table d'index, puis utiliser cette clé primaire pour rechercher les données dans la table de faits.

La troisième stratégie consiste à créer des tables d'index partiellement normalisées, organisées selon différentes clés qui dupliquent les champs fréquemment consultés. Référencez la table de faits pour accéder aux champs moins fréquemment consultés. La figure suivante montre comment des données couramment consultées sont dupliquées dans chaque table d'index.

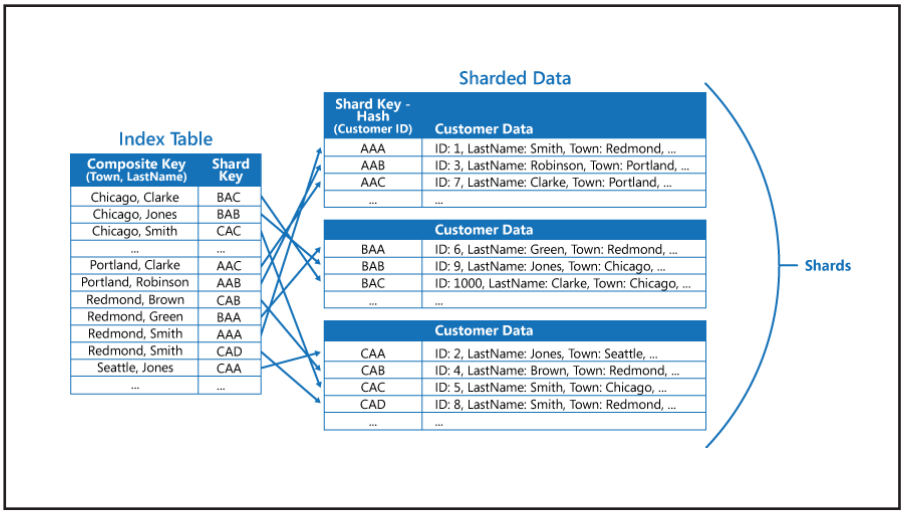


Cette stratégie offre un équilibre entre les deux premières approches. Il est possible de récupérer rapidement les données des requêtes courantes en utilisant une seule recherche, et les charges de maintenance et d'espace ne sont pas aussi importantes que lors de la duplication du jeu de données complet.

Si une application recherche fréquemment des données en combinant des valeurs (par exemple, « Rechercher tous les clients qui vivent à Redmond et dont le nom de famille est Smith »), vous pouvez implémenter les clés des éléments dans la table d'index sous la forme d'une concaténation de l'attribut Town et de l'attribut LastName. La figure suivante montre une table d'index basée sur des clés composites. Les clés sont triées par ville (Town), puis par nom de famille (LastName) pour les enregistrements qui ont une même valeur de ville.



Les tables d'index peuvent accélérer les opérations de requête sur des données fragmentées et elles sont particulièrement utiles lorsque la clé de fragment est hachée. La figure suivante montre un exemple où la clé de fragment est un hachage de l'ID client. La table d'index peut organiser les données en fonction de la valeur non hachée (Town et LastName) et fournir la clé de fragment hachée comme données de recherche. Cela peut éviter à l'application d'avoir à calculer de façon répétée des clés de hachage (opération coûteuse) si elle a besoin de récupérer des données qui se situent dans une plage ou si elle a besoin d'extraire des données dans l'ordre de la clé non hachée. Par exemple, une requête telle que « Rechercher tous les clients qui vivent à Redmond » peut être rapidement résolue en localisant les éléments correspondants dans la table d'index, où ils sont tous stockés dans un bloc contigu. Ensuite, suivez les références vers les données client en utilisant les clés de fragment stockées dans la table d'index.



Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- Les charges de la maintenance des index secondaires peuvent être importantes. Vous devez analyser et comprendre les requêtes que votre application utilise. Créez uniquement des tables d'index quand elles sont susceptibles d'être utilisées régulièrement. Ne créez pas de tables d'index spéculatives pour prendre en charge des requêtes qu'une application n'effectue pas, ou n'effectue qu'occasionnellement.

La duplication des données dans une table d'index peut générer une surcharge importante en coûts de stockage et en efforts nécessaires pour conserver plusieurs copies des données.

- L'implémentation d'une table d'index comme structure normalisée référençant les données d'origine nécessite une application pour effectuer deux opérations de recherche afin de trouver les données. La première opération recherche dans la table d'index pour récupérer la clé primaire, et la seconde utilise cette clé primaire pour extraire les données.
- Si un système intègre plusieurs tables d'index sur de très grands jeux de données, il peut être difficile de maintenir la cohérence entre les tables d'index et les données d'origine. Il peut être possible de concevoir l'application autour du modèle de cohérence à terme. Par exemple, pour insérer, mettre à jour ou supprimer des données, une application pourrait publier un message à une file d'attente et laisser une tâche distincte effectuer l'opération et maintenir les tables d'index qui référencent ces données de façon asynchrone. Pour plus d'informations sur l'implémentation de la cohérence à terme, consultez la rubrique Introduction à la cohérence des données.
- Les tables de stockage Microsoft Azure prennent en charge les mises à jour transactionnelles pour les modifications apportées aux données détenues dans la même partition (appelées transactions de groupe d'entités). Si vous pouvez stocker les données d'une table de faits, ainsi qu'une ou plusieurs tables d'index dans la même partition, vous pouvez utiliser cette fonctionnalité pour assurer la cohérence.
- Les tables d'index peuvent elles-mêmes être partitionnées ou fragmentées.

Quand utiliser ce patron

Utilisez ce patron pour améliorer les performances des requêtes lorsqu'une application a souvent besoin de récupérer des données en utilisant une clé différente de la clé primaire (ou de fragment).

Ce patron peut ne pas être approprié quand :

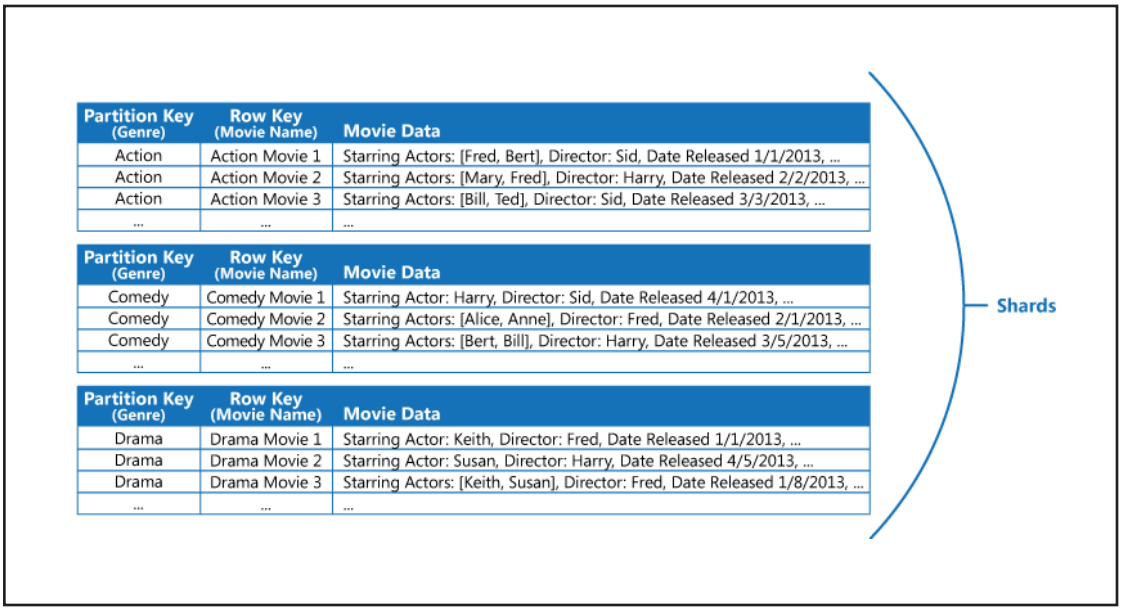
- Les données sont volatiles. Une table d'index peut très rapidement devenir obsolète, ce qui la rend inefficace ou fait que les charges de maintenance de la table d'index dépassent toutes les économies liées à son utilisation.
- Un champ sélectionné comme clé secondaire pour une table d'index est non discriminatoire et ne peut avoir qu'un petit ensemble de valeurs (par exemple, le sexe).
- Le bilan des valeurs de données pour un champ sélectionné comme clé secondaire pour une table d'index est fortement déséquilibré. Par exemple, si 90 % des enregistrements contiennent la même valeur dans un champ, la création et la maintenance d'une table d'index pour rechercher des données en fonction de ce champ peuvent générer plus de charges que l'analyse séquentielle des données. Toutefois, si les requêtes ciblent très fréquemment les valeurs figurant dans les 10 % restants, cet index peut être utile. Vous devez comprendre les requêtes que votre application effectue et la fréquence à laquelle elles surviennent.

Exemple

Les tables de stockage Azure fournissent un magasin de données clé/valeur hautement évolutif pour les applications qui s'exécutent dans le cloud. Les applications stockent et récupèrent des valeurs de données en spécifiant une clé. Les valeurs de données peuvent contenir plusieurs champs, mais la structure d'un élément de données est opaque pour le stockage de table, qui gère simplement un élément de données comme un tableau d'octets.

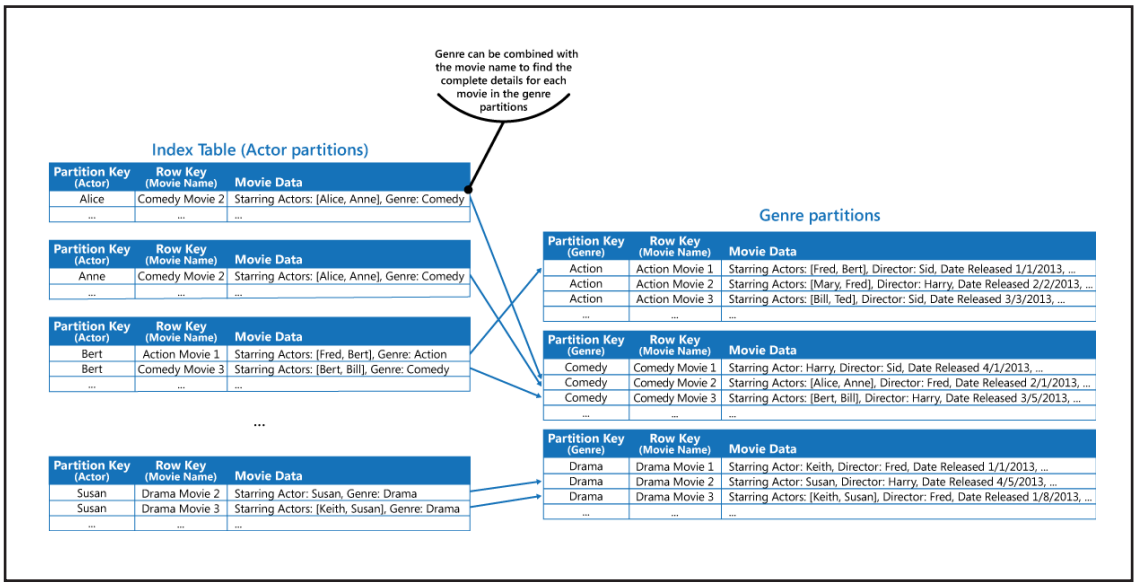
Les tables de stockage Azure prennent également en charge la fragmentation. La clé de fragmentation inclut deux éléments, une clé de partition et une clé de ligne. Les éléments qui ont la même clé de partition sont stockés dans la même partition (fragment), et les éléments sont stockés dans l'ordre des clés de ligne au sein d'un fragment. Le stockage de table est optimisé pour effectuer des requêtes qui extraient les données incluses dans une plage contigüe de valeurs de clé de ligne au sein d'une partition. Si vous créez des applications cloud qui stockent des informations dans les tables Azure, vous devez structurer vos données en tenant compte de cette fonctionnalité.

Par exemple, considérez une application qui stocke des informations sur des films. L'application recherche fréquemment des films par genre (action, documentaire, historique, comédie, drame, etc.). Vous pouvez créer une table Azure avec des partitions pour chaque genre en utilisant le genre comme clé de partition et en définissant le titre comme clé de ligne, comme cela est illustré dans la figure suivante.



Cette approche est moins efficace si l'application a également besoin de rechercher des films via un acteur vedette. Dans ce cas, vous pouvez créer une table Azure séparée agissant comme une table d'index. La clé de partition est l'acteur et la clé de ligne est le titre du film. Les données relatives aux différents acteurs sont stockées dans des partitions distinctes. Si un film met en scène plusieurs acteurs, le même film apparaît dans plusieurs partitions.

Vous pouvez dupliquer les données de film dans les valeurs détenues par chaque partition en adoptant la première approche décrite dans la section Solution ci-dessus. Toutefois, il est probable que chaque film sera répliqué plusieurs fois (une fois pour chaque acteur). Il peut donc être plus efficace de dénormaliser partiellement les données pour favoriser les requêtes les plus courantes (telles que les noms des autres acteurs) et permettre à une application de récupérer tous les détails restants en insérant la clé de partition nécessaire pour rechercher les informations complètes dans les partitions de genre. Cette approche est décrite par la troisième option abordée dans la section Solution. La figure suivante illustre cette approche.



Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- **Manuel de cohérence de données.** Une table d'index doit être maintenue comme les données dont elle indexe les modifications. Dans le cloud, il peut s'avérer impossible ou inapproprié d'effectuer des opérations qui mettent à jour un index dans le cadre de la même transaction qui modifie les données. Dans ce cas, une approche cohérente à terme est plus appropriée. Elle fournit des informations sur les problèmes entourant la cohérence à terme.

- **Patron Sharding (Partitionnement).** Le patron Index Table (Tableau indexé) est fréquemment utilisé conjointement aux données partitionnées à l'aide de fragments. Le patron Sharding (Partitionnement) fournit plus d'informations sur la façon de diviser un magasin de données en un ensemble de fragments.

- **Patron Materialized View (Vue matérialisée).** Au lieu d'indexer les données pour favoriser les requêtes qui synthétisent les données, il peut être plus approprié de créer une vue matérialisée des données. Cela décrit comment favoriser des requêtes de résumé efficaces en générant des vues préremplies des données.

Patron Leader Election (Élection du leader)

Coordonnez les actions effectuées par un ensemble d'instances collaboratrices dans une application distribuée en choisissant une instance comme le leader qui endossera la responsabilité de la gestion des autres. Ceci contribue à garantir que les instances n'entreront pas en conflit les unes avec les autres, ne provoqueront pas de contentions pour des ressources partagées et n'interféreront pas par inadvertance avec le travail effectué par d'autres instances.

Contexte et problème

Une application cloud standard a de nombreuses tâches qui agissent de manière coordonnée. Ces tâches pourraient toutes être des instances exécutant le même code et exigeant l'accès aux mêmes ressources, ou elles pourraient travailler ensemble en parallèle pour effectuer les parties individuelles d'un calcul complexe.

Les instances de tâche peuvent s'exécuter séparément pendant la majeure partie du temps, mais il peut aussi être nécessaire de coordonner les actions de chaque instance pour s'assurer qu'elles n'entrent pas en conflit, ne provoquent pas de contentions pour des ressources partagées et n'interfèrent pas accidentellement avec le travail que d'autres instances de tâche effectuent.

Par exemple :

- Dans un système basé sur le cloud qui implémente la mise à l'échelle horizontale, plusieurs instances de la même tâche peuvent s'exécuter en même temps, chaque instance desservant un utilisateur différent. Si ces instances écrivent dans une ressource partagée, il est nécessaire de coordonner leurs actions pour empêcher chaque instance d'écraser les modifications apportées par les autres.
- Si les tâches effectuent des éléments individuels d'un calcul complexe en parallèle, les résultats doivent être agrégés quand toutes les tâches se terminent.

Les instances de tâche sont toutes des homologues. Il n'y a donc pas de leader naturel pouvant agir comme coordonnateur ou agrégateur.

Solution

Une instance de tâche unique doit être élue pour agir en tant que leader et cette instance doit coordonner les actions des autres instances de tâche subordonnées. Si toutes les instances de tâche exécutent le même code, elles sont toutes capables d'agir en tant que leader. Par conséquent, le processus d'élection doit être géré avec soin pour empêcher deux instances ou plus d'adopter le rôle de leader en même temps.

Le système doit fournir un mécanisme fiable de sélection du leader. Cette méthode doit faire face à des événements tels que l'indisponibilité du réseau et des échecs de processus. Dans de nombreuses solutions, les instances de tâche subordonnées surveillent le leader au moyen d'une sorte de méthode de vérification de pulsations, ou par interrogation. Si le leader désigné s'interrompt inopinément, ou si une défaillance réseau rend le leader indisponible pour les instances de tâche subordonnées, celles-ci doivent élire un nouveau leader.

Plusieurs stratégies permettent d'élire un leader parmi un ensemble de tâches dans un environnement distribué, y compris les suivantes :

- Sélection de l'instance de tâche avec l'ID de processus ou l'instance de rang le plus bas.
- Course pour acquérir un mutex distribué, partagé. La première instance de tâche qui acquiert le mutex est le leader. Toutefois, le système doit garantir que, si le leader s'interrompt ou est déconnecté du reste du système, le mutex sera libéré pour permettre à une autre instance de tâche de devenir le leader.
- Implémentation d'un des algorithmes courants d'élection de leader, tel que l'algorithme de la brute (Bully algorithm) ou l'algorithme d'élection sur un anneau. Ces algorithmes supposent que chaque candidat à l'élection possède un ID unique et qu'il peut communiquer avec les autres candidats de façon fiable.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- Le processus d'élection d'un leader doit être résistant aux défaillances transitoires et permanentes.

- Il doit être possible de détecter le moment où le leader connaît une défaillance ou cesse d'être disponible (par exemple, en raison d'un échec de la communication). La rapidité nécessaire de la détection dépend du système. Certains systèmes peuvent fonctionner pendant une courte période sans leader, au cours de laquelle une anomalie transitoire peut être corrigée. Dans d'autres cas, il est nécessaire de détecter immédiatement la défaillance du leader et de déclencher une nouvelle élection.
- Dans un système qui implémente la mise à l'échelle automatique horizontale, le leader peut être interrompu si le système est dimensionné à la baisse et arrête certaines ressources de calcul.
- L'utilisation d'un mutex distribué et partagé introduit une dépendance sur le service externe qui fournit le mutex. Le service constitue un point de défaillance unique. S'il devient indisponible pour une raison quelconque, le système ne peut pas élire de leader.
- L'utilisation d'un processus dédié unique comme leader est une approche directe. Toutefois, si ce processus échoue, un retard important pourrait survenir avant son redémarrage. La latence qui en résulte peut affecter les performances et les temps de réponse des autres processus s'ils attendent que le leader coordonne une opération.
- L'implémentation d'un des algorithmes d'élection de leader fournit manuellement une souplesse optimale pour l'ajustage et l'optimisation du code.

Quand utiliser ce patron

Utilisez ce patron lorsque les tâches d'une application distribuée, telle qu'une solution hébergée dans le cloud, ont besoin d'une coordination soignée et qu'il n'y a pas de leader naturel.

Évitez de faire du leader un goulot d'étranglement dans le système. L'objectif du leader est de coordonner le travail des tâches subordonnées et il ne doit pas nécessairement participer à ce travail lui-même, bien qu'il doive être en mesure de le faire si la tâche n'est pas élue comme leader.

Ce patron peut ne pas être approprié si :

- Il existe un leader naturel ou un processus dédié qui peut toujours agir comme leader. Par exemple, il est parfois possible d'implémenter un processus de singleton qui coordonne les instances de tâche. Si ce processus échoue ou perd son intégrité, le système peut l'arrêter et le redémarrer.
- La coordination entre les tâches peut être réalisée à l'aide d'une méthode plus légère. Par exemple, si plusieurs instances de tâche ont besoin simplement d'un accès coordonné à une ressource partagée, une meilleure solution consiste à utiliser un verrouillage optimiste ou pessimiste pour contrôler l'accès.
- Une solution tierce est plus appropriée. Par exemple, le service Microsoft Azure HDInsight (basé sur Apache Hadoop) utilise les services fournis par Apache Zookeeper pour coordonner la carte et réduire les tâches qui recueillent et synthétisent les données.

Exemple

Le projet DistributedMutex dans la solution LeaderElection (un exemple illustrant ce patron est disponible sur GitHub) montre comment utiliser un bail sur un objet blob Azure Storage pour fournir un mécanisme d'implémentation d'un mutex distribué et partagé. Ce mutex peut être utilisé pour élire un leader parmi un groupe d'instances de rôle dans un service cloud Azure. La première instance de rôle à acquérir le bail est élue en tant que leader et elle reste leader jusqu'à ce qu'elle libère le bail ou ne soit pas en mesure de renouveler le bail. D'autres instances de rôle continuent de surveiller le bail de l'objet blob au cas où le leader ne serait plus disponible.

Un bail d'objet blob est un verrou d'écriture exclusif sur un objet blob. Un objet blob unique peut être l'objet d'un seul bail à n'importe quel moment. Une instance de rôle peut demander un bail sur un objet blob spécifié, et le bail lui est accordé si aucune autre instance de rôle ne détient de bail sur le même objet blob. Dans le cas contraire, la demande lève une exception.

Pour éviter une instance de rôle défaillante qui conserverait le bail indéfiniment, spécifiez une durée de vie. Lorsqu'il arrivera à expiration, le bail sera disponible. Toutefois, même si une instance de rôle détient le bail, elle peut demander son renouvellement et une nouvelle période lui sera accordée. L'instance de rôle peut répéter continuellement ce processus pour conserver le bail. Pour plus d'informations sur la location d'un blob, reportez-vous à la section Location d'un blob (REST API).

La classe BlobDistributedMutex dans l'exemple C# suivant contient la méthode RunTaskWhenMutexAquired qui permet à une instance de rôle d'acquérir un bail d'un blob donné. Les détails de l'objet blob (le nom, le conteneur et le compte de stockage) sont transmis au constructeur dans un objet BlobSettings au moment où l'objet BlobDistributedMutex est créé (cet objet est une structure simple qui est incluse dans l'exemple de code). Le constructeur accepte également une tâche qui référence le code à exécuter par l'instance de rôle si elle obtient le bail du blob et est élue pour être le leader. Notez que le code qui gère les détails de bas niveau sur l'acquisition du bail est implémenté dans une classe d'assistance distincte nommée BlobLeaseManager.

```
public class BlobDistributedMutex
{
  ...
  private readonly BlobSettings blobSettings;
  private readonly Func<CancellationToken, Task> taskToRunWhenLeaseAcquired;
  ...

  public BlobDistributedMutex(BlobSettings blobSettings,
           Func<CancellationToken, Task> taskToRunWhenLeaseAquired)
  {
    this.blobSettings = blobSettings;
    this.taskToRunWhenLeaseAquired = taskToRunWhenLeaseAquired;
  }

  public async Task RunTaskWhenMutexAcquired(CancellationToken token)
  {
    var leaseManager = new BlobLeaseManager(blobSettings);
    await this.RunTaskWhenBlobLeaseAcquired(leaseManager, token);
  }
  ...
```

La méthode RunTaskWhenMutexAquired dans l'exemple de code ci-dessus appelle la méthode RunTaskWhenBlobLeaseAcquired illustrée dans l'exemple de code suivant pour acquérir le bail. La méthode RunTaskWhenBlobLeaseAcquired est exécutée en mode asynchrone. Si l'acquisition du bail aboutit, l'instance de rôle est élue pour être le leader. Le délégué taskToRunWhenLeaseAcquired a pour objectif d'exécuter les tâches qui coordonnent les autres instances de rôle. Si l'acquisition du bail échoue, une autre instance de rôle est élue pour être le leader et l'instance de rôle actuelle reste une instance subordonnée. Notez que la méthode TryAcquireLeaseOrWait est une méthode d'assistance qui utilise l'objet BlobLeaseManager pour acquérir le bail.

```
private async Task RunTaskWhenBlobLeaseAcquired(
    BlobLeaseManager leaseManager, CancellationToken token)
  {
    while (!token.IsCancellationRequested)
    {
      // Try to acquire the blob lease.
      // Otherwise wait for a short time before trying again.
      string leaseId = await this.TryAquireLeaseOrWait(leaseManager, token);

      if (!string.IsNullOrEmpty(leaseId))
      {
        // Create a new linked cancellation token source so that if either the
        // original token is canceled or the lease can't be renewed, the
        // leader task can be canceled.
        using (var leaseCts =
          CancellationTokenSource.CreateLinkedTokenSource(new[] { token }))
        {
          // Run the leader task.
          var leaderTask = this.taskToRunWhenLeaseAquired.Invoke(leaseCts.Token);
          ...
        }
      }
    }
    ...
  }
```

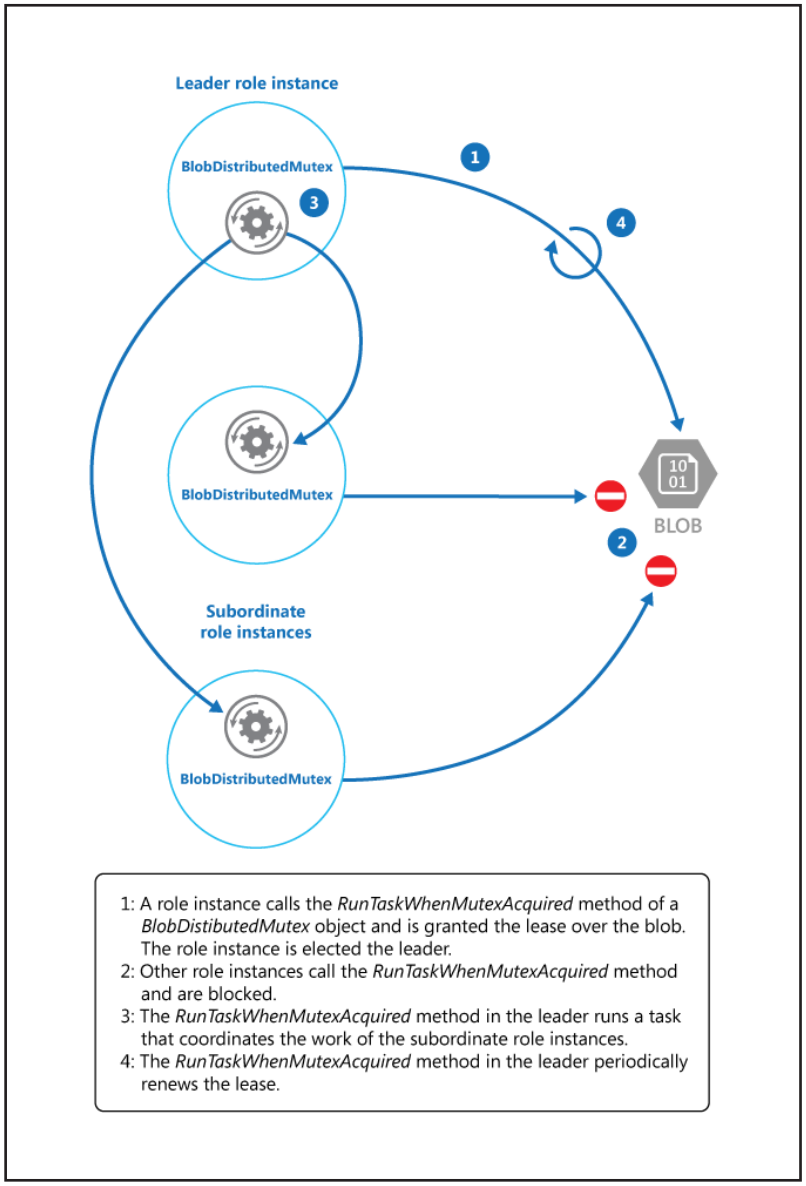
La tâche démarrée par le leader s'exécute également en mode asynchrone. Pendant l'exécution de la tâche, la méthode RunTaskWhenBlobLeaseAquired illustrée dans l'exemple de code suivant tente périodiquement de renouveler le bail. Cela permet de s'assurer que l'instance de rôle reste le leader. Dans l'exemple de la solution, le délai entre les demandes de renouvellement est inférieur à la durée spécifiée pour le bail afin d'éviter qu'une autre instance de rôle ne soit élue pour être le leader. Si le renouvellement échoue pour une raison quelconque, la tâche est annulée.

Si le renouvellement du bail échoue ou que la tâche est annulée (peut-être en raison de l'arrêt de l'instance de rôle), le bail est libéré. À ce stade, cette instance, ou une autre, peut être élue pour être le leader. L'extrait de code ci-dessous illustre cette partie du processus.

```
private async Task RunTaskWhenBlobLeaseAcquired(
    BlobLeaseManager leaseManager, CancellationToken token)
  {
    while (...)
    {
      ...
      if (...)
      {
        ...
        using (var leaseCts = ...)
        {
          ...
          // Keep renewing the lease in regular intervals.
          // If the lease can't be renewed, then the task completes.
          var renewLeaseTask =
            this.KeepRenewingLease(leaseManager, leaseId, leaseCts.Token);

          // When any task completes (either the leader task itself or when it
          // couldn't renew the lease) then cancel the other task.
          await CancelAllWhenAnyCompletes(leaderTask, renewLeaseTask, leaseCts);
        }
      }
    }
  }
  ...
}
```

La méthode KeepRenewingLease est une autre méthode d'assistance qui utilise l'objet BlobLeaseManager pour renouveler le bail. La méthode CancelAllWhenAnyCompletes annule les tâches spécifiées dans les deux premiers paramètres. Le diagramme suivant illustre l'utilisation de la classe BlobDistributedMutex pour élire un leader et exécuter une tâche qui coordonne les opérations.



L'exemple de code suivant illustre la manière d'utiliser la classe BlobDistributedMutex dans un rôle de travail. Ce code permet d'acquérir le bail d'un objet blob nommé MyLeaderCoordinatorTask dans le conteneur du bail du stockage de développement et indique que le code défini dans la méthode MyLeaderCoordinatorTask doit être exécuté si l'instance de rôle est élue pour être le leader.

```
var settings = new BlobSettings(CloudStorageAccount.DevelopmentStorageAccount,
  "leases", "MyLeaderCoordinatorTask");
var cts = new CancellationTokenSource();
var mutex = new BlobDistributedMutex(settings, MyLeaderCoordinatorTask);
mutex.RunTaskWhenMutexAcquired(this.cts.Token);
...

// Method that runs if the role instance is elected the leader
private static async Task MyLeaderCoordinatorTask(CancellationToken token)
{
  ...
}
```

Notez les points suivants dans l'exemple de solution :

- Le blob est un point de défaillance unique potentiel. Si le service blob devient indisponible, ou inaccessible, le leader ne pourra pas renouveler le bail et aucune autre instance de rôle ne sera en mesure d'acquérir le bail. Dans ce cas, aucune instance de rôle ne sera en mesure d'agir à titre de leader. Toutefois, comme le service blob est conçu pour résister, une défaillance complète du service blob est hautement improbable.
- Si la tâche exécutée par le leader n'aboutit pas, il peut continuer de renouveler le bail, ce qui empêche les autres instances de rôle d'acquérir le bail et de reprendre le rôle de leader afin de coordonner les tâches. En réalité, l'intégrité du leader doit être vérifiée à intervalles réguliers.
- Le processus d'élection est non déterministe. Vous ne pouvez pas deviner quelle instance de rôle pourra acquérir le bail de l'objet blob et devenir le leader.
- Le blob, utilisé comme cible du bail, ne doit pas être utilisé à d'autres fins. Si une instance de rôle stocke des données dans cet objet blob, ces données ne seront pas accessibles à moins que l'instance de rôle ne soit le leader et ne détienne le bail de l'objet blob.

Patrons et informations connexes

Les informations suivantes peuvent également être utiles lors de l'implémentation de ce patron :

- Ce patron comprend un exemple d'application téléchargeable.
- Conseils sur la mise à l'échelle automatique. Il est possible de démarrer et d'arrêter les instances des hôtes de tâche lorsque la charge de l'application varie. La mise à l'échelle automatique permet de maintenir le débit et les performances pendant les périodes de pics de traitement.
- Conseils sur le partitionnement des calculs. Cette section décrit comment allouer les tâches aux hôtes dans un service cloud dans le but de minimiser les coûts d'exploitation tout en préservant l'évolutivité, les performances, la disponibilité et la sécurité du service.
- Patron asynchrone basé sur les tâches.
- Exemple illustrant l'algorithme d'élection Bully.
- Exemple illustrant l'algorithme d'élection Ring.
- Apache Curator, une bibliothèque cliente pour Apache ZooKeeper.
- L'article Location d'un blob (REST API) sur MSDN.

Patron Materialized View (Vue matérialisée)

Générez des vues préremplies sur les données dans un ou plusieurs magasins de données lorsque les données ne sont pas idéalement formatées pour des opérations de requête requises. Cela permet de soutenir l'efficacité du système d'interrogation et d'extraction des données et d'améliorer les performances des applications.

Contexte et problème

Pendant le stockage des données, la priorité, pour les développeurs et les administrateurs de données, est souvent centrée sur le mode de stockage des données, par opposition au mode d'accès. Le format de stockage choisi est généralement étroitement lié au format des données, aux critères de gestion du volume de données et de l'intégrité des données, et du type de stockage actuellement utilisé. Par exemple, lorsque vous utilisez le magasin de documents NoSQL, les données sont souvent représentées sous la forme d'une série d'agrégats, chacun contenant toutes les informations de cette entité.

Toutefois, cela peut avoir un impact négatif sur les requêtes. Lorsqu'une requête a besoin uniquement d'un sous-ensemble de données de certaines entités, par exemple un résumé des commandes de plusieurs clients sans tous les détails de la commande, elle doit extraire toutes les données des entités concernées pour obtenir les informations recherchées.

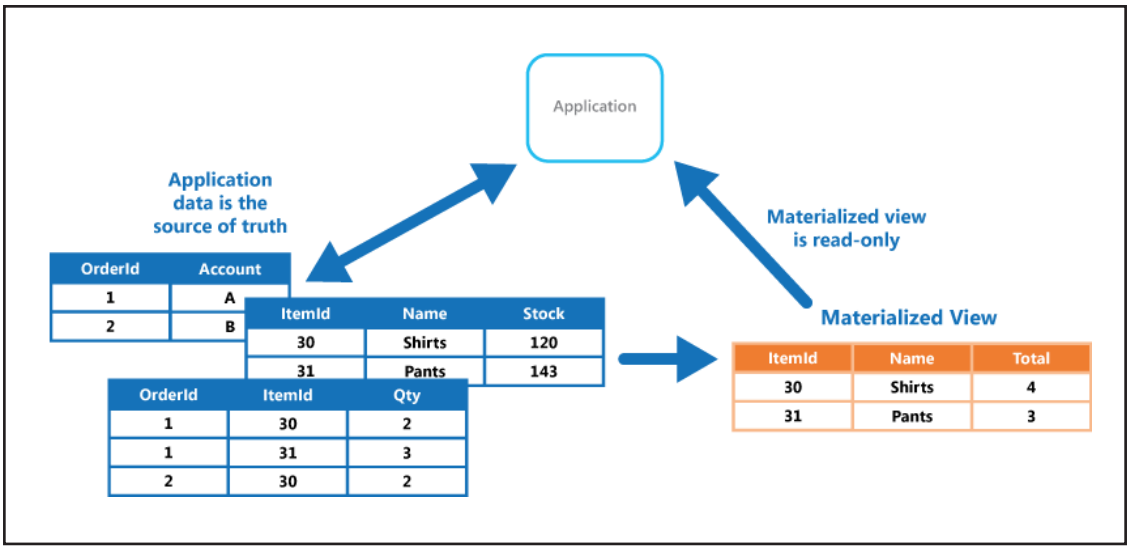
Solution

Pour prendre en charge un système d'interrogation efficace, une solution courante consiste à générer, à l'avance, une vue qui matérialise les données dans un format adapté à l'ensemble des résultats requis. Le patron Materialized View (Vue matérialisée) décrit la génération de vues de données préremplies dans des environnements où le format des données source n'est pas adapté à l'interrogation, où la génération d'une requête appropriée est difficile et où les performances des requêtes sont médiocres en raison de la nature des données ou du magasin de données.

Ces vues matérialisées, qui ne contiennent que les données requises par les requêtes, permettent aux applications d'obtenir rapidement les informations dont elles ont besoin. En plus de la possibilité de joindre les tables ou de combiner les entités de données, les vues matérialisées peuvent inclure les valeurs actuelles des colonnes calculées ou des éléments de données, les résultats de la combinaison des valeurs ou de l'exécution des transformations sur les éléments de données, et les valeurs spécifiées dans le cadre de la requête. Une vue matérialisée peut même être optimisée pour juste une seule requête.

Le plus important c'est qu'une vue matérialisée et les données qu'elle contient peuvent être entièrement supprimées, car elles peuvent être entièrement recréées dans les magasins de données source. Une vue matérialisée n'est jamais mise à jour directement par une application ; il s'agit donc d'un cache spécialisé.

Lorsque les données source de la vue sont mises à jour, la vue doit être actualisée pour refléter les nouvelles données. Vous pouvez planifier cette opération en mode automatique ou chaque fois que le système identifie une modification des données d'origine. Dans certains cas, il peut être nécessaire de regénérer manuellement la vue. La figure illustre un mode d'utilisation du patron Materialized View (Vue matérialisée).



Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Modalités et périodes d'actualisation. L'idéal c'est la regénération en réponse à un événement indiquant une modification des données source, même si cela peut conduire à une surcharge du système si les données source changent fréquemment. Pour regénérer la vue, vous pouvez également envisager d'utiliser une tâche planifiée, un déclencheur externe ou une opération manuelle.

Sur certains systèmes, lorsqu'il s'agit d'utiliser le patron Event Sourcing (Matérialisation d'événements) pour maintenir un magasin contenant uniquement les événements qui ont modifié les données, les vues matérialisées sont nécessaires. Le préremplissage des vues, avec analyse de tous les événements pour déterminer l'état actuel, peut être la seule manière d'obtenir les informations du magasin d'événements. Si vous n'utilisez pas le patron de matérialisation d'événements, vous devez examiner l'utilité d'avoir recours ou non à une vue matérialisée. Les vues matérialisées ont tendance à être spécifiquement conçues pour un petit nombre de requêtes, voire une seule. Si vous utilisez de nombreuses requêtes, les vues matérialisées peuvent entraîner des contraintes en matière de capacité de stockage et de coût de stockage inacceptables.

Prenez en compte l'impact sur la cohérence des données lorsque vous générez une vue et mettez à jour la vue si l'opération est planifiée. Si les données source changent au moment où la vue est générée, la copie des données de la vue ne reflètera pas entièrement les données d'origine.

Vous devez réfléchir à l'emplacement de stockage de la vue. La vue ne doit pas se trouver dans le même magasin ou la même partition que les données d'origine. Elle peut être un sous-ensemble de plusieurs partitions différentes combinées.

Si elle est perdue, une vue peut être recréée. Pour cette raison, si la vue est temporaire et n'est utilisée que pour améliorer les performances des requêtes qui rendent compte de l'état actuel des données, ou pour améliorer l'évolutivité, elle peut être stockée dans une mémoire cache ou dans un emplacement moins fiable.

Lorsque vous définissez une vue matérialisée, maximisez sa valeur en lui ajoutant des éléments de données ou des colonnes en fonction du calcul ou de la transformation des éléments de données existants, des valeurs transmises dans la requête ou des combinaisons de ces valeurs lorsque cela s'avère nécessaire.

Lorsque le système de stockage prend en charge l'indexation, vous pouvez indexer la vue matérialisée pour améliorer les performances. La plupart des bases de données relationnelles prennent en charge l'indexation des vues, comme les solutions Big Data basées sur Apache Hadoop.

Quand utiliser ce patron

Ce patron est utile dans les cas suivants :

- Création de vues matérialisées de données qui sont difficiles à interroger directement, ou dans le cas de requêtes très complexes utilisées pour extraire les données stockées de manière standard dans un format semi-structuré ou non structuré.

- Création de vues temporaires qui peuvent améliorer considérablement les performances des requêtes ou qui peuvent agir directement comme vues source ou objets de transfert de données pour l'interface utilisateur, sous forme de rapports ou à l'écran.

- Prise en charge occasionnelle de scénarios en mode connecté ou déconnecté dans lesquels la connexion au magasin de données n'est pas toujours disponible. La vue peut être mise en cache localement dans ce cas.

- Simplification des requêtes et exposition des données à titre d'essai, sans exiger l'identification du format des données source. Par exemple, en joignant différentes tables dans une ou plusieurs bases de données, ou dans un ou plusieurs domaines de magasins NoSQL, puis en mettant en forme les données en fonction de leur utilisation finale.

- Fourniture d'accès à des sous-ensembles de données source spécifiques qui, pour des raisons de sécurité ou de confidentialité, ne doivent pas être accessibles, modifiables ou entièrement exposés aux utilisateurs.

- Pontage de différents magasins de données pour profiter de leurs fonctions individuelles. Par exemple, l'utilisation d'un magasin dans le cloud, efficace pour les opérations d'écriture, comme magasin de données de référence et d'une base de données relationnelle qui offre un bon système d'interrogation et de belles performances en lecture pour contenir les vues matérialisées.

Ce patron n'est pas utile dans les situations suivantes :

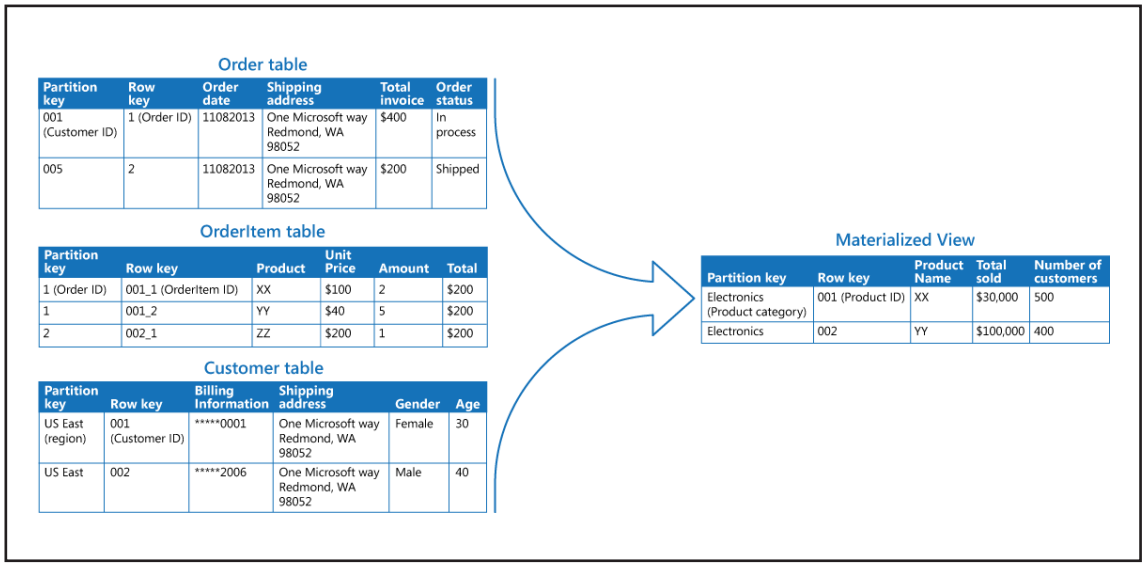
- Les données source sont simples et faciles à interroger.

- Les données source changent rapidement ou sont accessibles sans utiliser de vue. Dans ces cas, vous devez éviter les charges de traitement liées à la création de vues.

- La cohérence est une priorité élevée. Les vues ne reflètent pas toujours avec exactitude les données d'origine.

Exemple

La figure suivante illustre l'utilisation du patron Materialized View (Vue matérialisée) pour générer une synthèse des ventes. Les données des tables Order, OrderItem et Customer sur différentes partitions d'un compte de stockage Azure sont combinées pour produire une vue contenant la valeur totale des ventes de chaque produit dans la catégorie des produits électroniques, ainsi que le nombre de clients qui ont effectué des achats pour chaque produit.



La création de cette vue matérialisée nécessite des requêtes complexes. Cependant, en présentant le résultat de la requête comme vue matérialisée, les utilisateurs peuvent obtenir facilement ces résultats et les utiliser directement, ou peuvent les incorporer dans une autre requête. La vue doit être utilisée dans un système de génération de rapports ou dans un tableau de bord, et peut être mise à jour régulièrement, chaque semaine par exemple.

Bien que cet exemple utilise le mode de stockage de tables Azure, de nombreux systèmes de gestion de bases de données relationnelles offrent également une prise en charge native des vues matérialisées.

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Manuel de cohérence de données. Le résumé d'informations dans une vue matérialisée doit être tenu à jour pour qu'elle reflète les valeurs des données sous-jacentes. Comme la valeur des données change, il n'est peut-être pas pratique de mettre à jour les données de synthèse en temps réel. Il serait préférable finalement d'adopter une méthode qui privilégie la cohérence. Résume les questions autour du respect de la cohérence des données distribuées et décrit les avantages et les inconvénients des différents modèles de cohérence.
- Patron CQRS - Command and Query Responsibility Segregation (séparation des responsabilités commande / requête). Ce patron est utilisé pour mettre à jour les informations d'une vue matérialisée en répondant aux événements qui se produisent lorsque les valeurs des données sous-jacentes sont modifiées.
- Patron Event Sourcing (Matérialisation d'événements). Ce patron est utilisé conjointement avec le patron CQRS pour gérer les données d'une vue matérialisée. Lorsque les valeurs de données d'une vue matérialisée sont modifiées, le système peut déclencher des événements qui décrivent ces changements et les enregistrer dans un magasin d'événements.

- Patron Index Table (Tableau indexé). Les données d'une vue matérialisée sont généralement organisées avec une clé primaire, mais les requêtes doivent extraire les informations de cette vue en examinant les données d'autres champs. Il sert à créer des index secondaires sur des ensembles de données pour les magasins de données qui ne prennent pas en charge les index secondaires natifs.

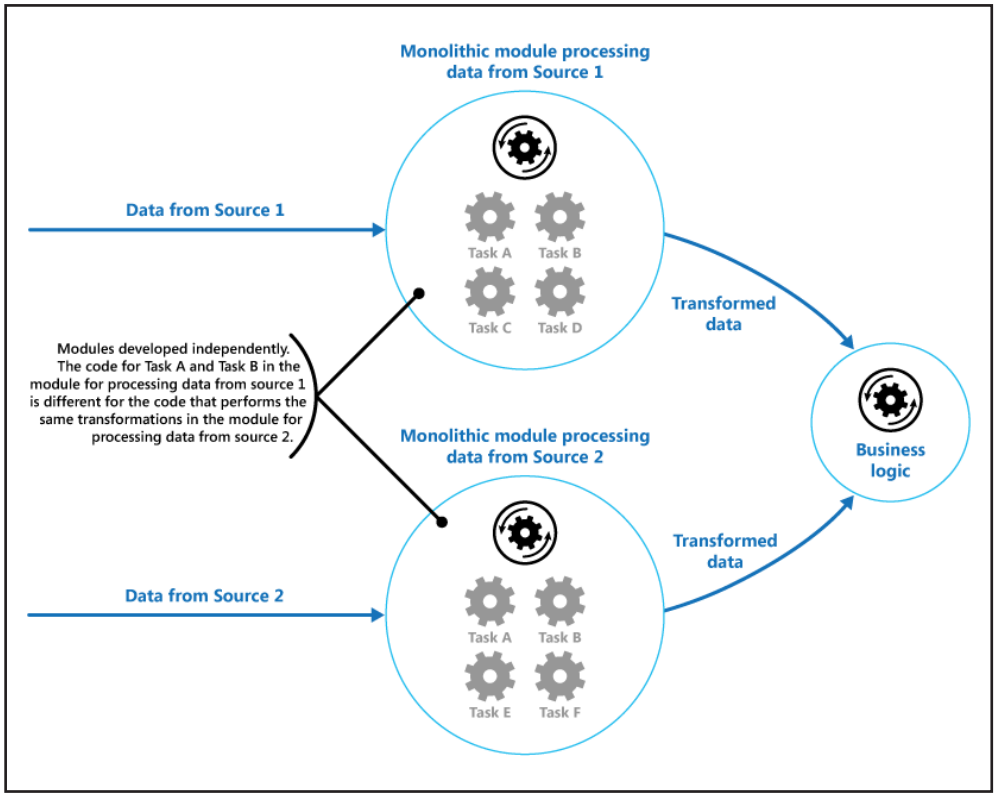
Patron Pipes and Filters (Tubes et filtres)

Décompose une tâche qui effectue un traitement complexe en une série d'éléments discrets qui peuvent être réutilisés. Ce patron peut améliorer les performances, l'évolutivité et la réutilisabilité en permettant aux éléments de la tâche qui effectuent le traitement d'être déployés et mis à l'échelle de façon indépendante.

Contexte et problème

Une application doit effectuer une série de tâches de complexité variable pour les informations qu'elle traite. Une méthode simple mais rigide de mise en œuvre d'une application consiste à effectuer ce traitement comme module monolithique. Toutefois, cette méthodologie risque de réduire les possibilités de refactorisation du code, d'optimisation ou de réutilisation si des éléments du même traitement sont nécessaires ailleurs dans l'application.

La figure illustre les problèmes de traitement des données avec cette méthode monolithique. Une application reçoit et traite les données provenant de deux sources. Les données de chaque source sont traitées par un module distinct qui effectue une série de tâches pour transformer ces données, avant de transmettre le résultat à la logique métier de l'application.

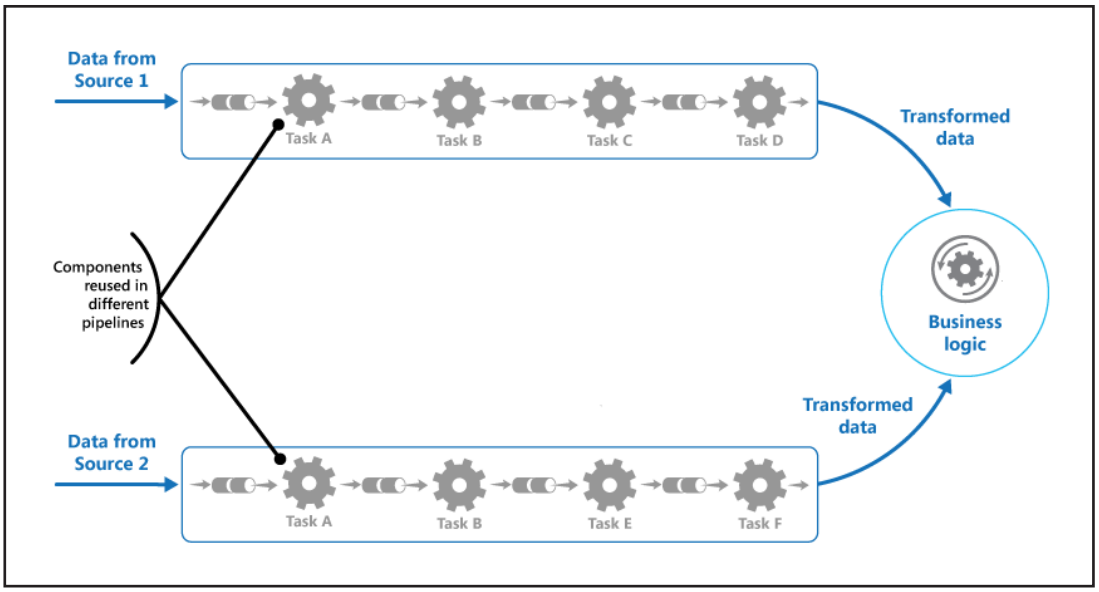


Certaines tâches exécutées par les modules monolithiques sont semblables sur le plan fonctionnel, pourtant les modules ont été conçus séparément. Le code qui implémente les tâches est étroitement associé à un module et a été développé avec très peu ou aucune réflexion sur la réutilisabilité ou l'évolutivité.

Toutefois, les tâches de traitement exécutées par chaque module ou les exigences de déploiement pour chaque tâche, pourraient changer en fonction de l'évolution des besoins de l'entreprise. Certaines tâches à calculs intensifs gagneraient à être exécutées sur des machines puissantes, tandis que d'autres n'ont pas besoin de faire appel à ces ressources coûteuses. Aussi, un traitement supplémentaire peut être nécessaire à l'avenir, ou l'ordre d'exécution des tâches par le système pourrait changer. Une solution est nécessaire pour répondre à ces questions et augmenter les possibilités de réutilisation du code.

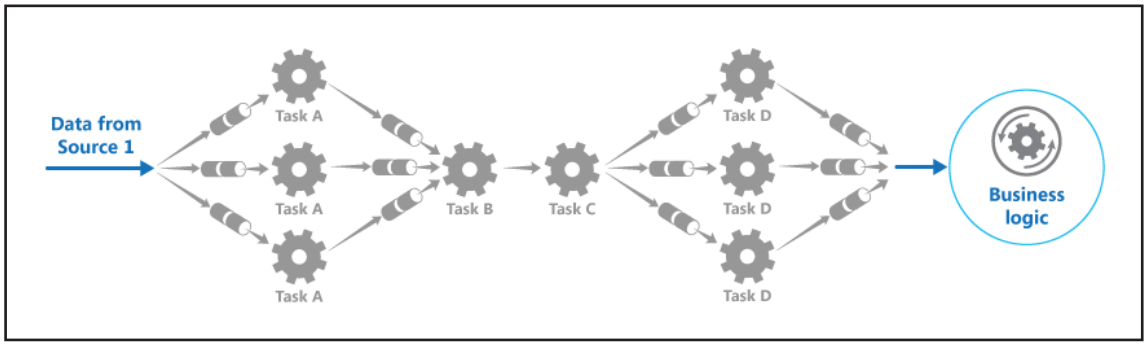
Solution

Décomposez les opérations de traitement requises pour chaque flux dans un ensemble de composants (ou filtres) distincts, chacun exécutant une seule tâche. En standardisant le format des données que chaque composant reçoit et envoie, ces filtres peuvent être combinés dans un pipeline. Cela permet d'éviter la duplication du code et facilite la suppression, le remplacement ou l'intégration de composants supplémentaires si les exigences de traitement évoluent. La figure suivante illustre une solution implémentée à l'aide du patron de tuyaux et de filtres.



Le délai de traitement d'une seule requête varie en fonction de la vitesse du filtre le plus lent dans le pipeline. Un ou plusieurs filtres pourraient former un goulot d'étranglement, en particulier si un nombre élevé de requêtes se retrouvent dans un flux de données provenant d'une source de données particulière. L'avantage principal de la structure du pipeline est la possibilité d'exécuter des instances parallèles de filtres lents, ce qui permet au système de répartir la charge et d'améliorer le débit.

Les filtres qui forment un pipeline peuvent fonctionner sur des machines différentes, ce qui leur permet d'être redimensionnées de façon indépendante et de profiter de l'extraordinaire évolutivité que fournissent de nombreux environnements dans le cloud. Un filtre à calculs intensifs peut fonctionner sur du matériel de haute performance, tandis que d'autres filtres moins gourmands peuvent être hébergés sur du matériel moins cher. Il n'est pas nécessaire de regrouper les filtres dans le même centre de données ou emplacement géographique. Ainsi, chaque élément d'un pipeline peut être exécuté dans un environnement qui se trouve à proximité des ressources requises. La figure suivante illustre un exemple appliqué au pipeline des données de la source 1.



Si l'entrée et la sortie d'un filtre sont structurées comme un flux, il est possible d'effectuer un traitement en parallèle pour chaque filtre. Le premier filtre du pipeline peut démarrer ses travaux et afficher ses résultats qui sont transmis directement au filtre suivant dans la séquence, avant même que le premier filtre n'ait terminé sa tâche.

Un autre avantage est la résilience que présente ce modèle. Si un filtre échoue ou que la machine sur laquelle il est exécuté n'est plus disponible, le pipeline peut replanifier le travail de ce filtre et l'attribuer à une autre instance du composant. La défaillance d'un seul filtre ne provoque pas nécessairement la défaillance entière du pipeline.

L'utilisation du patron Pipes and Filters (Tubes et filtres) conjointement avec le Patron Compensating Transaction (Transaction de compensation) est une alternative à l'implémentation de transactions distribuées. Une transaction distribuée peut être décomposée en tâches distinctes compensables dont chacune peut être mise en œuvre à l'aide d'un filtre qui implémente également le patron Compensating Transaction (Transaction de compensation). Les filtres d'un pipeline peuvent être implémentés comme tâches hébergées distinctes et exécutées à proximité des données qu'elles gèrent.

Problèmes et considérations

Prenez en compte les points suivants lorsque vous choisissez le mode d'implémentation de ce patron :

- **Complexité.** La flexibilité accrue qu'offre ce patron peut aussi introduire un certain degré de complexité, surtout si les filtres d'un pipeline sont répartis sur plusieurs serveurs.
- **Fiabilité.** Utilisez une infrastructure qui permet de s'assurer que les données circulant entre les filtres d'un pipeline ne seront pas perdues.
- **Idempotence.** Si un filtre dans un pipeline échoue après avoir reçu un message et que le travail est replanifié pour une autre instance du filtre, une partie de ce travail a peut-être déjà été réalisée. Si ce travail met à jour certains aspects de l'état global (par exemple, les informations stockées dans une base de données), la même mise à jour pourrait se répéter. Un problème similaire peut se produire si un filtre échoue après la publication de ses résultats dans le filtre suivant du pipeline, mais avant l'indication que la tâche s'est déroulée correctement. Dans ce cas, le même travail pourrait être répété par une autre instance du filtre, entraînant ainsi la publication en double des mêmes résultats. Cela peut entraîner au niveau des filtres suivants du pipeline le traitement en double des mêmes données. Par conséquent, les filtres d'un pipeline doivent intégrer l'idempotence. Pour plus d'informations, consultez Patrons d'idempotence sur le blog de Jonathan Oliver.
- **Messages répétés.** Si un filtre dans un pipeline échoue après avoir envoyé un message à l'étape suivante du pipeline, une autre instance du filtre peut être exécutée et publier une copie du même message pour le pipeline. Cela entraîne l'envoi au filtre suivant du même message par deux instances. Pour éviter cela, le pipeline doit détecter et éliminer les messages en double.

Si vous mettez en œuvre le pipeline en utilisant des files d'attente de messages (par exemple les files d'attente Service Bus de Microsoft Azure), l'infrastructure de mise en file d'attente des messages peut offrir un système de détection automatique des messages en double.

- **Contexte et état.** Dans un pipeline, chaque filtre fonctionne essentiellement en parallèle et ignore la manière dont il a été appelé. Cela signifie que chaque filtre doit être assorti d'un contexte suffisant pour effectuer son travail. Ce contexte peut inclure un volume important d'informations d'état.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

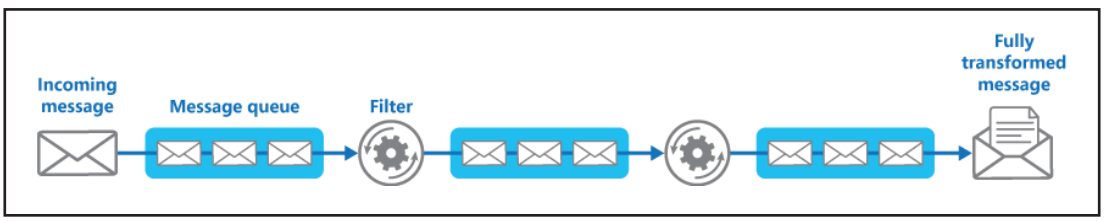
- Le traitement requis par une application peut facilement être décomposé en une série d'étapes indépendantes.
- Les étapes de traitement effectuées par une application répondent à des critères d'évolutivité différents. Vous pouvez regrouper les filtres qui doivent être redimensionnés ensemble dans le même processus. Pour plus d'informations, reportez-vous au patron Compute Resource Consolidation (Consolidation des ressources de calcul).
- Une certaine flexibilité est nécessaire pour permettre la réorganisation des étapes du traitement effectuées par une application, ainsi que la possibilité d'ajouter et de supprimer des étapes.
- Le système peut bénéficier de la répartition du traitement des étapes sur différents serveurs.
- Une solution fiable est nécessaire pour minimiser l'impact des défaillances d'une étape pendant le traitement des données.

Ce patron peut ne pas être approprié quand :

- Les étapes de traitement exécutées par une application ne sont pas indépendantes ou doivent être exécutées simultanément dans le cadre de la même transaction.
- La quantité d'informations de contexte ou d'état requises par une étape rend cette méthode inefficace. Il est possible à la place de conserver les informations d'état dans une base de données, mais n'utilisez pas cette stratégie si la charge supplémentaire sur la base de données entraîne un problème de contention excessif.

Exemple

Vous pouvez utiliser une séquence de files d'attente des messages pour fournir l'infrastructure nécessaire à la mise en œuvre d'un pipeline. Une file d'attente des messages initial reçoit des messages non traités. Un composant implémenté comme tâche de filtre reste à l'écoute d'un message sur cette file d'attente, effectue sa tâche, puis envoie le message transformé à la file d'attente suivante dans la séquence. Une autre tâche de filtre peut se mettre à l'écoute de messages dans cette file d'attente, les traiter, afficher les résultats dans une autre file d'attente, et ainsi de suite jusqu'à ce que les données entièrement transformées s'affichent dans le dernier message de la file d'attente. La figure suivante illustre la mise en œuvre d'un pipeline à l'aide des files d'attente de messages.



Si vous créez une solution sur Azure, vous pouvez utiliser les files d'attente Service Bus pour profiter d'un système de mise en file d'attente à la fois fiable et évolutif. La classe ServiceBusPipeFilter ci-dessous en langage C# illustre la manière d'implémenter un filtre qui reçoit des messages d'une file d'attente, traite ces messages et envoie les résultats à une autre file d'attente.

La classe ServiceBusPipeFilter est définie dans le projet PipesAndFilters.Shared disponible sur GitHub.

```
public class ServiceBusPipeFilter
{
  ...
  private readonly string inQueuePath;
  private readonly string outQueuePath;
  ...
  private QueueClient inQueue;
  private QueueClient outQueue;
  ...

  public ServiceBusPipeFilter(..., string inQueuePath, string outQueuePath = null)
  {
     ...
     this.inQueuePath = inQueuePath;
     this.outQueuePath = outQueuePath;
  }

  public void Start()
  {
    ...
    // Create the outbound filter queue if it doesn't exist.
    ...
    this.outQueue = QueueClient.CreateFromConnectionString(...);

    ...
    // Create the inbound and outbound queue clients.
    this.inQueue = QueueClient.CreateFromConnectionString(...);
  }

  public void OnPipeFilterMessageAsync(
    Func<BrokeredMessage, Task<BrokeredMessage>> asyncFilterTask, ...)
  {
    ...

    this.inQueue.OnMessageAsync(
      async (msg) =>
    {
      ...
      // Process the filter and send the output to the
      // next queue in the pipeline.
      var outMessage = await asyncFilterTask(msg);

      // Send the message from the filter processor
      // to the next queue in the pipeline.
      if (outQueue != null)
      {
        await outQueue.SendAsync(outMessage);
      }

      // Note: There's a chance that the same message could be sent twice
      // or that a message gets processed by an upstream or downstream
      // filter at the same time.
      // This would happen in a situation where processing of a message was
      // completed, it was sent to the next pipe/queue, and then failed
      // to complete when using the PeekLock method.
      // Idempotent message processing and concurrency should be considered
      // in a real-world implementation.
    },
    options);
  }

  public async Task Close(TimeSpan timespan)
  {
    // Pause the processing threads.
    this.pauseProcessingEvent.Reset();

    // There's no clean approach for waiting for the threads to complete
    // the processing. This example simply stops any new processing, waits
    // for the existing thread to complete, then closes the message pump
    // and finally returns.
    s Thread.Sleep(timespan);

    this.inQueue.Close();
    ...
  }
```

La méthode Start de la classe ServiceBusPipeFilter se connecte à une paire de files d'attente d'entrée et de sortie, et la méthode Close se déconnecte de la file d'attente d'entrée. La méthode OnPipeFilterMessageAsync exécute le traitement réel des messages ; le paramètre asyncFilterTask associé à cette méthode désigne la transformation à exécuter. La méthode OnPipeFilterMessageAsync attend les messages entrants dans la file d'attente d'entrée, exécute le code spécifié par le paramètre asyncFilterTask sur chaque message, au fur et à mesure qu'ils arrivent, et envoie les résultats dans la file d'attente de sortie. Les files d'attente sont spécifiées par le constructeur.

L'exemple de solution illustre l'implémentation des filtres dans un jeu de rôles de travail. Chaque rôle de travail peut être mis à l'échelle indépendamment, en fonction de la complexité des transactions commerciales effectuées ou des ressources nécessaires au traitement. En outre, plusieurs instances de chaque rôle de travail peuvent être exécutées en parallèle pour améliorer le débit.

Le code suivant illustre un rôle de travail Azure, nommé PipeFilterARoleEntry, défini dans le projet PipeFilterA de l'exemple de solution.

```
public class PipeFilterARoleEntry : RoleEntryPoint
{
  ...
  private ServiceBusPipeFilter pipeFilterA;

  public override bool OnStart()
  {
    ...
    this.pipeFilterA = new ServiceBusPipeFilter(
      ...,
      Constants.QueueAPath,
      Constants.QueueBPath);

    this.pipeFilterA.Start();
    ...
  }

  public override void Run()
  {
    this.pipeFilterA.OnPipeFilterMessageAsync(async (msg) =>
    {
      // Clone the message and update it.
      // Properties set by the broker (Deliver count, enqueue time, ...)
      // aren't cloned and must be copied over if required.
      var newMsg = msg.Clone();

      await Task.Delay(500); // DOING WORK

      Trace.TraceInformation("Filter A processed message:{0} at {1}",
        msg.MessageId, DateTime.UtcNow);

      newMsg.Properties.Add(Constants.FilterAMessageKey, "Complete");

      return newMsg;
    });

    ...
  }

  ...
}
```

Ce rôle contient un objet ServiceBusPipeFilter. La méthode OnStart du rôle se connecte aux files d'attente pour recevoir les messages d'entrée et envoyer les messages de sortie (le nom des files d'attente est défini dans la classe Constants). La méthode Run appelle la méthode OnPipeFilterMessagesAsync pour effectuer un traitement sur chaque message reçu (dans cet exemple, le traitement est simulé avec une attente sur une courte période). Quand le traitement est terminé, un nouveau message est créé avec les résultats (dans ce cas, une propriété personnalisée est ajoutée au message d'entrée), et ce message est envoyé à la file d'attente de sortie.

L'exemple de code contient un autre rôle de travail nommé PipeFilterBRoleEntry dans le projet PipeFilterB. Ce rôle est semblable à PipeFilterARoleEntry, sauf qu'il effectue un traitement différent dans la méthode Run. Dans l'exemple de solution, ces deux rôles sont combinés pour construire un pipeline. La file d'attente de sortie du rôle PipeFilterARoleEntry correspond à la file d'attente d'entrée du rôle PipeFilterBRoleEntry.

L'exemple de solution fournit également deux autres rôles nommés InitialSenderRoleEntry (dans le projet InitialSender) et FinalReceiverRoleEntry (dans le projet FinalReceiver). Le rôle InitialSenderRoleEntry fournit le message initial dans le pipeline. La méthode OnStart se connecte à une seule file d'attente et la méthode Run envoie une méthode dans cette file d'attente. Cette file d'attente correspond à la file d'attente d'entrée utilisée par le rôle de PipeFilterARoleEntry. Ainsi, l'envoi d'un message entraîne la réception et le traitement de ce message par le rôle PipeFilterARoleEntry. Le message traité est alors transmis via le rôle PipeFilterBRoleEntry.

La file d'attente d'entrée du rôle FinalReceiveRoleEntry correspond à la file d'attente de sortie du rôle PipeFilterBRoleEntry. La méthode Run du rôle FinalReceiveRoleEntry, illustrée ci-dessous, reçoit le message et effectue un traitement final. Elle enregistre ensuite les valeurs des propriétés personnalisées ajoutées par les filtres dans le pipeline à la sortie de trace.

```
public class FinalReceiverRoleEntry : RoleEntryPoint
{
  ...
  // Final queue/pipe in the pipeline to process data from.
  private ServiceBusPipeFilter queueFinal;

  public override bool OnStart()
  {
    ...
    // Set up the queue.
    this.queueFinal = new ServiceBusPipeFilter(...,Constants.QueueFinalPath);
    this.queueFinal.Start();
    ...
  }

  public override void Run()
  {
    this.queueFinal.OnPipeFilterMessageAsync(
      async (msg) =>
      {
        await Task.Delay(500); // DOING WORK

        // The pipeline message was received.
        Trace.TraceInformation(
          "Pipeline Message Complete - FilterA:{0} FilterB:{1}",
          msg.Properties[Constants.FilterAMessageKey],
          msg.Properties[Constants.FilterBMessageKey]);

        return null;
      });
    ...
  }

  ...
}
```

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Un exemple illustrant ce patron est disponible sur GitHub.

- Patron Competing Consumers (Consommateurs concurrents) Un pipeline peut contenir plusieurs instances d'un ou plusieurs filtres. Cette méthode est pratique pour exécuter en parallèle des instances de filtres lents, ce qui permet au système de répartir la charge et d'améliorer le débit. Chaque instance d'un filtre est en compétition à l'entrée avec les autres instances ; deux instances d'un filtre ne peuvent pas traiter les mêmes données. Fournit une explication de cette méthode.

- Patron Compute Resource Consolidation (consolidation des ressources de calcul). Vous pouvez regrouper les filtres qui doivent être redimensionnés ensemble dans le même processus. Fournit plus d'informations sur les avantages et les inconvénients de cette stratégie.
- Patron Compensating Transaction (Transaction de compensation). Vous pouvez mettre en œuvre un filtre sous la forme d'une opération qui peut être inversée, ou qui contient une opération de compensation qui rétablit l'état à une version précédente en cas de défaillance. Explique comment l'implémentation peut être réalisée pour maintenir ou parvenir à une cohérence à terme.
- Patrons d'idempotence sur le blog de Jonathan Oliver.

Patron Priority Queue (File d'attente prioritaire)

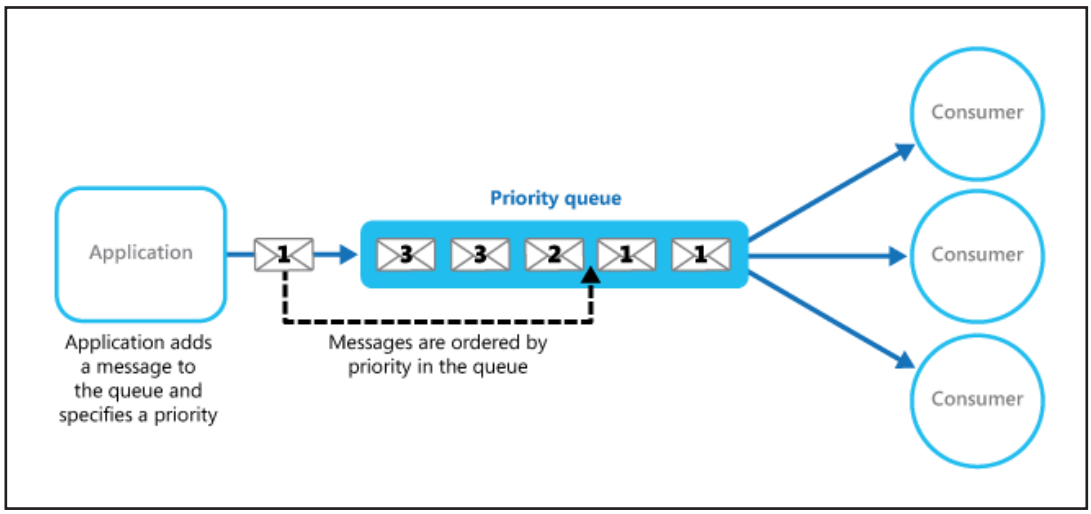
Classez les requêtes par ordre de priorité envoyées aux services afin que celles avec une priorité plus élevée soient reçues et traitées plus rapidement que celles dont la priorité est plus basse. Ce patron est utile dans les applications qui proposent différentes garanties de niveau de service en fonction des clients.

Contexte et problème

Les applications peuvent déléguer des tâches spécifiques à d'autres services, par exemple exécuter un traitement de fond ou s'intégrer avec d'autres applications ou services. Dans le cloud, une file d'attente des messages permet généralement de déléguer des tâches au traitement en arrière-plan. La plupart du temps, l'ordre de réception des requêtes par un service n'a pas d'importance. Toutefois, dans certains cas, cependant, il est nécessaire d'accorder la priorité à certaines requêtes. Celles-ci doivent être traitées avant des requêtes moins prioritaires pourtant envoyées plus tôt par l'application.

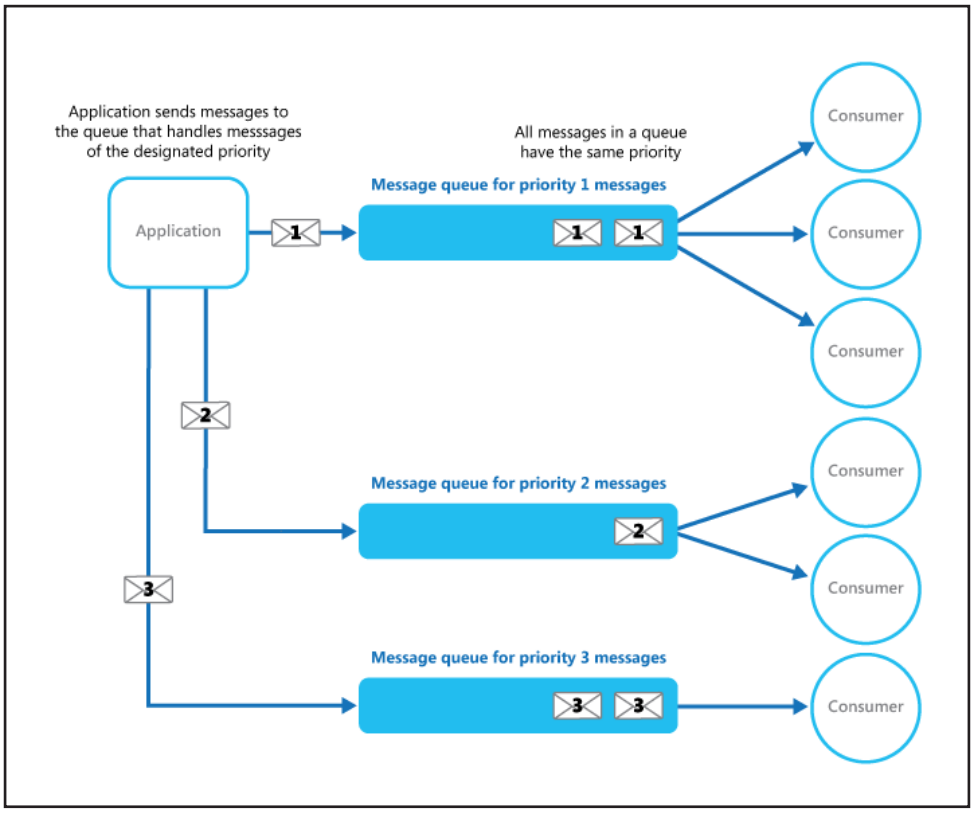
Solution

Une file d'attente est généralement une structure de type premier entré, premier sorti (FIFO), et les utilisateurs reçoivent généralement les messages dans l'ordre dans lequel ils ont été déposés dans la file d'attente. Cependant, certaines files d'attente prennent en charge la priorité de messagerie. L'application qui dépose un message peut affecter une priorité pour que les messages de la file d'attente soient automatiquement réorganisés, de telle sorte que les plus prioritaires soient reçus avant ceux de moindre priorité. La figure ci-dessous représente une file d'attente avec priorité de messagerie.



La plupart des implémentations de files d'attente des messages prennent en charge plusieurs consommateurs (selon le patron Competing Consumers [Consommateurs concurrents]), et le nombre de processus consommateurs peut augmenter ou réduire en fonction de la demande.

Dans les systèmes qui ne prennent pas en charge les files d'attente de messages avec priorité, vous pouvez adopter une autre solution, qui consiste à gérer une file d'attente distincte pour chaque priorité. L'application est chargée de déposer les messages dans la file d'attente appropriée. Chaque file d'attente peut être associée à un pool de consommateurs distinct. Les files d'attente plus prioritaires peuvent bénéficier d'un pool d'utilisateurs plus important sur du matériel plus rapide que celles à basse priorité. La figure suivante représente des files d'attente distinctes utilisées pour chaque priorité.



Une variante de cette stratégie consiste à gérer un seul pool de consommateurs qui recherche tout d'abord les messages dans les files d'attente haute priorité avant de commencer à extraire les messages des files d'attente moins prioritaires. Il existe des différences sémantiques entre une solution qui fait usage d'un pool unique de processus consommateurs (à partir d'une seule file d'attente qui prend en charge les messages de différentes priorités ou de plusieurs files d'attente qui gèrent chacune les messages d'une certaine priorité) et une solution qui utilise plusieurs files d'attente, dont chacune est associée à un pool distinct.

Avec l'approche du pool unique, les messages les plus prioritaires sont toujours reçus et traités avant les messages moins prioritaires. Théoriquement, les messages qui ont une très faible priorité peuvent être supplantés en permanence et risquent de ne jamais être traités. Dans l'approche avec plusieurs pools, les messages de priorité moindre sont toujours traités, mais moins rapidement que les prioritaires (en fonction de la taille relative des pools et des ressources dont ils disposent).

Un mécanisme de file d'attente prioritaire peut offrir les avantages suivants :

- Il permet aux applications répondre à des besoins de l'entreprise qui demandent une hiérarchisation de la disponibilité ou des performances, notamment proposer des niveaux de service différents à des groupes de clients spécifiques.

- Il peut aider à réduire les coûts d'exploitation. L'approche de la file d'attente unique vous permet de réduire le nombre d'utilisateurs, si nécessaire. Les messages hautement prioritaires sont toujours traités en premier (éventuellement plus lentement), et les messages de priorité moindre peuvent être retardés plus longtemps. Si vous avez implémenté l'approche impliquant plusieurs files d'attente de messages avec des pools d'utilisateurs distincts pour chacune d'entre elles, vous pouvez réduire le pool pour les moins prioritaires, voire suspendre le traitement de certaines files d'attente de très faible priorité en bloquant tous les utilisateurs qui sont à l'écoute des messages qui en proviennent.
- Cette approche avec plusieurs files d'attente de messages permet d'optimiser les performances et l'évolutivité de l'application en partitionnant les messages en fonction des besoins de traitement. Par exemple, les tâches essentielles peuvent être mises en priorité afin d'être traitées par des récepteurs exécutés immédiatement, alors que les tâches de fond moins importantes peuvent être gérées par des récepteurs dont l'exécution est planifiée pendant les périodes creuses.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Définissez les priorités dans le contexte de la solution. Par exemple, une priorité élevée peut indiquer que les messages doivent être traités dans les dix secondes. Identifiez les exigences de traitement des éléments hautement prioritaires et les autres ressources qui doivent être affectées pour répondre à ces critères.

Déterminez si tous les éléments hautement prioritaires doivent être traités avant les éléments de moindre priorité. Si les messages sont traités par un seul pool de consommateurs, vous devez fournir un mécanisme qui peut anticiper et suspendre une tâche qui gère un message de faible priorité si un message plus prioritaire fait son apparition.

Avec la méthode qui utilise plusieurs files d'attente, lorsque vous utilisez un seul pool de processus consommateurs à l'écoute de toutes les files d'attente plutôt qu'un pool dédié pour chacune d'entre elles, le consommateur doit appliquer un algorithme qui garantit qu'il traite toujours les messages provenant des files d'attente à haute priorité avant ceux des moins prioritaires.

Surveillez la vitesse de traitement des files d'attente à haute et basse priorité pour vérifier que les messages qu'elles contiennent sont traités à la vitesse attendue.

Si vous souhaitez que les messages à priorité faible soit absolument traités, vous devez nécessairement appliquer la méthode impliquant plusieurs files d'attente et plusieurs pools de consommateurs. Lorsqu'une file d'attente prend en charge la hiérarchisation des messages, vous avez également la possibilité d'augmenter dynamiquement la priorité d'un message en file d'attente à mesure qu'il prend de l'ancienneté. Cette approche dépend toutefois de la file d'attente de messages qui offre cette fonction.

Il est préférable d'utiliser une file d'attente distincte par priorité de message avec les systèmes qui ont un petit nombre de priorités clairement définies.

La priorité des messages peut être déterminée logiquement par le système. Par exemple, au lieu d'indiquer explicitement la priorité des messages, vous pouvez déterminer qu'ils proviennent de « clients avec accès payant » ou de « clients avec accès gratuit ». En fonction de votre modèle d'entreprise, votre système peut affecter plus de ressources au traitement des messages des clients payants qu'à ceux des clients gratuits.

La recherche d'un message dans une file d'attente peut entraîner des coûts financiers et de traitement (en effet, certains systèmes de messagerie commerciaux facturent une somme modique à chaque message envoyé ou récupéré et à chaque interrogation d'une file d'attente). Ce coût augmente avec le nombre de files d'attente interrogées.

Il est possible d'ajuster dynamiquement la taille d'un pool de consommateurs en fonction de la longueur de la file d'attente qu'il dessert. Pour plus d'informations, consultez les Conseils sur la mise à l'échelle automatique.

Quand utiliser ce patron

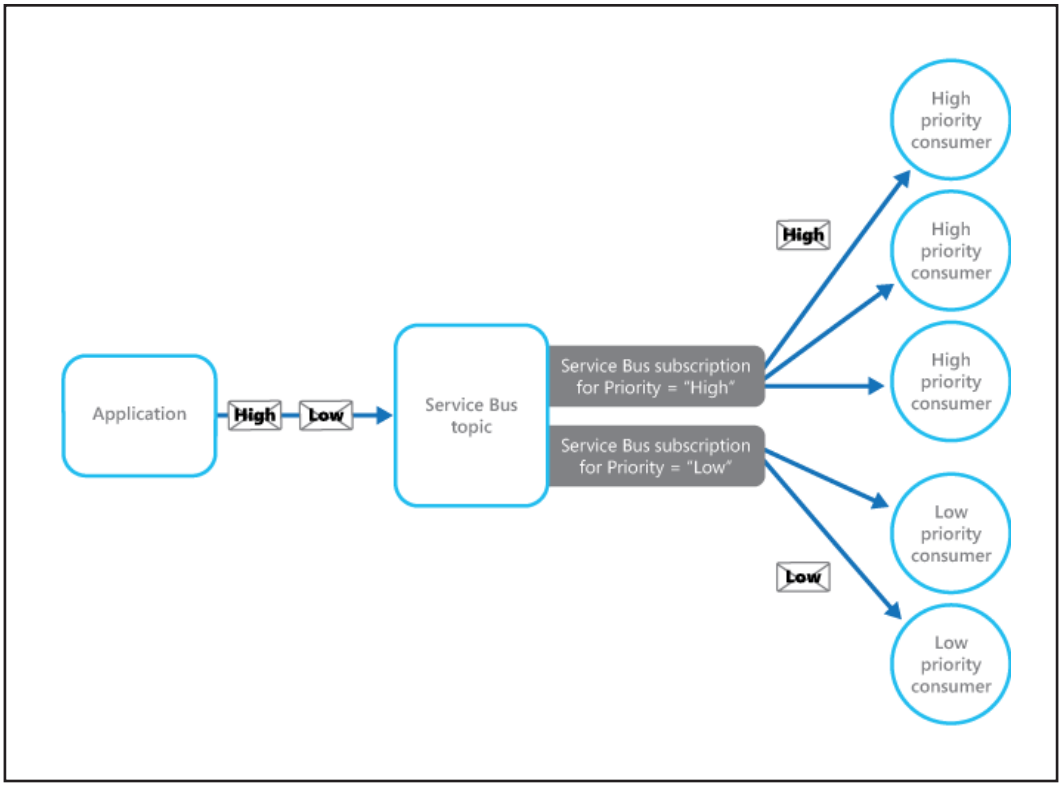
Ce patron s'avère utile sur les scénarios dans lesquels :

- Le système doit gérer plusieurs tâches dont les priorités diffèrent.
- Les priorités varient en fonction des utilisateurs ou des locataires.

Exemple

Microsoft Azure n'offre aucun mécanisme de file d'attente natif qui prenne en charge la hiérarchisation automatique des messages par le tri. En revanche, il offre des rubriques et des abonnements Azure Service Bus qui prennent en charge un mécanisme de file d'attente permettant de filtrer les messages, avec une large gamme de fonctionnalités flexibles qui le rendent idéal dans la plupart des implémentations de files d'attente prioritaires.

Une solution Azure peut implémenter une rubrique Service Bus dans laquelle une application peut publier des messages, de la même manière que dans une file d'attente. Les messages peuvent contenir des métadonnées sous la forme de propriétés personnalisées définies par l'application. Des abonnements Service Bus peuvent être associés à la rubrique et peuvent filtrer les messages en fonction de leurs propriétés. Lorsqu'une application envoie un message à une rubrique, celui-ci est redirigé vers l'abonnement approprié, où il peut être lu par un consommateur. Les processus consommateurs peuvent extraire les messages d'un abonnement à l'aide de la même sémantique qu'une file d'attente de messages (un abonnement est une file d'attente logique). La figure suivante illustre l'implémentation d'une file d'attente à priorités avec des rubriques et des abonnements Azure Service Bus.



Dans la figure ci-dessus, l'application crée plusieurs messages et attribue une propriété personnalisée nommée Priority (priorité) dans chaque message, avec une valeur High (haute) ou Low (basse). L'application publie ces messages dans une rubrique. La rubrique comporte deux abonnements associés qui filtrent tous deux les messages en examinant la propriété Priority. Un abonnement accepte les messages lorsque la propriété Priority a la valeur High, l'autre les accepte lorsque cette propriété est définie sur Low. Un pool de consommateurs lit les messages de chaque abonnement. L'abonnement prioritaire est associé à un pool de plus grande taille, et les consommateurs correspondants peuvent utiliser des ordinateurs plus puissants et disposer de plus de ressources que ceux du pool moins prioritaire.

Notez que la désignation de la priorité High et Low des messages n'a pas de signification particulière dans cet exemple. Il s'agit simplement d'étiquette servant de propriétés dans chaque message, utilisées pour rediriger les messages vers un abonnement spécifique. Si des priorités supplémentaires sont nécessaires, il est relativement facile de créer d'autres abonnements et pools de processus consommateurs pour les gérer.

La solution PriorityQueue disponible sur GitHub contient une implémentation de cette méthode. Elle contient projets de rôles de travail nommés PriorityQueue.High et PriorityQueue.Low. Ces rôles de travail héritent de la classe PriorityWorkerRole qui contient la fonction permettant de se connecter à un abonnement spécifié dans la méthode OnStart.
Les rôles de travail PriorityQueue.High et PriorityQueue.Low se connectent à différents abonnements, définis par leurs paramètres de configuration. Un administrateur peut configurer différents nombres de chaque rôle à exécuter. Il existera généralement plus d'instances du rôle de travail PriorityQueue. High que du rôle de travail PriorityQueue.Low.

La méthode Run de la classe PriorityWorkerRole organise l'exécution de la méthode ProcessMessage virtuelle (également définie dans la classe PriorityWorkerRole) pour chaque message reçu dans la file d'attente. Le code suivant illustre les méthodes Run et ProcessMessage. La classe QueueManager, définie dans le projet PriorityQueue.Shared, fournit des méthodes d'assistance à l'utilisation de files d'attente d'Azure Service Bus.

```
public class PriorityWorkerRole : RoleEntryPoint
{
  private QueueManager queueManager;
  ...

  public override void Run()
  {
    // Start listening for messages on the subscription.
    var subscriptionName = CloudConfigurationManager.GetSetting("SubscriptionName");
    this.queueManager.ReceiveMessages(subscriptionName, this.ProcessMessage);
    ...;
  }
  ...

  protected virtual async Task ProcessMessage(BrokeredMessage message)
  {
    // Simulating processing.
    await Task.Delay(TimeSpan.FromSeconds(2));
  }
}
```

Les rôles de travail PriorityQueue.High et PriorityQueue.Low se substituent tous deux à la fonctionnalité par défaut de la méthode ProcessMessage. Le code ci-dessous illustre la méthode ProcessMessage pour le rôle de travail PriorityQueue.High.

```
protected override async Task ProcessMessage(BrokeredMessage message)
{
  // Simulate message processing for High priority messages.
  await base.ProcessMessage(message);
  Trace.TraceInformation("High priority message processed by " +
    RoleEnvironment.CurrentRoleInstance.Id + " MessageId: " + message.MessageId);
}
```

Lorsqu'une application envoie des messages à la rubrique associée aux abonnements utilisés par les rôles de travail PriorityQueue.High et PriorityQueue.Low, elle spécifie la priorité à l'aide de la propriété personnalisée Priority, comme illustré dans l'exemple de code suivant. Ce code (implémenté dans la classe WorkerRole du projet PriorityQueue.Sender) utilise la méthode d'assistance SendBatchAsync de la classe QueueManager pour publier des messages par lots dans une rubrique.

```
// Send a low priority batch.
var lowMessages = new List<BrokeredMessage>();

for (int i = 0; i < 10; i++)
{
  var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
  message.Properties["Priority"] = Priority.Low;
  lowMessages.Add(message);
}

this.queueManager.SendBatchAsync(lowMessages).Wait();
...

// Send a high priority batch.
var highMessages = new List<BrokeredMessage>();

for (int i = 0; i < 10; i++)
{
  var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
  message.Properties["Priority"] = Priority.High;
  highMessages.Add(message);
}

this.queueManager.SendBatchAsync(highMessages).Wait();
```

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Un exemple illustrant ce patron est disponible sur GitHub.
- Introduction à la messagerie asynchrone. Un service consommateur qui traite une requête peut envoyer une réponse à l'instance de l'application qui a publié cette requête. Fournit des informations sur les stratégies que vous pouvez utiliser pour implémenter la messagerie de requête/réponse.
- Patron Competing Consumers (Consommateurs concurrents) Pour augmenter le débit des files d'attente, il est possible de mettre plusieurs consommateurs à l'écoute sur une même file et de traiter les tâches en parallèle. Ces consommateurs seront en concurrence pour récupérer les messages, mais un seul d'être eux doit être en mesure de traiter chaque message. Fournit des informations complémentaires sur les avantages et les compromis à concéder pour implémenter cette méthode.
- Patron Throttling (Limitation). Vous pouvez implémenter une limitation à l'aide de files d'attente. La messagerie prioritaire peut être utilisée pour garantir que les requêtes provenant d'applications stratégiques ou exécutées par des clients privilégiés aient la priorité sur les celles issues d'applications moins importantes.

- Conseils sur la mise à l'échelle automatique. Il peut être possible de mettre à l'échelle le pool de processus consommateurs qui gèrent une file d'attente en fonction de la longueur de celle-ci. Cette stratégie peut améliorer les performances, notamment pour les pools qui gèrent les messages hautement prioritaires.

- Patrons d'intégration d'entreprise avec Service Bus sur le blog d'Abhishek Lal.

Patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente)

Utilisez une file d'attente en tant que tampon entre une tâche et le service qu'elle appelle afin de lisser les charges volumineuses intermittentes qui peuvent entraîner l'échec du service ou l'expiration du délai de la tâche. Vous pouvez ainsi réduire l'impact des pics de demande sur la disponibilité et la réactivité de la tâche et du service.

Contexte et problème

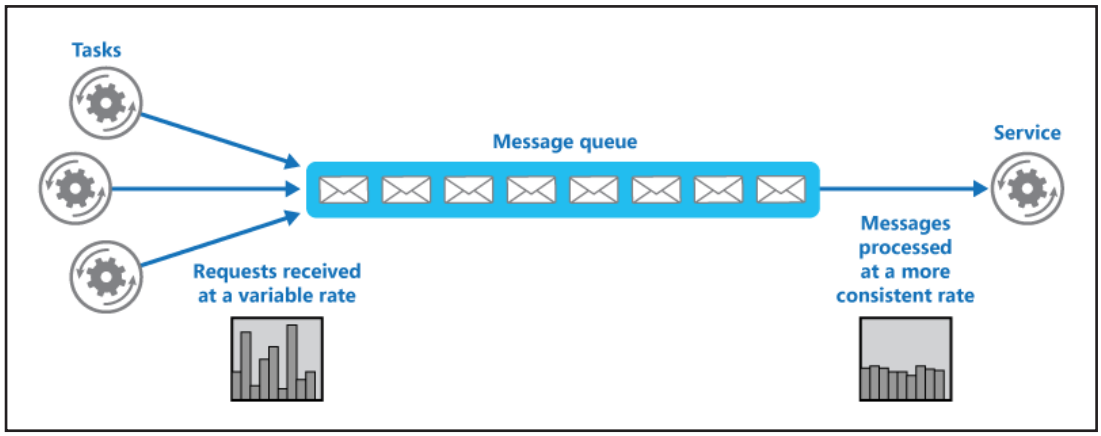
De nombreuses solutions dans le cloud exécutent des tâches qui appellent des services. Dans cet environnement, si un service est soumis à des charges volumineuses intermittentes, il peut être à l'origine de problèmes de performances ou de fiabilité.

Un service peut être intégré à la même solution que les tâches qui l'utilisent. Il peut également s'agir d'un service tiers offrant un accès à des ressources fréquemment utilisées, telles qu'un cache ou un service de stockage. Si le même service est utilisé par plusieurs tâches simultanées, il peut être difficile de prévoir le volume des demandes qu'il peut recevoir à tout moment.

Un service peut faire l'objet de pics de demande qui entraînent une surcharge et l'empêchent de répondre aux requêtes en temps opportun. L'encombrement d'un service par un nombre élevé de requêtes simultanées peut également entraîner sa défaillance s'il n'est pas capable de gérer les conflits provoqués par ces requêtes.

Solution

Refactorisez la solution et placez une file d'attente entre la tâche et le service. Le service et la tâche s'exécutent de façon asynchrone. La tâche publie un message contenant les données requises par le service dans une file d'attente. Cette dernière a fonction de tampon et stocke le message jusqu'à ce qu'il soit récupéré par le service. Le service extrait les messages de la file d'attente et les traite. Les requêtes de plusieurs tâches, dont la fréquence de génération peut être très variable, peuvent être transmises au service par la même file d'attente de messages. La figure ci-dessous illustre la manière dont une file d'attente permet de niveler la charge d'un service.



La file d'attente dissocie les tâches du service, lequel peut gérer les messages à son propre rythme quel que soit le volume de demandes issues de tâches simultanées. Par ailleurs, rien ne vient retarder une tâche si le service n'est pas disponible au moment où elle publie un message dans la file d'attente.

Ce patron offre les avantages suivants :

- Il permet d'optimiser la disponibilité parce que les retards qui surviennent au niveau des services n'ont pas d'impact direct et immédiat sur la demande, qui peut continuer à publier des messages dans la file d'attente même lorsque le service n'est pas disponible ou n'est pas en train de traiter des messages.
- Il offre une évolutivité optimale, puisque le nombre de files d'attente et le nombre de services peuvent être modulés en fonction de la demande.
- Il permet de contrôler les coûts, car il suffit que le nombre d'instances du service déployées réponde adéquatement à une charge moyenne plutôt qu'à la charge de pointe.

Certains services implémentent une limitation lorsque la demande atteint un seuil au-delà duquel une défaillance du système peut survenir. La limitation peut réduire les fonctionnalités disponibles. Vous pouvez implémenter un nivellement de la charge avec ces services pour éviter d'atteindre ce seuil.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- Il est nécessaire d'implémenter une logique d'application qui régit la fréquence à laquelle les services gèrent les messages pour éviter de surcharger la ressource cible. Évitez de transmettre les pics de demande à l'étape suivante du système. Testez le système sous charge pour garantir qu'il fournit le nivellement nécessaire et ajustez le nombre de files d'attente et le nombre d'instances de service qui gèrent les messages pour y parvenir.
- Les files d'attente de messages constituent un mécanisme de communication unidirectionnel. Si une tâche attend une réponse d'un service, il peut s'avérer nécessaire de mettre en place un mécanisme utilisable par le service pour envoyer une réponse. Pour plus d'informations, consultez l'introduction à la messagerie asynchrone.
- Soyez prudent si vous appliquez la mise à l'échelle automatique à des services qui sont à l'écoute de requêtes sur la file d'attente. En effet, vous risquez d'augmenter les conflits sur les ressources partagées par ces services et de rendre moins efficace le nivellement de la charge par la file d'attente.

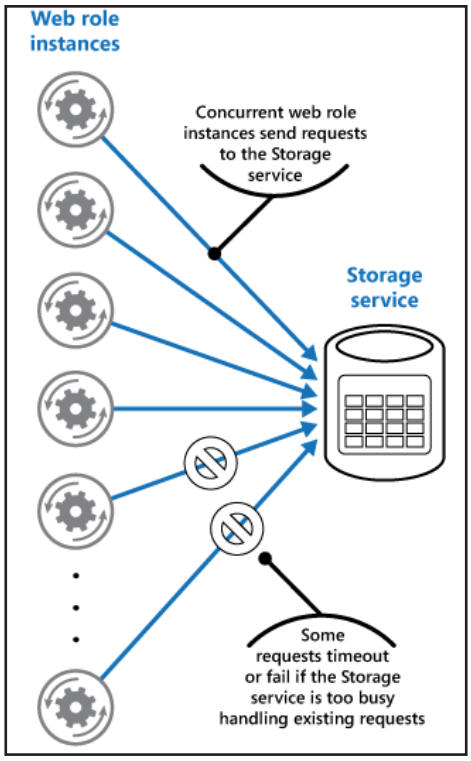
Quand utiliser ce patron

Ce patron est utile avec les applications qui utilisent des services soumis à des surcharges.

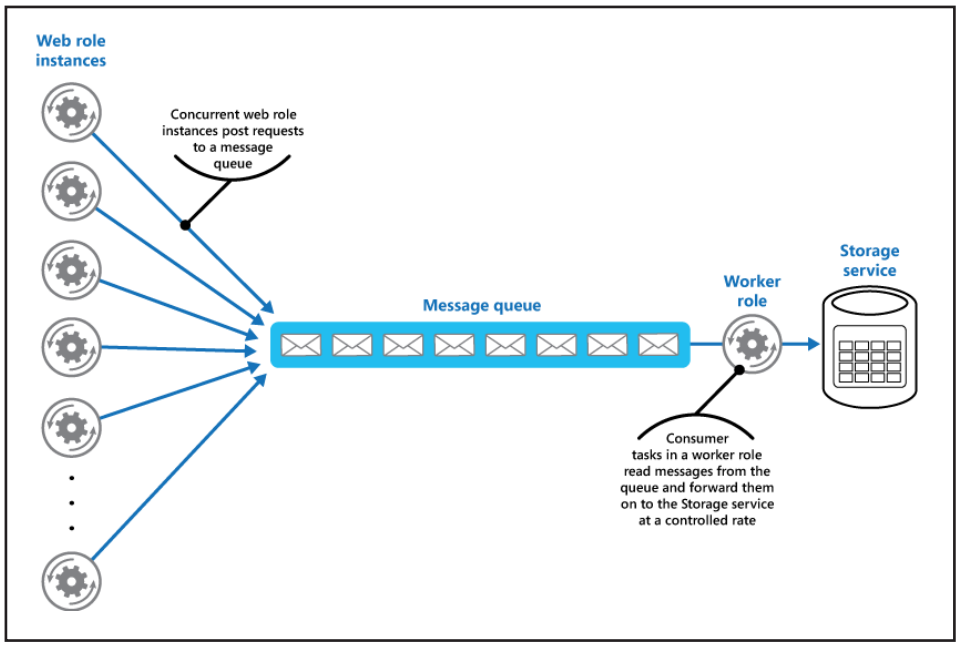
Il ne l'est pas si l'application attend une réponse du service avec un minimum de latence.

Exemple

Un rôle Web Microsoft Azure stocke les données à l'aide d'un service de stockage distinct. Si un grand nombre d'instances du rôle Web sont exécutées simultanément, il se peut que le service de stockage soit incapable de répondre suffisamment vite aux demandes pour empêcher le dépassement de délai ou l'échec de ces demandes. Cette figure illustre un service surchargé par un grand nombre de requêtes simultanées émises par les instances d'un rôle Web.



Pour résoudre ce problème, vous pouvez utiliser une file d'attente pour niveler la charge entre les instances du rôle Web et le service de stockage. Toutefois, le service de stockage est conçu pour accepter les demandes synchrones et il n'est pas facile de le modifier pour lire des messages et gérer le débit. Vous pouvez mettre en place un rôle de travail agissant comme un service proxy destiné à recevoir les requêtes de la file d'attente et à les transférer au service de stockage. La logique d'application du rôle de travail peut contrôler la fréquence à laquelle il transmet les demandes au service de stockage pour éviter à ce dernier d'être débordé. La figure ci-dessous illustre le nivellement de la charge entre le rôle Web et le service par une file d'attente et un rôle de travail.



Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Introduction à la messagerie asynchrone. Par nature, les files d'attente de messages sont asynchrones. Il peut s'avérer nécessaire de remanier la logique d'application d'une tâche si vous l'adaptez pour passer d'une communication directe avec les services à une utilisation avec file d'attente de messages. De même, il peut être nécessaire de refactoriser un service pour accepter les requêtes d'une file d'attente de messages. Il peut également être possible d'implémenter un service proxy tel que celui décrit dans l'exemple.
- Patron Competing Consumers (Consommateurs concurrents) Il peut être possible d'exécuter plusieurs instances d'un service, chacune d'entre elles jouant le rôle d'un consommateur de messages issus de la file d'attente de nivellement de la charge. Vous pouvez suivre cette méthode pour ajuster la fréquence de réception des messages ensuite transmis à un service.
- Patron Throttling (Limitation). Pour implémenter facilement une limitation sur un service, appliquez un nivellement de la charge basé sur une file d'attente et acheminez toutes les requêtes à un service au moyen d'une file d'attente de messages. Le service peut traiter les requêtes à une fréquence qui évite l'épuisement des ressources requises par le service et pour réduire les conflits éventuels.
- Concepts du service de file d'attente. Informations sur le choix d'un mécanisme de messagerie et gestion des files d'attente dans les applications Azure.

Patron Retry (Nouvelle tentative)

Autorisez une application à gérer les défaillances transitoires lorsqu'elle tente de se connecter à un service ou une ressource réseau, en réessayant de manière transparente une opération qui a échoué. Vous pouvez ainsi améliorer la stabilité de l'application.

Contexte et problème

Une application qui communique avec des éléments exécutés dans le cloud doit être sensible aux fautes transitoires qui peuvent survenir dans cet environnement. Les erreurs sont notamment la perte momentanée de la connectivité réseau aux composants et aux services, l'indisponibilité temporaire d'un service, ou les expirations de délai qui surviennent lorsqu'un service est occupé.

Ces erreurs se corrigent généralement d'elles-mêmes et, si l'action qui a déclenché l'erreur est répétée après un délai approprié, il est probable qu'elle réussisse. Par exemple, un service de base de données qui traite un grand nombre de requêtes simultanées peut implémenter une stratégie de limitation qui rejette temporairement toutes les requêtes ultérieures jusqu'à ce que sa charge se soit allégée. La connexion d'une application qui tente d'accéder à la base de données peut échouer, mais réussir ensuite si retentée au bout d'un certain temps.

Solution

Dans le cloud, les erreurs temporaires ne sont pas rares et une application doit être conçue pour les gérer de façon élégante et transparente. Ainsi, l'effet éventuel des erreurs sur les tâches effectuées par l'application est réduit.

Si une application détecte une défaillance lorsqu'elle tente d'envoyer une requête à un service distant, elle peut la gérer à l'aide des stratégies suivantes :

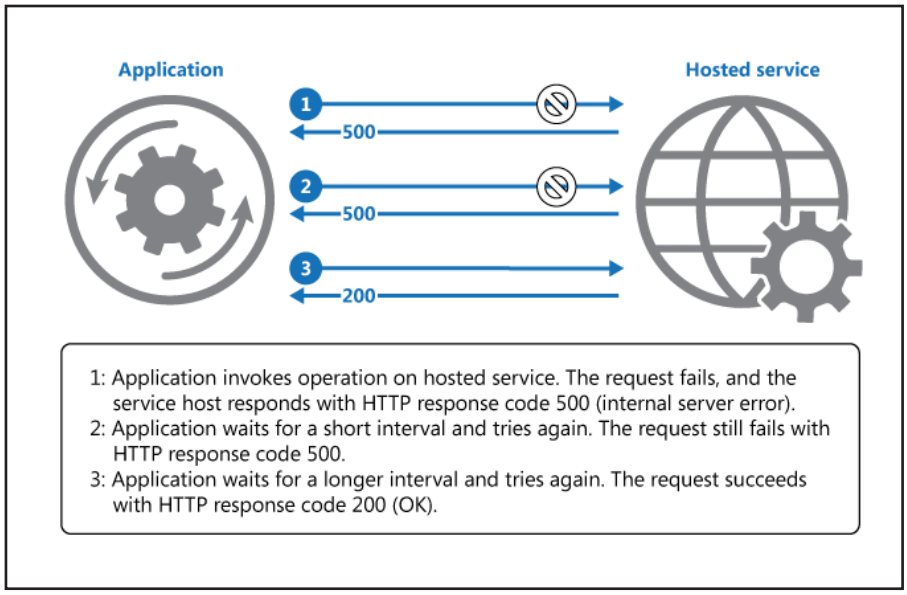
- **Annuler.** Si l'erreur indique que la panne n'est pas transitoire ou qu'elle est peu susceptible de réussir si répétée, l'application doit annuler l'opération et signaler une exception. Par exemple, un échec d'authentification causé par des informations d'identification incorrectes n'est pas susceptible de réussir, quel que soit le nombre de tentatives.

- **Nouvelle tentative.** Si l'erreur signalée est inhabituelle ou rare, il se peut qu'elle ait pour cause des circonstances inhabituelles, telles que la détérioration d'un paquet réseau au cours de sa transmission. Dans ce cas, l'application peut retenter immédiatement la requête ayant échoué car il est improbable que l'erreur se répète ; la requête réussira probablement.
- **Nouvelle tentative après délai.** Si l'échec est provoqué par l'une des défaillances de connectivité ou de disponibilité les plus courantes, le réseau ou le service peut avoir besoin d'une brève période d'attente, le temps que les problèmes de connectivité soient corrigés ou que le retard soit rattrapé. L'application doit alors laisser s'écouler un temps approprié avant de faire une nouvelle tentative.

Pour les défaillances transitoires les plus courantes, la période entre les tentatives doit répartir aussi régulièrement que possible les requêtes provenant des plusieurs instances de l'application. Ainsi, un service très actif risque moins de continuer à être surchargé. Si plusieurs instances d'une application demandent constamment à un service d'effectuer de nouvelles tentatives, celui-ci mettra plus de temps à récupérer.

Si la requête échoue encore, l'application peut marquer une pause et faire une autre tentative. Le cas échéant, les périodes d'attente entre les tentatives doivent être de plus en plus longues, jusqu'à ce que le nombre maximum de requêtes ait été atteint. L'augmentation du délai peut être incrémentielle ou exponentielle, en fonction du type de défaillance et de sa probabilité de résolution au cours de cette période.

Le diagramme ci-dessous illustre l'appel d'une opération au moyen de ce patron dans un service hébergé. Si la requête échoue après un nombre prédéfini de tentatives, l'application doit considérer l'erreur comme une exception et la traiter en conséquence.



L'application doit encapsuler toutes les tentatives d'accès à un service distant dans un code qui implémente une stratégie d'exécution des nouvelles tentatives correspondant à l'une des méthodes ci-dessus. Les requêtes envoyées à différents services peuvent être soumises à des stratégies différentes. Certains fournisseurs proposent des bibliothèques qui mettent en œuvre des stratégies de nouvelles tentatives selon lesquelles l'application peut spécifier le nombre maximum d'essais, l'intervalle entre les tentatives et d'autres paramètres encore.

Une application doit consigner les détails des erreurs et des opérations qui ont échoué. En effet, ces informations sont utiles pour les opérateurs. S'il arrive fréquemment qu'un service soit indisponible ou occupé, c'est souvent parce qu'il a épuisé ses ressources. Pour que ces erreurs soient moins fréquentes, vous pouvez monter en charge le service. Par exemple, si un service de base de données est constamment surchargé, il peut être avantageux de partitionner la base de données et de répartir la charge entre plusieurs serveurs.

Microsoft Entity Framework offre des fonctionnalités permettant de réessayer les opérations de base de données. Par ailleurs, la plupart des services Azur et des SDK clientes comportent un mécanisme permettant d'effectuer de nouvelles tentatives. Pour plus d'informations, consultez Conseils sur les nouvelles tentatives de services spécifiques.

Problèmes et considérations

Prenez en compte les points suivants lorsque vous choisissez le mode d'implémentation de ce patron.

Le paramétrage de la stratégie de nouvelles tentatives doit toujours correspondre aux exigences professionnelles de l'application et à la nature de la défaillance. Il est préférable que certaines opérations non critiques échouent rapidement plutôt que d'effectuer plusieurs tentatives qui risquent de nuire aux performances de l'application. Par exemple, dans une application web interactive qui accède à un service distant, il est préférable de connaître un échec après un petit nombre de tentatives séparées par un bref délai, et d'afficher un message adapté à l'utilisateur (par exemple, « Réessayez ultérieurement »). Pour une application de traitement par lots, il peut être plus approprié d'augmenter le nombre de tentatives en augmentant potentiellement le délai entre chacune d'entre elles.

En effet, si la stratégie est trop agressive, avec une pause trop réduite entre les tentatives et un grand nombre d'essais, vous risquez d'aggraver encore l'état d'un service arrivant au maximum de sa capacité. Cette stratégie risqué également de jouer sur la réactivité de l'application si elle persiste à répéter une opération vouée à l'échec.

Si une requête échoue toujours après un nombre élevé de tentatives, il est préférable de signaler tout simplement la panne, afin d'éviter que d'autres demandes continuent à solliciter la même ressource. À l'expiration du délai, l'application peut toujours tenter une ou deux requêtes supplémentaires pour voir si elles réussissent. Pour plus de détails sur cette stratégie, consultez l'article Patron Circuit Breaker (Disjoncteur).

Déterminez si l'opération est idempotente. Dans l'affirmative, vous pouvez effectuer une autre tentative en toute sécurité. Dans le cas contraire, les tentatives risquent de provoquer plusieurs exécutions de l'opération, avec des effets secondaires non souhaitables. Par exemple, un service peut recevoir la demande, la traiter correctement, mais ne pas parvenir à envoyer une réponse. À ce stade, la logique des tentatives risque de déterminer que la première requête n'a pas été reçue, et donc de la renvoyer.

Une requête auprès d'un service peut échouer pour diverses raisons et déclencher différentes exceptions, en fonction de la nature de la défaillance. Certaines exceptions signalent une défaillance rapide à résoudre ; d'autres indiquent que l'échec est plus durable. Il est utile que la stratégie s'aligne sur le type de l'exception pour déterminer le délai entre chaque tentative.

En l'occurrence, réfléchissez aux conséquences que peuvent avoir plusieurs tentatives d'une opération sur la cohérence globale d'une transaction. Affinez la stratégie relative aux opérations des transactions afin de mettre toutes les chances de votre côté et d'éviter ainsi d'avoir à annuler toutes les étapes de la transaction.

Veillez à tester intégralement le code des nouvelles tentatives sous diverses conditions d'échec. Assurez-vous qu'il n'a aucune répercussion sur les performances ou la fiabilité de l'application, qu'il ne surcharge pas les services et les ressources, ou encore qu'il ne génère pas de condition de concurrence ou de goulot d'étranglement.

Implémentez uniquement une logique de répétition des tentatives lorsque vous avez bien compris le contexte d'une opération défaillante. Par exemple, si une tâche qui contient une stratégie de tentatives appelle une autre tâche qui en contient également une, cette couche supplémentaire risque de retarder encore le traitement. Il est préférable de configurer l'interruption rapide de la tâche de niveau inférieur et de signaler le motif de l'échec à la tâche qui l'a appelée. Celle-ci peut alors s'appuyer sur sa propre stratégie pour gérer l'erreur.
Il est important de consigner tous les échecs de connectivité pouvant être à l'origine d'une nouvelle tentative, et ainsi d'identifier les problèmes sous-jacents au niveau de l'application, des services ou des ressources.

Examinez les erreurs les plus susceptibles de se produire sur un service ou une ressource, afin de déterminer si elles risquent d'être de longue durée, voire terminales. Si c'est le cas, il est préférable gérer l'erreur comme une exception. L'application peut signaler ou consigner l'exception, puis tenter de poursuivre son exécution en appelant un autre service (s'il en existe un) ou en fournissant des fonctionnalités dégradées. Pour plus d'informations sur la détection et la gestion des erreurs de longue durée, consultez l'article Patron Circuit Breaker (Disjoncteur).

Patron Scheduler Agent Supervisor (Planificateur-agent-superviseur)

Coordonnez un ensemble d'actions distribuées sous la forme d'une opération unique. En cas d'échec de l'une des actions, essayez de résoudre les défaillances de manière transparente ou annulez tout le travail effectué, de telle sorte que l'opération réussisse ou échoue dans sa globalité. Vous permettez ainsi à un système distribué de récupérer et de retenter des actions que des exceptions transitoires, des erreurs de longue durée et des échecs de processus ont fait échouer, et le rendez ainsi plus résilient.

Contexte et problème

Une application exécute des tâches qui comprennent plusieurs étapes, dont certaines peuvent appeler des services à distance ou accéder à des ressources distantes. Chaque étape peut être indépendante des autres, mais elles sont toutes orchestrées par la logique de l'application qui implémente la tâche.

Dans la mesure du possible, l'application doit garantir que la tâche s'exécute jusqu'à la fin et résoudre les échecs qui peuvent survenir lors des accès aux services ou ressources à distance. Les défaillances peuvent avoir de nombreuses origines. Il peut, par exemple, s'agir d'un arrêt du réseau, d'une interruption des communications, de l'absence de réaction ou de l'instabilité d'un service distant, ou encore d'une ressource distante temporairement inaccessible en raison d'une limitation des ressources. Dans de nombreux cas les échecs sont transitoires et peuvent être réglés à l'aide du patron Retry (Nouvelle tentative).

Si l'application détecte une erreur plus permanente et plus difficilement récupérable, elle doit pouvoir rétablir un état plus cohérent du système et garantir l'intégrité de toute l'opération.

Solution

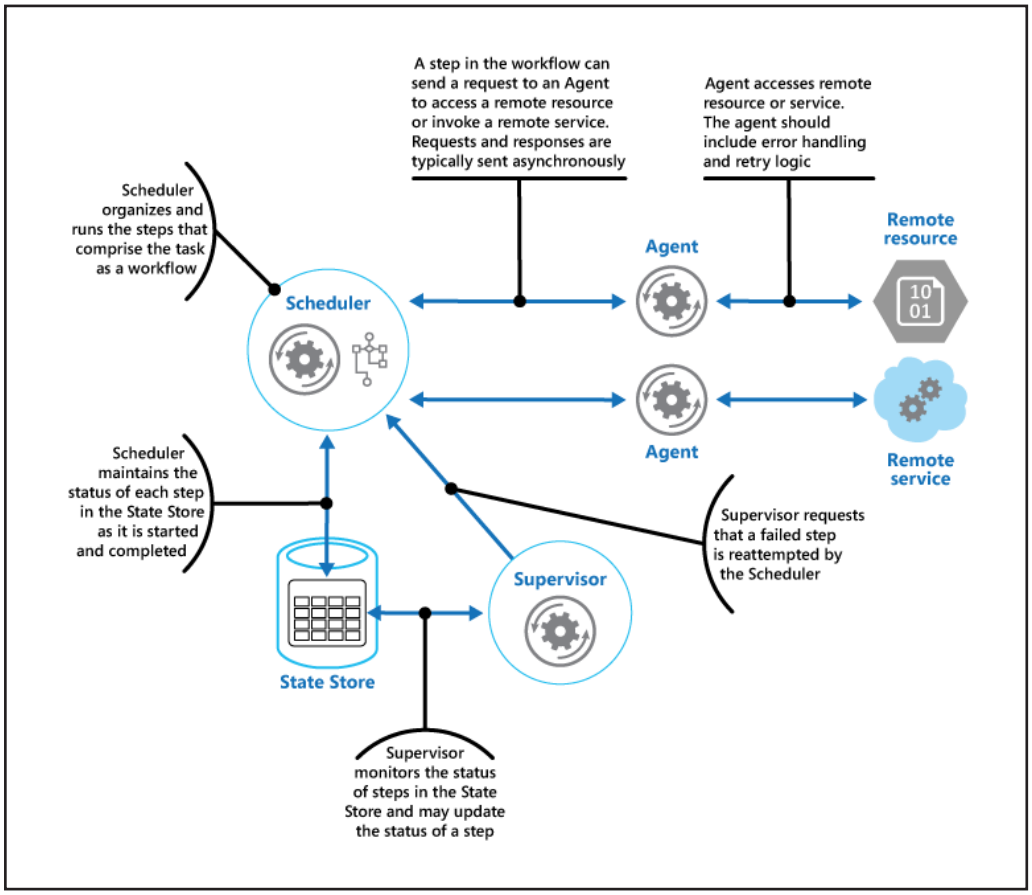
Le patron Scheduler Agent Supervisor (Planificateur-agent-superviseur) définit les acteurs suivants. Ces acteurs orchestrent les étapes à suivre dans le cadre de la globalité de la tâche.

- Le **planificateur** organise les étapes qui constituent la tâche à exécuter et gère leur fonctionnement. Ces étapes peuvent être combinées dans un pipeline ou un workflow. Le planificateur est chargé de s'assurer que les étapes décrites dans ce workflow sont effectuées dans le bon ordre. À mesure de l'exécution de chaque étape, il enregistre l'état du workflow, par exemple « étape non démarrée », « étape en cours d'exécution » ou « étape effectuée ». Les informations sur l'état doivent également contenir un délai d'achèvement, c'est-à-dire une limite de temps maximum d'exécution autorisé. Si une étape a besoin d'accéder à une ressource ou un service à distance, le planificateur appelle l'agent approprié et lui transmet les détails des travaux à effectuer. Le planificateur communique généralement avec un agent au moyen de messages asynchrones de type requête/réponse. Ceux-ci peuvent être implémentés par l'intermédiaire de files d'attente, mais d'autres technologies de messagerie distribuée peuvent également être utilisées.
- La fonction du planificateur est similaire à celle du gestionnaire de processus dans le patron de gestionnaire de processus. Le workflow en tant que tel est généralement défini et implémenté par un moteur de workflow contrôlé par le planificateur. Cette approche permet de dissocier la logique métier du workflow de celle du planificateur.

- L'**agent** contient une logique encapsulant un appel à un service distant ou l'accès à une ressource distante référencé par une étape dans une tâche. Généralement, chaque agent encapsule les appels à un seul service ou à une seule ressource et implémente la gestion des erreurs et la logique de nouvelle tentative appropriées (sujettes à un délai d'attente limité, voir plus bas). Si les étapes du workflow en cours d'exécution par le planificateur utilisent plusieurs services et ressources à différentes étapes, chaque étape peut faire référence à un autre agent (il s'agit d'un détail d'implémentation du patron).
- Le **responsable** surveille l'état des étapes de la tâche accomplie par le planificateur. Il s'exécute régulièrement (la fréquence dépend du système) et examine l'état des étapes effectuées par le planificateur. S'il détecte des étapes qui ont expiré ou échoué, il veille à ce que l'agent adéquat récupère l'étape ou effectue la mesure corrective nécessaire (cela peut impliquer la modification de l'état d'une étape). Notez que la récupération de la tâche ou les mesures correctives sont implémentées par le planificateur et les agents. Le superviseur doit simplement demander à ce que ces actions soient effectuées.

Le planificateur, l'agent et le superviseur sont des composants logiques, et leur implémentation physique dépend de la technologie utilisée. Par exemple, plusieurs agents logiques peuvent être implémentés dans le cadre d'un seul service web.

Le planificateur conserve des informations sur l'état d'avancement de la tâche et l'état de chaque étape dans un magasin de données durable, appelé le magasin d'état. Le superviseur peut utiliser ces informations pour déterminer si une étape a échoué. La figure illustre la relation entre le planificateur, les agents, le superviseur et le magasin d'état.



Ce diagramme montre une version simplifiée du patron. Dans une implémentation réelle, il pourrait y avoir plusieurs instances du planificateur en cours d'exécution en même temps, chacune étant un sous-ensemble de tâches. De même, le système peut exécuter plusieurs instances de chaque agent, voire plusieurs superviseurs. Dans ce cas, les superviseurs doivent veiller à coordonner soigneusement leur travail entre eux afin d'éviter toute concurrence pour récupérer les mêmes étapes et tâches ayant échoué. Le patron Leader Election (Élection du leader) offre une solution à ce problème.

Lorsque l'application est prête à exécuter une tâche, elle envoie une demande au planificateur. Le planificateur enregistre les informations d'état initial de la tâche et ses étapes (par exemple, étape pas encore entamée) dans le magasin d'état, puis commence à effectuer les opérations définies par le workflow. En entamant chaque étape, le planificateur met à jour les informations sur l'état de l'étape dans le magasin d'état (par exemple, étape en cours d'exécution).

Si une étape fait référence à une ressource distante ou à un service distant, le planificateur envoie un message à l'agent adéquat. Le message contient les informations dont l'agent a besoin pour accéder au service ou à la ressource, ainsi que le délai de clôture pour l'opération. Si l'agent termine son opération correctement, il envoie une réponse au planificateur. Le planificateur peut ensuite mettre à jour les informations d'état dans le magasin d'état (par exemple, étape terminée), puis effectuer l'étape suivante. Ce processus se poursuit jusqu'à ce que la tâche soit entièrement terminée.

Un agent peut implémenter toute logique de nouvelle tentative nécessaire pour effectuer son travail. Cependant, si l'agent ne termine pas son travail avant l'expiration du délai, le planificateur suppose que l'opération a échoué. Dans ce cas, l'agent doit interrompre son travail et ne faire aucune tentative d'envoi au planificateur (pas même un message d'erreur), ni aucune tentative de récupération d'aucune nature. Cette restriction est appliquée car après l'expiration ou l'échec d'une étape, il est parfois prévu qu'une autre instance de l'agent exécute l'étape qui a échoué (ce processus est décrit plus bas).

Si l'agent échoue, le planificateur ne recevra aucune réponse. Le patron ne distingue pas les étapes qui ont expiré de celles qui ont véritablement échoué.

Si une étape expire ou échoue, le magasin d'état contient un enregistrement qui indique que l'étape est en cours d'exécution, mais le délai de clôture sera dépassé. Le superviseur recherche les étapes de ce type et tente de les récupérer. Plusieurs stratégies sont envisageables pour le superviseur : il peut par exemple mettre à jour la valeur de délai de clôture pour prolonger le délai accordé pour terminer l'étape, puis envoyer un message au planificateur en précisant l'étape qui a expiré. Le planificateur peut alors tenter de répéter cette étape. Toutefois, cette conception nécessite que les tâches soient idempotentes.

Le superviseur devra peut-être empêcher la répétition de la même étape si elle continue d'expirer ou d'échouer. Pour ce faire, le superviseur peut conserver un nombre de tentatives pour chaque étape, ainsi que les informations d'état, dans le magasin d'état. Si ce nombre dépasse un seuil prédéfini, le superviseur peut adopter une stratégie d'attente pendant une période prolongée avant de notifier le planificateur qu'il doit retenter l'étape, dans l'espoir que le dysfonctionnement se résolve pendant cette période. Le superviseur peut aussi envoyer un message au planificateur pour lui demander d'annuler la tâche entière en implémentant un Patron Compensating Transaction (Transaction de compensation). Cette approche nécessite que le planificateur et les agents fournissent les informations nécessaires pour implémenter les opérations de compensation pour chaque étape terminée correctement.

Le superviseur n'est pas censé surveiller le planificateur et les agents ni les redémarrer s'ils échouent. Cet aspect du système doit être géré par l'infrastructure dans laquelle ces composants s'exécutent. De même, le superviseur ne doit pas connaître les opérations commerciales exactes exécutées par les tâches que le planificateur effectue (y compris la stratégie de compensation en cas d'échec de ces tâches). C'est là le but de la logique de workflow implémentée par le planificateur. La seule responsabilité du superviseur est de déterminer si une étape a échoué et de veiller à ce que cette étape soit répétée ou à ce que la tâche entière qui la contient soit annulée.

Si le planificateur est redémarré après un échec, ou si le workflow en cours d'exécution par le planificateur s'interrompt inopinément, le planificateur doit être en mesure de déterminer l'état de toute tâche en cours qu'il gérait avant qu'elle n'échoue, et être prêt à reprendre cette tâche à partir de ce moment. Les détails d'implémentation de ce processus sont susceptibles d'être spécifiques au système. Si la tâche ne peut pas être récupérée, il peut être nécessaire d'annuler le travail déjà effectué par la tâche. Cela pourrait aussi nécessiter l'implémentation d'une transaction de compensation.

Le principal avantage de ce patron est la résistance du système en cas d'échec irrécupérable ou temporaire imprévu. Le système peut être conçu pour s'auto-régénérer. Par exemple, si un agent ou le planificateur échoue, un nouvel agent ou planificateur peut être démarré, et le superviseur peut veiller à ce qu'une tâche soit reprise. Si le superviseur échoue, une autre instance peut être démarrée et peut prendre le relais à partir du moment où l'erreur s'est produite. S'il est prévu que le superviseur s'exécute périodiquement, une nouvelle instance peut être démarrée automatiquement après un intervalle prédéfini. Le magasin d'état peut être répliqué pour atteindre un degré de résilience encore plus élevé.

Problèmes et considérations

Prenez en compte les points suivants lorsque vous choisissez le mode d'implémentation de ce patron :

- Ce patron peut être difficile à implémenter et nécessite des tests approfondis de chaque mode d'échec potentiel du système.
- La logique de récupération/de nouvelle tentative implémentée par le planificateur est complexe et dépend des informations d'état contenues dans le magasin d'état. Il peut aussi être nécessaire d'enregistrer les informations requises pour implémenter une transaction de compensation dans un magasin de données durable.
- La fréquence d'exécution du superviseur a son importance. Il doit s'exécuter assez souvent pour empêcher toute étape ayant échoué de bloquer une application pendant une période prolongée, mais une fréquence d'exécution trop élevée engendrera une surcharge.
- Les étapes effectuées par un agent peuvent être exécutées plus d'une fois. La logique qui implémente ces étapes doit être idempotente.

Quand utiliser ce patron

Utilisez ce patron lorsqu'un processus qui s'exécute dans un environnement distribué, comme le cloud, doit résister aux échecs de communication et/ou aux échecs opérationnels.

Ce patron ne conviendra peut-être pas aux tâches qui n'invoquent pas de services distants ou n'accèdent pas à des ressources distantes.

Exemple

Une application web qui implémente un système de commerce électronique a été déployée sur Microsoft Azure. Les utilisateurs peuvent exécuter cette application pour parcourir les produits disponibles et passer des commandes. L'interface utilisateur s'exécute comme un rôle web, et les éléments de traitement de commande de l'application sont implémentés comme un ensemble de rôles de travail. Une partie de la logique de traitement des commandes implique l'accès à un service distant, et cet aspect du système peut être sujet à des pannes transitoires ou de plus longue durée. C'est pourquoi les concepteurs ont utilisé le patron Scheduler Agent Supervisor (Planificateur-agent-superviseur) pour implémenter les éléments de traitement de commande du système.

Lorsqu'un client passe une commande, l'application génère un message qui décrit la commande et publie ce message dans une file d'attente. Un processus d'envoi distinct s'exécutant dans un rôle de travail récupère le message, insère les détails de la commande dans la base de données des commandes et crée un enregistrement pour le processus de commande dans le magasin d'état. Notez que les insertions dans la base de données des commandes et le magasin d'état sont effectuées dans le cadre de la même opération. Le processus d'envoi est conçu pour faire en sorte que ces deux insertions soient complémentaires.

Les informations d'état créées par le processus d'envoi pour la commande incluent :

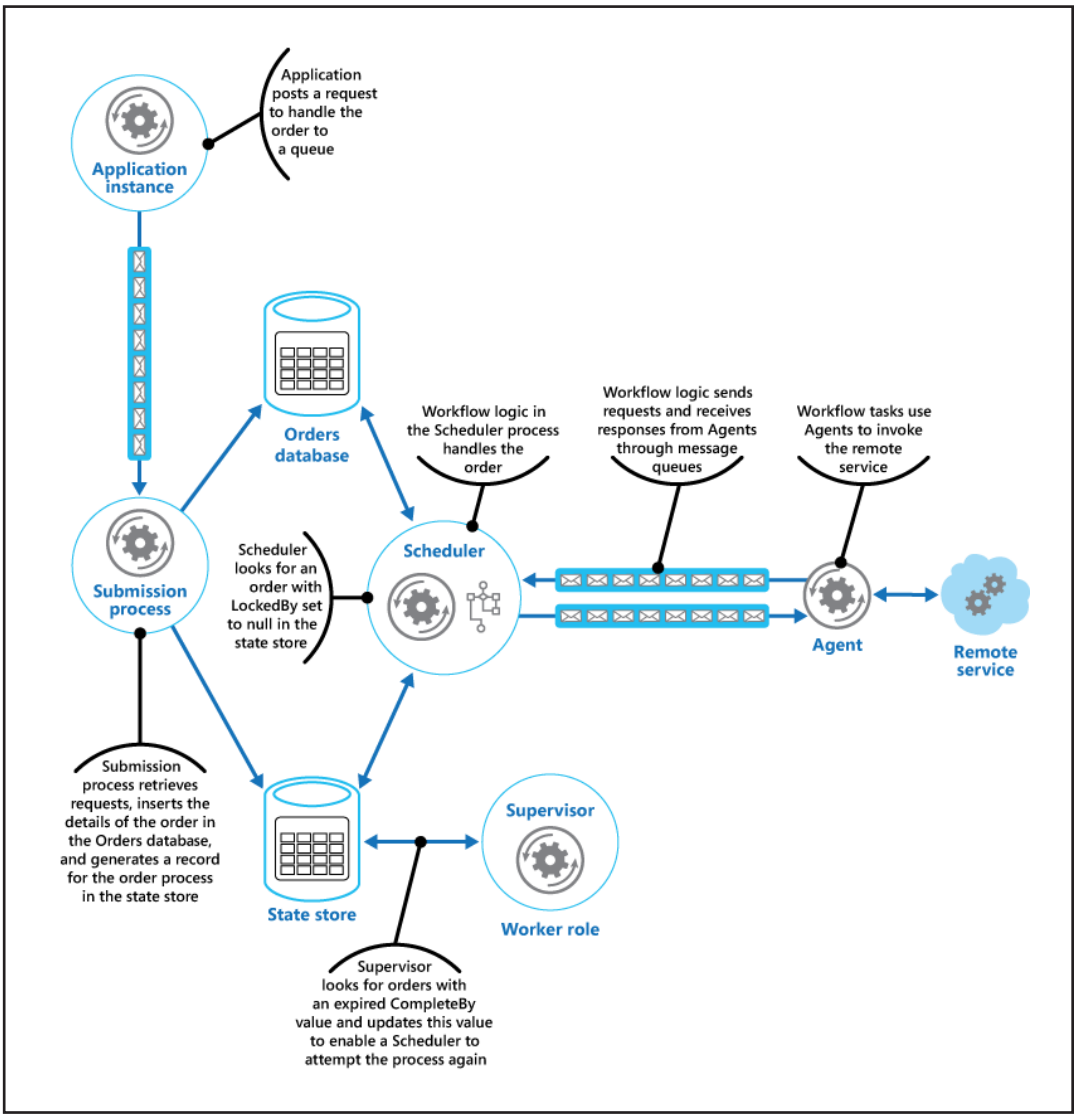
- **OrderID.** L'identifiant de la commande dans la base de données des commandes.
- **LockedBy.** L'identifiant de l'instance du rôle de travail qui gère la commande. Plusieurs instances actuelles du rôle de travail peuvent exécuter le planificateur, mais chaque commande doit être traitée par une seule instance.
- **CompleteBy.** L'échéance pour le traitement de la commande.
- **ProcessState.** L'état actuel de la tâche de traitement de la commande. Les états possibles sont les suivants :
    - **En attente.** La commande a été créée, mais n'est pas encore en cours de traitement.
    - **En cours de traitement.** La commande est en cours de traitement.
    - **Traitée.** La commande a bien été traitée.
    - **Erreur.** Le traitement de la commande a échoué.
- **FailureCount.** Le nombre de tentatives de traitement de la commande.

Dans ces informations d'état, le champ OrderID est copié à partir de l'identifiant de commande de la nouvelle commande. Les champs LockedBy et CompleteBy sont vides, le champ ProcessState est défini sur En attente et le champ FailureCount est défini sur 0.

Dans cet exemple, la logique de traitement de la commande est relativement simple et ne comporte qu'une seule étape qui invoque un service distant. Dans un scénario plus complexe comportant plusieurs étapes, le processus d'envoi impliquerait probablement plusieurs étapes, et plusieurs enregistrements seraient donc créés dans le magasin d'état (chacun d'entre eux décrivant l'état d'une étape individuelle).

Le planificateur s'exécute aussi dans le cadre d'un rôle de travail et implémente la logique commerciale qui traite la commande. Une instance du planificateur qui détecte les nouvelles commandes examine le magasin d'état pour trouver des enregistrements dans lesquels le champ LockedBy est vide et le champ ProcessState est défini sur En attente. Lorsque le planificateur trouve une nouvelle commande, il remplit immédiatement le champ LockedBy avec son propre identifiant d'instance, définit le champ CompleteBy sur une échéance appropriée et définit le champ ProcessState sur En cours de traitement. Le code est conçu pour être exclusif et atomique afin de veiller à ce que deux instances simultanées du planificateur ne puissent pas essayer de traiter la même commande en même temps.

Le planificateur exécute ensuite le workflow commercial pour traiter la commande de façon asynchrone, en transmettant la valeur dans le champ OrderID du magasin d'état. Le workflow qui traite la commande récupère les détails de celle-ci à partir de la base de données des commandes et effectue son travail. Lorsqu'une étape du workflow de traitement de commande doit invoquer le service à distance, elle utilise un agent. L'étape de workflow communique avec l'agent à l'aide d'une paire de files d'attente de messages Azure Service Bus qui agissent comme un canal de demande/réponse. La figure donne une vue d'ensemble de la solution.



Le message envoyé à l'agent par une étape de workflow décrit la commande et inclut le délai de clôture. Si l'agent reçoit une réponse du service distant avant l'expiration du délai de clôture, il publie un message de réponse dans la file d'attente Service Bus sur laquelle le workflow est à l'écoute. Lorsque l'étape de workflow reçoit le message de réponse valable, elle termine le traitement, et le planificateur définit le champ ProcessState de l'état de commande sur Traitée. À ce stade, le traitement de la commande s'est terminé correctement.

Si le délai de clôture expire avant que l'agent ne reçoive une réponse du service distant, l'agent interrompt simplement son processus et met fin au traitement de la commande. De même, si le workflow qui traite la commande dépasse le délai de clôture, il prend également fin. Dans les deux cas, l'état de la commande dans le magasin d'état reste défini sur En cours de traitement, mais le délai de clôture indique que le délai de traitement de la commande est dépassé, et le processus est considéré comme ayant échoué. Notez que si l'agent qui accède au service distant ou le workflow qui traite la commande (ou les deux) se terminent de façon inattendue, les informations enregistrées dans le magasin d'état resteront aussi définies sur En cours de traitement, et auront en fin de compte une valeur de délai expiré.

Si l'agent détecte une erreur irrécupérable et non transitoire alors qu'il tente de contacter le service distant, il peut renvoyer une réponse d'erreur au workflow. Le planificateur peut définir l'état de la commande sur Erreur et déclencher un événement qui alerte un opérateur. L'opérateur peut alors tenter de résoudre le motif de l'échec manuellement et recommencer l'étape de traitement qui a échoué.

Le superviseur examine périodiquement le magasin d'état et recherche des commandes dont le délai de clôture est expiré. Si le superviseur trouve un enregistrement, il adapte le champ FailureCount par incréments. Si le nombre d'échecs est inférieur à un seuil spécifié, le superviseur rétablit la valeur nulle du champ LockedBy, met à jour le champ CompleteBy avec un nouveau délai d'expiration et définit le champ ProcessState sur En attente. Une instance du planificateur peut récupérer cette commande et procéder à son traitement comme avant. Si le nombre d'échecs dépasse un seuil spécifié, le motif de l'échec est supposé être non transitoire. Le superviseur définit l'état de la commande sur Erreur et déclenche un événement qui alerte un opérateur.

Dans cet exemple, le superviseur est implémenté dans un rôle de travail distinct. Vous pouvez utiliser plusieurs stratégies différentes pour veiller à ce que la tâche superviseur soit exécutée. Vous pouvez par exemple utiliser le service Azure Scheduler (ne pas confondre avec le composant Planificateur de ce patron). Pour plus d'informations sur le service Azure Scheduler, consultez la page consacrée au planificateur.

Même si ce n'est pas illustré dans cet exemple, le planificateur peut avoir besoin d'informer l'application qui a envoyé la commande au sujet de la progression et de l'état de la commande. L'application et le planificateur sont isolés l'un de l'autre pour éviter toute dépendance entre eux. L'application ne connaît pas l'instance du planificateur qui traite la commande, et le planificateur ignore quelle instance de l'application a publié la commande.

Pour permettre l'envoi de l'état de la commande, l'application peut utiliser sa propre file d'attente de réponse privée. Les détails sur cette file d'attente de réponse seraient inclus dans la demande envoyée au processus d'envoi, ce qui signifie que ces informations seraient incluses dans le magasin d'état. Le planificateur publierait alors des messages dans cette file d'attente pour indiquer l'état de la commande (demande reçue, commande terminée, échec de la commande, etc.). Le planificateur doit inclure l'identifiant de commande dans ces messages, afin qu'ils puissent être associés à la demande d'origine de l'application.

Patrons et informations connexes

Les informations et les patrons suivants peuvent également être pertinents lors de l'implémentation de ce patron :

- Patron Retry (Nouvelle tentative). Un agent peut utiliser ce patron pour retenter en toute transparence une opération qui accède à un service distant ou à une ressource distante qui a échoué auparavant. À utiliser lorsque la cause de l'échec est probablement transitoire et peut être corrigée.
- Patron Circuit Breaker (Disjoncteur). Un agent peut utiliser ce patron pour traiter les pannes qui nécessitent un temps d'attente variable pour se corriger lors de la connexion à un service distant ou à une ressource distante.
- Patron Compensating Transaction (Transaction de compensation). Si le workflow exécuté par un planificateur ne peut pas se terminer correctement, il peut être nécessaire d'annuler tout travail déjà effectué. Le Patron Compensating Transaction (Transaction de compensation) décrit comment faire pour les opérations qui suivent le modèle final de cohérence. Ces types d'opérations sont généralement implémentées par un planificateur qui exécute des workflows et des processus commerciaux complexes.
- Introduction à la messagerie asynchrone. Les composants du patron Scheduler Agent Supervisor (Planificateur-agent-superviseur) s'exécutent généralement en étant découplés les uns des autres et communiquent de façon asynchrone. Décrit certaines approches qui permettent d'implémenter une communication asynchrone basée sur les files d'attente de messages.

- Patron Leader Election (Élection du leader). Il peut être nécessaire de coordonner les actions de plusieurs instances d'un superviseur pour les empêcher de tenter de récupérer le même processus ayant échoué. Le patron Leader Election (Élection du leader) décrit comment procéder.
- Architecture cloud : le patron planificateur/agent/superviseur sur le blog de Clemens Vasters
- Patron de gestionnaire de processus
- Référence 6 : Une saga sur les sagas. Un exemple qui illustre comment le patron CQRS utilise un gestionnaire de processus (dans le cadre des conseils sur la transition vers le patron CQRS).
- Microsoft Azure Scheduler

Patron Sharding (Partitionnement)

Divisez un magasin de données en un ensemble de partitions horizontales ou de partitions. Cela peut améliorer l'évolutivité lors du stockage et de l'accès à d'importants volumes de données.

Contexte et problème

Un magasin de données hébergé par un serveur unique peut être soumis aux restrictions suivantes :

- **Espace de stockage.** Un magasin de données pour une application cloud à grande échelle contient un très grand volume de données qui est susceptible d'augmenter considérablement au fil du temps. Un serveur fournit généralement un espace limité de stockage sur disque, mais vous pouvez remplacer les disques existants par des disques offrant plus d'espace ou ajouter des disques à une machine à mesure que les volumes de données s'accroissent. Toutefois, le système finira par atteindre une limite, et il ne sera plus possible d'augmenter facilement la capacité de stockage sur un serveur donné.
- **Ressources informatiques.** Une application cloud doit pouvoir prendre en charge un grand nombre d'utilisateurs simultanément. Chacun de ces utilisateurs exécute des requêtes qui récupèrent des informations à partir du magasin de données. Un serveur unique hébergeant le magasin de données ne sera peut-être pas en mesure de fournir la puissance de calcul nécessaire à une telle charge, ce qui engendrera des délais de réponse plus longs pour les utilisateurs et des échecs fréquents, puisque les applications qui tentent de stocker et de récupérer les données vont expirer. Il est peut-être possible d'ajouter de la mémoire ou de mettre à niveau les processeurs, mais le système atteindra sa limite lorsqu'il ne sera plus possible d'accroître les ressources de calcul.
- **Bande passante du réseau.** En fin de compte, les performances d'un magasin de données qui s'exécute sur un serveur unique dépendent de la vitesse à laquelle le serveur est capable de recevoir des demandes et d'envoyer des réponses. Il est possible que le volume de trafic réseau dépasse la capacité du réseau utilisé pour se connecter au serveur, ce qui entraîne l'échec des demandes.
- **Zone géographique.** Il peut être nécessaire de stocker les données générées par des utilisateurs spécifiques dans la même région que ces utilisateurs pour des raisons juridiques, de conformité ou de performances, ou encore pour réduire la latence d'accès aux donnés. Si les utilisateurs sont répartis dans plusieurs pays ou régions, il ne sera peut-être pas possible de stocker l'ensemble des données de l'application dans un seul magasin de données.

Une mise à l'échelle verticale, réalisée en augmentant la capacité de disque, la puissance de traitement, la mémoire et les connexions réseau, peut retarder les effets de certaines de ces contraintes, mais cela ne constitue probablement qu'une solution temporaire. Une application cloud commerciale capable de prendre en charge un grand nombre d'utilisateurs et de larges volumes de données doit pouvoir évoluer presque indéfiniment ; la mise à l'échelle verticale n'est donc pas nécessairement la meilleure solution.

Solution

Diviser le magasin de données en partitions horizontales. Chaque partition possède le même schéma, mais détient son propre sous-ensemble distinct de données. Une partition est un magasin de données à part entière (elle peut contenir les données provenant de nombreuses entités de différents types) qui s'exécute sur un serveur agissant comme un nœud de stockage.

Ce patron présente les avantages suivants :

- Vous pouvez faire évoluer le système en ajoutant davantage de partitions qui s'exécutent sur des nœuds de stockage supplémentaires.
- Un système peut utiliser du matériel trouvé dans le commerce plutôt que des ordinateurs spécialisés coûteux pour chaque nœud de stockage.
- Vous pouvez réduire la contention et améliorer les performances en équilibrant le scénario d'usage sur différentes partitions.
- Dans le cloud, les partitions peuvent se trouver à proximité des utilisateurs qui accèderont aux données.

Lorsque vous divisez un magasin de données en partitions, déterminez quelles données doivent être placées dans chaque partition. Une partition contient généralement des éléments appartenant à un type spécifique, déterminé par un ou plusieurs attributs des données. Ces attributs forment la clé de partition. La clé de partition doit être statique. Elle ne doit pas être basée sur des données susceptibles de changer.

Le partitionnement organise les données sur le plan physique. Lorsqu'une application stocke et récupère des données, la logique de partitionnement redirige l'application vers la partition adéquate. Cette logique de partitionnement peut être implémentée dans le cadre du code d'accès aux données de l'application ou via le système de stockage de données, si celui-ci prend en charge le partitionnement en toute transparence.

L'abstraction de l'emplacement physique des données dans la logique de partitionnements permet d'obtenir un niveau de contrôle très élevé des données contenues dans chaque partition. Cela permet également de faire migrer les données d'une partition à l'autre sans devoir remanier la logique commerciale d'une application, si les données des partitions doivent être redistribuées par la suite (par exemple, si les partitions se déséquilibrent). L'inconvénient de cette méthode est la charge supplémentaire en termes d'accès aux données requise pour déterminer l'emplacement de chaque élément de données lorsqu'il est récupéré.

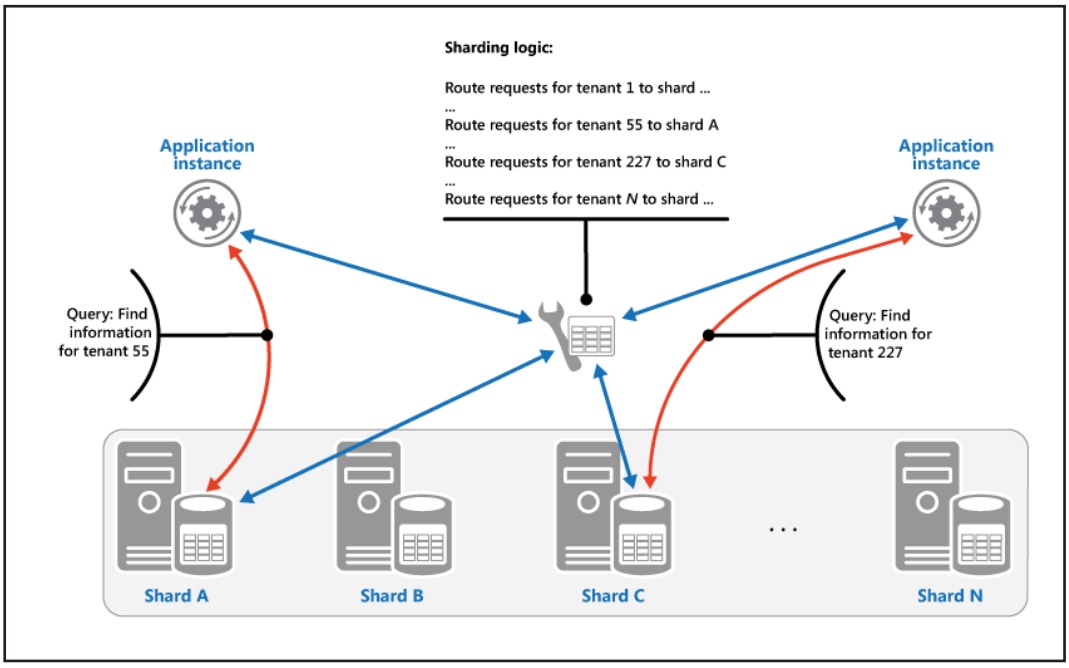
Pour garantir une évolutivité et des performances optimales, il est important de séparer les données d'une façon adaptée aux types de requêtes effectuées par l'application. Bien souvent, le système de partitionnement ne correspondra probablement pas aux exigences de chaque requête. Par exemple, dans un système à plusieurs locataires, une application peut avoir besoin de récupérer les données des locataires à l'aide de l'identifiant du locataire, mais elle pourrait aussi être amenée à effectuer des recherches dans ces données en fonction d'autres attributs, comme le nom ou l'emplacement du locataire. Pour gérer ces situations, implémentez une stratégie de partitionnement avec une clé de partition qui prend en charge les requêtes les plus fréquentes.

Si les requêtes récupèrent régulièrement des données à l'aide d'une combinaison de valeurs d'attribut, vous pouvez probablement définir une clé de partition composite en associant plusieurs attributs. Vous pouvez aussi utiliser un patron comme la table d'index pour permettre une recherche rapide dans les données sur la base des attributs qui ne sont pas couverts par la clé de partition.

Stratégies de partitionnement

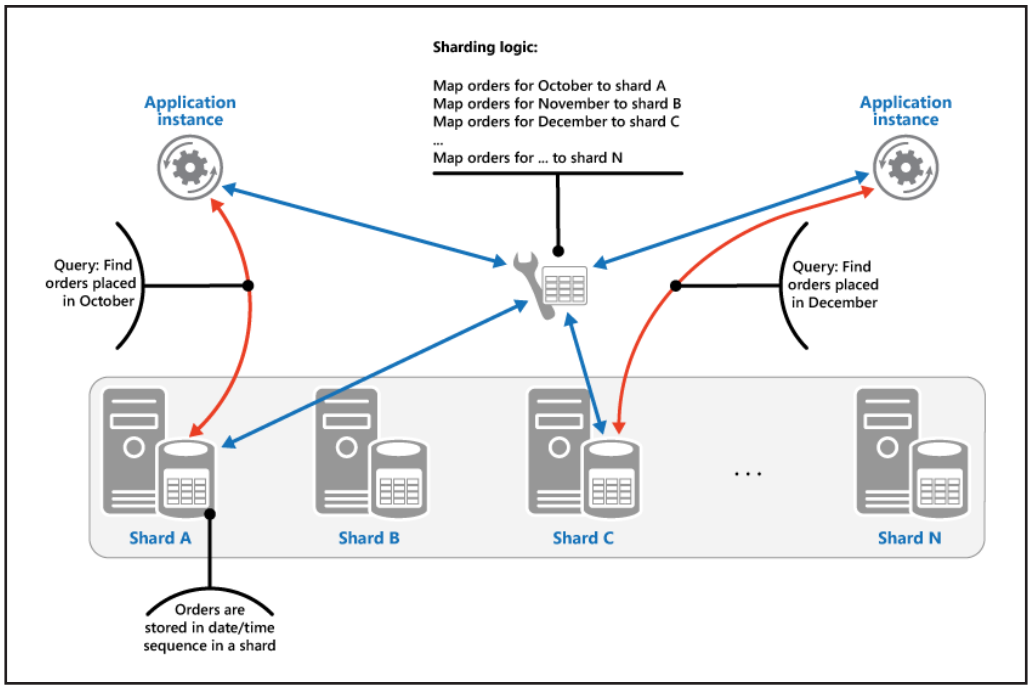
Trois stratégies sont fréquemment utilisées pour sélectionner la clé de partition et définir la distribution des données entre les différentes partitions. Notez que chaque serveur ne doit pas nécessairement correspondre à une seule partition : un seul serveur peut héberger plusieurs partitions. Les stratégies possibles sont les suivantes :

La stratégie de recherche. Dans cette stratégie, la logique de partitionnement implémente un mappage qui achemine une requête de données vers la partition contenant ces données à l'aide de la clé de partition. Dans une application à plusieurs locataires, toutes les données d'un locataire peuvent être stockées ensemble dans une partition si l'identifiant du locataire est utilisé comme clé de partition. Plusieurs locataires peuvent partager la même partition, mais les données d'un seul locataire ne seront pas réparties sur plusieurs partitions. La figure illustre le processus de partitionnement des données de locataires sur la base des identifiants de locataires.



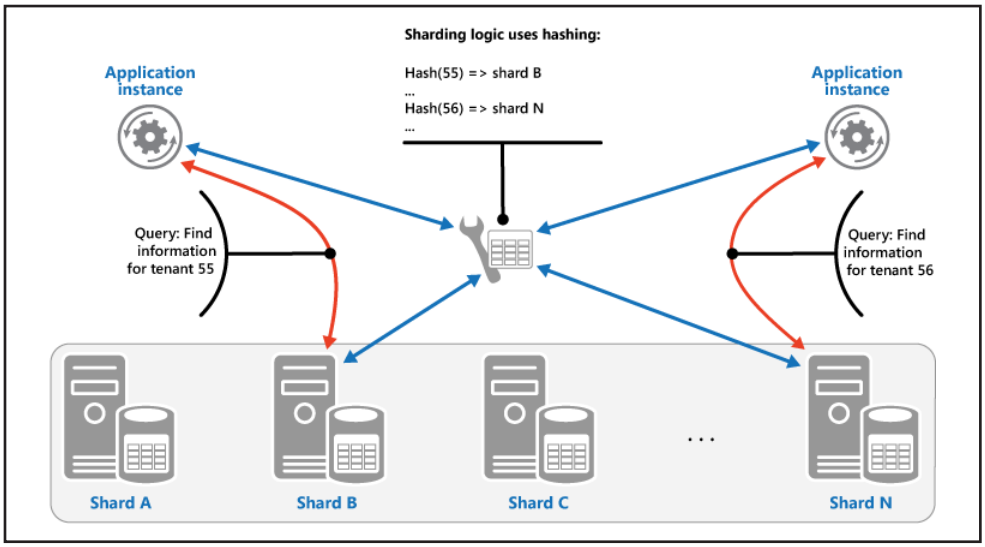
Le mappage entre la clé de partition et le stockage physique peut se baser sur des partitions physiques, chaque clé de partition étant mappée à une partition physique. Une autre technique plus flexible pour le rééquilibrage des partitions est le partitionnement virtuel : les clés de partition sont mappées au même nombre de partitions virtuelles, lesquelles sont mappées à un nombre plus restreint de partitions physiques. Dans cette approche, une application localise les données à l'aide d'une clé de partition faisant référence à une partition virtuelle, et le système mappe les partitions virtuelles aux partitions physiques de manière transparente. Le mappage entre une partition virtuelle et une partition physique peut changer sans qu'il ne faille modifier le code de l'application pour utiliser un autre ensemble de clés de partition.

La stratégie de plage. Cette stratégie regroupe les éléments associés dans une même partition et les ordonne par clé de partition (les clés de partition sont séquentielles). Cette stratégie est utile pour les applications qui récupèrent souvent des ensembles d'éléments à l'aide de requêtes de plage (des requêtes qui renvoient un ensemble d'éléments de données pour une clé de partition comprise dans la plage donnée). Par exemple, si une application doit régulièrement retrouver toutes les commandes passées au cours d'un mois donné, ces données peuvent être récupérées plus rapidement si toutes les commandes d'un mois sont stockées dans la même partition par ordre de date et d'heure. Si les commandes étaient stockées dans des partitions différentes, elles devraient être récupérées individuellement au moyen de très nombreuses requêtes ponctuelles (des requêtes qui renvoient un seul élément de données). La figure suivante illustre le processus de stockage d'ensembles séquentiels (plages) de données dans une partition.



Dans cet exemple, la clé de partition est une clé composite contenant le mois de la commande comme élément le plus important, suivi du jour et de l'heure de la commande. Les données des commandes sont triées naturellement à mesure que de nouvelles commandes sont créées et ajoutées à une partition. Certains magasins de données prennent en charge les clés de partition en deux parties contenant une clé de partition qui identifie la partition et une clé de ligne qui identifie un élément unique de la partition. Les données sont généralement conservées dans l'ordre des clés de ligne dans la partition. Les éléments sujets aux requêtes de plage et devant être regroupés peuvent utiliser une clé qui a la même valeur pour la clé de partition mais une valeur unique pour la clé de ligne.

La stratégie de hachage. Cette stratégie vise à réduire les risques de points d'accès (des partitions qui reçoivent une charge disproportionnée). Elle répartit les données dans les partitions de manière à obtenir un équilibre entre la taille de chaque partition et la charge moyenne supportée par chaque partition. La logique de partitionnement fait en sorte que la partition stocke un élément en fonction du hachage d'un ou plusieurs attributs des données. La fonction de hachage sélectionnée doit répartir les données de façon égale entre les partitions, en introduisant éventuellement un élément aléatoire dans le calcul. La figure suivante illustre le processus de partitionnement des données de locataire sur la base d'un hachage des identifiants des locataires.



Pour comprendre l'avantage de la stratégie de hachage par rapport aux autres stratégies de partitionnement, il faut noter qu'une application à plusieurs locataires qui accueille de nouveaux locataires de façon séquentielle peut attribuer les locataires à des partitions dans le magasin de données. Avec la stratégie de plage, les données pour les locataires 1 à n seront toutes stockées dans la partition A, les données pour les locataires n+1 à m seront toutes stockées dans la partition B, et ainsi de suite. Si les locataires enregistrés le plus récemment sont également les plus actifs, la plupart des activités de données auront lieu dans un petit nombre de partitions, ce qui pourrait provoquer une surcharge des points d'accès. En revanche, la stratégie de hachage alloue des locataires aux partitions en fonction d'un hachage de leur identifiant de locataire. Cela signifie que les locataires séquentiels seront probablement alloués à des partitions différentes, ce qui permet de répartir la charge entre elles. La figure précédente illustre cette procédure pour les locataires 55 et 56.

Ces trois stratégies de partitionnement présentent les caractéristiques et les avantages suivants :

- **Recherche.** Cette stratégie vous offre davantage de contrôle sur la configuration et l'utilisation des partitions. L'utilisation de partitions virtuelles réduit l'impact du rééquilibrage des données, car de nouvelles partitions physiques peuvent être ajoutées pour compenser le scénario d'usage. Le mappage entre une partition virtuelle et les partitions physiques qui implémentent la partition peut être modifié sans affecter le code d'application qui utilise une clé de partition pour stocker et récupérer des données. La recherche d'emplacements de partitions peut imposer une charge supplémentaire.
- **Plage.** Cette stratégie est facile à implémenter et fonctionne bien avec les requêtes de plage, car elles peuvent souvent récupérer plusieurs éléments de données depuis une partition unique en une seule opération. Cette stratégie simplifie la gestion des données. Par exemple, si des utilisateurs d'une même région se trouvent dans la même partition, les mises à jour peuvent être programmées dans chaque fuseau horaire en fonction de la charge et du patron de demande locaux. Toutefois, cette stratégie n'offre pas un équilibre optimal entre les partitions. Le rééquilibrage des partitions est complexe et peut ne pas résoudre le problème de charge inégale si la majorité de l'activité concerne des clés de partition adjacentes.
- **Hachage.** Cette stratégie offre une meilleure chance d'obtenir une répartition plus égale des données et de la charge. Le routage des demandes peut se faire directement à l'aide de la fonction de hachage. Il n'est pas nécessaire de tenir une carte. Notez que le calcul du hachage peut imposer une charge supplémentaire. Le rééquilibrage des partitions est également difficile.

Les systèmes de partitionnement les plus courants implémentent l'une des méthodes décrites ci-dessus, mais vous devriez également tenir compte des exigences commerciales de vos applications et de leurs modes d'utilisation des données. Par exemple, dans une application à plusieurs locataires :

- Vous pouvez partitionner les données en fonction du scénario d'usage. Vous pourriez séparer les données pour les locataires très volatils dans des partitions distinctes. Cela pourrait améliorer la vitesse d'accès aux données pour les autres locataires.
- Vous pouvez partitionner des données en fonction de l'emplacement des locataires. Vous pouvez enregistrer les données de locataires situés dans une région géographique spécifique hors ligne pour la sauvegarde et la maintenance pendant les heures creuses dans cette région, tandis que les données de locataires dans d'autres régions restent en ligne et accessibles pendant leurs heures de bureau.
- Les locataires à forte valeur pourraient se voir attribuer leurs propres partitions privées, hautement performantes et à charge légère, tandis que les locataires à plus faible valeur partageront peut-être des partitions plus denses et plus volumineuses.
  Les données des locataires qui nécessitent un haut degré d'isolement et de confidentialité des données peuvent être stockées sur un serveur totalement distinct.
- Les données des locataires qui nécessitent un haut degré d'isolement et de confidentialité des données peuvent être stockées sur un serveur totalement distinct.

Opérations de mise à l'échelle et de déplacement de données

Chacune des stratégies de partitionnement implique des capacités et des niveaux de complexité différents pour la gestion de la mise à l'échelle, des déplacements de données et du maintien de l'état.

La stratégie de recherche permet d'effectuer les opérations de mise à l'échelle et de déplacement de données au niveau de l'utilisateur, que ce soit en ligne ou hors ligne. La technique consiste à suspendre certaines des activités des utilisateurs, voire toutes (éventuellement pendant les heures creuses), à déplacer les données vers la nouvelle partition virtuelle ou physique, à modifier les mappages, à invalider ou à actualiser tous les caches qui contiennent ces données, puis à autoriser la reprise des activités des utilisateurs. Il n'est pas rare que ce type d'opération puisse être gérée de façon centralisée. Pour permettre d'adopter une stratégie de recherche, l'état doit pouvoir être mis en cache facilement et être compatible avec les réplicas.

La stratégie de plage impose certaines restrictions concernant les opérations de mise à l'échelle et de mouvement des données, qui doivent généralement être effectuées lorsqu'une partie ou la totalité du magasin de données est hors connexion, car les données doivent être divisées et fusionnées sur l'ensemble des partitions. Transférer les données pour rééquilibrer les partitions ne résoudra pas forcément le problème de charge inégale si la majorité de l'activité porte sur des clés de partition adjacentes ou des identifiants de données qui se trouvent dans la même plage. La stratégie de plage peut également nécessiter qu'un certain état soit conservé afin de mettre en correspondance des plages avec les partitions physiques.

La stratégie de hachage rend les opérations de mise à l'échelle et de déplacement des données plus complexes, car les clés de partitionnement sont des hachages des clés de partition ou des identificateurs de données. Le nouvel emplacement de chaque partition doit être déterminé à partir de la fonction de hachage ou de la fonction modifiée, afin de fournir des mappages corrects. Toutefois, la stratégie de hachage ne nécessite pas la maintenance de l'état.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- Le partitionnement est complémentaire aux autres formes de partitionnement, telles que le partitionnement vertical et le partitionnement fonctionnel. À titre d'exemple, une seule partition peut contenir des entités qui ont été partitionnées verticalement, et une partition fonctionnelle peut être implémentée sous forme de partitions multiples. Pour plus d'informations sur le partitionnement, consultez l'assistance relative au partitionnement des données.
- Conservez les partitions équilibrées afin qu'elles gèrent toutes un volume similaire d'E/S. Comme les données sont insérées et supprimées, il est nécessaire de rééquilibrer régulièrement les partitions afin de garantir une distribution égale et de réduire le nombre de points d'accès. Le rééquilibrage peut être une opération coûteuse. Afin de réduire la nécessité de rééquilibrage, visez la croissance en veillant à ce que chaque partition contienne suffisamment d'espace libre pour traiter le volume attendu de changements. Vous devez également développer des stratégies et des scripts à utiliser pour rééquilibrer rapidement des partitions si la nécessité se présente.
- Utilisez des données stables pour la clé de partition. Si la clé de partition change, il se peut que l'élément de données correspondant doive passer d'une partition à l'autre, ce qui engendre une augmentation du volume de travail réalisé par des opérations de mise à jour. Par conséquent, évitez de baser la clé de partition sur des informations potentiellement instables. Au lieu de cela, recherchez des attributs qui ne varient pas ou qui forment naturellement une clé.
- Veillez à ce que les clés de partitions soient uniques. À titre d'exemple, évitez d'utiliser des champs auto-incrémentés comme clé de partition. Dans certains systèmes, les champs auto-incrémentés ne peuvent pas être coordonnés entre différentes partitions. Ainsi, il se peut que plusieurs partitions partagent la même clé de partition.
    - Les valeurs auto-incrémentées dans d'autres domaines qui ne sont pas des clés de partition peuvent également causer des problèmes. À titre d'exemple, si vous utilisez des champs auto-incrémentés pour générer des identifiants uniques, deux éléments différents situés sur des partitions différentes peuvent être assignés au même identifiant.

Il se peut qu'il ne soit pas possible de concevoir une clé de partition qui répond aux critères de toutes les requêtes possibles par rapport aux données. Partitionnez les données pour prendre en charge les requêtes les plus fréquemment exécutées. Si nécessaire, créez des tables d'index pour prendre en charge des requêtes qui récupèrent des données utilisant des critères basés sur des attributs qui ne font pas partie de la clé de partition. Pour plus d'informations, consultez le Patron Index Table (Tableau indexé).

- Les requêtes qui n'ont accès qu'à une seule partition sont plus efficaces que celles qui récupèrent des données à partir de plusieurs partitions. Évitez donc d'implémenter un système de partitionnement qui ferait que des applications exécuteraient de grands volumes de requêtes qui accéderaient à des données stockées dans plusieurs partitions. N'oubliez pas qu'une partition unique peut contenir les données pour plusieurs types d'entités. Envisagez la dénormalisation de vos données pour conserver des entités apparentées qui sont régulièrement interrogées ensemble (telles que les informations sur les clients et les commandes que ces derniers ont passées) dans la même partition, afin de réduire le nombre de lectures séparées qu'une application effectue.

    - Si une entité dans une partition fait référence à une entité stockée dans une autre partition, incluez la clé de partition pour la seconde entité comme partie du schéma pour la première entité. Cela peut améliorer les performances des requêtes qui font référence à des données apparentées sur l'ensemble des partitions.

- Si une application doit exécuter des requêtes qui récupèrent des données à partir de plusieurs partitions, il peut être possible de récupérer ces données en utilisant des tâches parallèles. Les exemples incluent des requêtes de distribution ramifiée où les données issues de plusieurs partitions sont récupérées en parallèle, puis regroupées en un seul résultat. Toutefois, cette approche ajoute inévitablement une certaine complexité à la logique d'accès aux données d'une solution.

- Pour plusieurs applications, la création d'un grand nombre de petites partitions peut être plus efficace que de disposer d'un petit nombre de partitions étendues qui peuvent fournir des opportunités accrues pour l'équilibrage de charge. Cela peut être utile si vous anticipez le besoin de migrer des partitions d'un emplacement physique à un autre. Il est plus rapide de déplacer une petite partition qu'une grande.

- Assurez-vous que les ressources disponibles pour chaque nœud de stockage de partition suffisent pour gérer les exigences d'évolutivité quant à la taille des données et au débit. Pour plus d'informations, consultez la section « Conception de partitions pour des raisons d'évolutivité » dans les Conseils sur le partitionnement des données.

- Envisagez la réplication des données de référence sur toutes les partitions. Si une opération qui extrait les données d'une partition fait également référence à des données statiques ou se déplaçant lentement dans le cadre de la même requête, ajoutez ces données à la partition. L'application peut ensuite récupérer facilement toutes les données pour la requête, sans qu'il soit nécessaire de faire un aller-retour supplémentaire vers un magasin de données distinct.

    - Si les données de référence conservées dans plusieurs partitions changent, le système doit synchroniser ces modifications sur l'ensemble des partitions. Le système peut être temporairement instable pendant le déroulement de cette synchronisation. Si vous faites cela, vous devez concevoir vos applications pour qu'elles n'en pâtissent pas.

- Il peut être difficile de conserver une intégrité et une cohérence référentielles entre les partitions. Vous devez donc limiter les opérations qui affectent les données sur plusieurs partitions. Si une application doit modifier les données sur l'ensemble des partitions, évaluez si la cohérence totale des données est bel et bien requise. Au lieu de cela, une approche commune dans le cloud consiste à implémenter une cohérence éventuelle. Les données de chaque partition sont mises à jour séparément, et la logique de l'application doit veiller à ce que les mises à jour se terminent toutes correctement, ainsi qu'à gérer les incohérences qui peuvent dériver de l'interrogation des données pendant qu'une opération probablement cohérente est en cours d'exécution. Pour plus d'informations sur l'implémentation de la cohérence à terme, consultez Introduction à la cohérence des données.

- Configurer et gérer un grand nombre de partitions peut représenter un défi. Des tâches telles que le pilotage, la sauvegarde ou la recherche de cohérence, la consignation ou le contrôle doivent être accomplies sur plusieurs partitions et serveurs, potentiellement conserver dans plusieurs emplacements. Ces tâches sont susceptibles d'être implémentées à l'aide de scripts ou d'autres solutions d'automatisation, mais qui pourraient ne pas complètement éliminer les exigences administratives supplémentaires.
- Les partitions peuvent être géolocalisées, si bien que les données qu'elles contiennent sont proches des instances d'une application qui les utilisent. Cette approche peut améliorer considérablement les performances, mais exige un examen supplémentaire pour les tâches qui doivent accéder à plusieurs partitions dans différents endroits.

Quand utiliser ce patron

Utilisez ce patron lorsqu'une banque de données est susceptible d'avoir besoin d'évoluer au-delà des ressources disponibles sur un nœud de stockage unique ou d'améliorer les performances en réduisant la contention dans un magasin de données.

Le partitionnement vise principalement à améliorer les performances et l'évolutivité d'un système, mais à l'instar d'un sous-produit, il peut également améliorer la disponibilité selon la manière dont les données sont divisées en partitions séparées. Un échec dans une partition n'empêche pas nécessairement une application d'accéder aux données détenues dans d'autres partitions, et un opérateur peut exécuter la maintenance ou le rétablissement d'une ou plusieurs partitions sans rendre inaccessibles les données d'une application. Pour plus d'informations, consultez les Conseils sur le partitionnement des données.

Exemple

L'exemple suivant en c# utilise un ensemble de bases de données SQL Server fonctionnant comme des partitions. Chaque base de données contient un sous-ensemble des données utilisées par une application. L'application récupère les données qui sont réparties entre les partitions en utilisant sa propre logique de partitionnement (il s'agit d'un exemple de requête de distribution ramifiée). Les détails des données qui se trouvent dans chaque partition sont retournés par une méthode appelée GetShards. Cette méthode renvoie une liste d'objets ShardInformation, où le type ShardInformation contient un identificateur pour chaque partition et la chaîne de connexion SQL Server qu'une application doit utiliser pour se connecter à la partition (les chaînes de connexion ne sont pas visibles dans l'exemple de code).

```
private IEnumerable<ShardInformation> GetShards()
{
  // Cela permet de récupérer les informations de connexion depuis un magasin de partitions
  // (généralement une base de données racine).
  return new[]
  {
    new ShardInformation
    {
      Id = 1,
      ConnectionString = ...
    },
    new ShardInformation
    {
      Id = 2,
      ConnectionString = ...
    }
  };
}
```

Le code ci-dessous montre comment l'application utilise la liste d'objets ShardInformation pour exécuter une requête qui extrait des données de chaque partition en parallèle. Les détails de la requête ne s'affichent pas, mais dans cet exemple, les données récupérées contiennent une chaîne qui peut contenir des informations telles que le nom du client si les partitions contiennent les détails des clients. Les résultats sont regroupés dans une collection ConcurrentBag dans l'attente d'un traitement réalisé par l'application.

```
// Récupérer les partitions comme une instance ShardInformation[].
var shards = GetShards();

var results = new ConcurrentBag<string>();

// Exécutez la requête par rapport à chaque partition dans la liste de partitions.
// Cette liste devrait généralement être récupérée à partir de la configuration
// ou à partir d'une racine/d'un magasin de partition maître.
Parallel.ForEach(shards, shard =>
{
  // REMARQUE : la gestion transitoire des pannes n'est pas incluse,
  // mais elle doit être incorporée quand elle est utilisée dans une application du monde réel.
  en utilisant (var con = new SqlConnection(shard.ConnectionString))
  {
    con.Open();
    var cmd = new SqlCommand("SELECT ... FROM ...", con);

    Trace.TraceInformation("Executing command against shard: {0}", shard.Id);

    var reader = cmd.ExecuteReader();
    // Lire les résultats dans une structure de données thread-safe.
    tandis que (reader.Read())
    {
      results.Add(reader.GetString(0));
    }
  }
});

Trace.TraceInformation("Fanout query complete - Record Count: {0}",
                       results.Count);
```

Patrons et informations connexes

Les patrons et les lignes directrices suivantes peuvent également être pertinents lors de l'implémentation de ce patron :

- Manuel de cohérence de données. Il peut être nécessaire de conserver la cohérence des données réparties sur différentes partitions. Résume les questions autour du respect de la cohérence des données distribuées et décrit les avantages et les inconvénients des différents modèles de cohérence.
- Conseils sur le partitionnement des données. Le partitionnement d'une banque de données peut introduire toute une série de problèmes supplémentaires. Décrit ces problèmes liés au partitionnement des banques de données dans le cloud pour améliorer l'évolutivité, réduire la contention et optimiser les performances.
- Patron Index Table (Tableau indexé). Il n'est parfois pas possible de prendre complètement en charge les requêtes uniquement au niveau de la conception de la clé de partition. Permet à une application de récupérer rapidement des données à partir d'une banque de données étendue en spécifiant une clé autre que la clé de partition.
- Patron Materialized View (Vue matérialisée). Pour conserver les performances de certaines opérations de requête, il est utile de créer des vues matérialisées qui regroupent et résument les données, surtout si ce récapitulatif des données est basé sur des informations distribuées entre les différentes partitions. Décrit comment générer et ajouter ces vues.
- Leçons sur les partitions sur le blog Adding Simplicity.
- Partitionnement des bases de données sur le site web CodeFutures.
- Introduction aux stratégies d'évolutivité : partitionnement des bases de données sur le blog de Max Indelicato.
- Création de bases de données évolutives : avantages et inconvénients de plusieurs schémas de partitionnement de bases de données sur le blog de Dare Obasanjo.

Patron Sidecar (Side-car)

Déployez des composants d'une application dans un conteneur ou un processus distinct pour fournir l'isolation et l'encapsulation. Ce patron peut également permettre aux applications d'être composées de technologies et de composants hétérogènes.

Ce patron se nomme side-car parce qu'il ressemble à un side-car fixé à une moto. Dans ce patron, le side-car est fixé à une application parente et il propose des fonctionnalités de prise en charge pour l'application. Le side-car partage également la même mise en service que l'application parente et est créé et mis hors service en même temps que celle-ci. Le patron Sidecar (Side-car) est parfois appelé « patron sidekick ». Il s'agit d'un patron de décomposition.

Contexte et problème

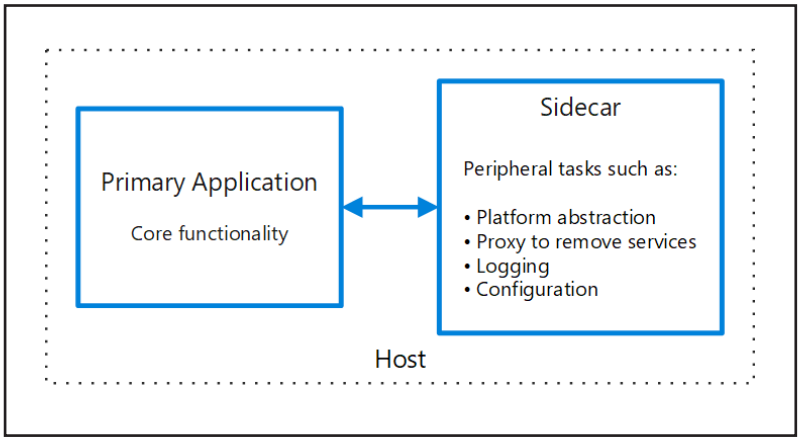
Applications et services nécessitent souvent des fonctionnalités associées, comme des services de pilotage, de journalisation, de configuration et de mise en réseau. Ces tâches périphériques peuvent être implémentées comme composants ou services distincts.

Si elles sont étroitement intégrées dans l'application, elles peuvent exécuter le même processus que l'application, en utilisant efficacement les ressources partagées. Cependant, cela signifie également qu'elles ne sont pas bien isolées, et une panne de l'un de ces composants peut en affecter d'autres, voire l'ensemble de l'application. En outre, elles doivent généralement être implémentées en utilisant le même langage que l'application parente. Par conséquent, le composant et l'application sont étroitement dépendants l'un de l'autre.

Si l'application est décomposée en services, chaque service peut être construit à l'aide de technologies et de langages différents. Outre une flexibilité supérieure, cela signifie que chaque composant a ses propres dépendances et nécessite des bibliothèques spécifiques au langage pour accéder à la plateforme sous-jacente et à n'importe quelle ressource partagée avec l'application parente. En outre, le déploiement de ces fonctionnalités comme des services distincts peut ajouter de la latence à l'application. Gérer le code et les dépendances pour ces interfaces spécifiques au langage peut également être vecteur d'une importante complexité, notamment pour l'hébergement, le déploiement et la gestion.

Solution

Regroupez un ensemble cohérent de tâches avec l'application principale, mais placez-les dans leur propre processus ou conteneur, ce qui permet de proposer une interface homogène pour des services de plateforme sur l'ensemble des langages.



Un service de side-car ne fait pas nécessairement partie de l'application, mais il est connecté à celle-ci. Il va partout où l'application parente va. Les side-cars prennent en charge des processus ou des services qui sont déployés avec l'application principale. Le side-car est attaché à une moto, et chaque moto peut avoir son propre side-car. De la même manière, un service de side-car partage le sort de son application parente. Pour chaque instance de l'application, une instance du side-car est déployée et hébergée en parallèle.

Avantages de l'utilisation du patron Sidecar (Side-car) :

- Un side-car est indépendant de son application principale en ce qui concerne l'environnement d'exécution et le langage de programmation. Vous n'avez donc pas besoin de développer un side-car par langage.
- Le side-car peut accéder aux mêmes ressources que l'application principale. À titre d'exemple, un side-car peut surveiller les ressources système utilisées par le side-car et l'application principale.
- En raison de sa proximité avec l'application principale, il n'y a pas de latence importante lors de la communication entre eux.
- Même pour les applications qui ne fournissent pas de mécanisme d'évolutivité, vous pouvez utiliser un side-car pour étendre les fonctionnalités en l'attachant comme processus à part entière à un même hôte ou sous-conteneur comme l'application principale.

Le patron Sidecar (Side-car) est souvent utilisé avec des conteneurs et appelé conteneur side-car ou conteneur sidekick.

Problèmes et considérations

- Envisagez le déploiement et le format de package que vous utiliserez pour déployer des services, des processus ou des conteneurs. Les conteneurs sont particulièrement bien adaptés au patron Sidecar (Side-car).
- Lorsque vous créez un service side-car, déterminez avec précaution le mécanisme de communication entre les processus. Essayez d'utiliser des technologies indépendantes du langage ou de l'infrastructure, à moins que les exigences de performance rendent cela peu pratique.
- Avant de mettre des fonctionnalités dans un side-car, examinez si cela fonctionnerait mieux comme un service distinct ou un démon plus traditionnel.
- Déterminez si la fonctionnalité pourrait être implémentée comme bibliothèque ou à l'aide d'un mécanisme d'extension traditionnel. Des bibliothèques spécifiques au langage peuvent avoir un niveau plus approfondi d'intégration et entraîner moins de frais liés au réseau.

Quand utiliser ce patron

Utilisez ce patron dans les circonstances suivantes :

- Votre application principale utilise un ensemble hétérogène de langages et d'infrastructures. Un composant situé dans un service side-car peut être utilisé par des applications écrites dans des langages différents à l'aide de plusieurs infrastructures.
- Un composant est détenu par une équipe distante ou une autre organisation.
- Un composant ou une fonction doit être co-localisé(e) sur le même hôte que l'application.
- Vous avez besoin d'un service qui partage la mise en service globale de votre application principale, mais qui peut être mis à jour de manière indépendante.

- Vous avez besoin d'un contrôle précis des limites de ressources pour une ressource ou un composant spécifique. À titre d'exemple, vous pourriez vouloir restreindre la quantité de mémoire utilisée par un composant spécifique. Vous pouvez déployer le composant comme un side-car et gérer l'utilisation de la mémoire indépendamment de l'application principale.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Quand la communication interprocessus doit être optimisée. La communication entre une application parente et des services side-car implique des frais généraux, liés notamment à la latence dans les appels. Il ne s'agit peut-être pas d'un compromis intéressant pour les interfaces verbeuses.

- Pour les petites applications où le coût de la ressource de déploiement d'un service side-car pour chaque instance ne rend pas l'avantage du cloisonnement intéressant.

- Quand le service doit être mis à l'échelle différemment ou indépendamment des applications principales. Le cas échéant, il peut être préférable de déployer la fonctionnalité comme un service distinct.

Exemple

Le patron Sidecar (Side-car) peut être appliqué à de nombreux scénarios. Voici quelques exemples courants :

- API d'infrastructure. L'équipe de développement d'infrastructure crée un service, qui est déployé avec chaque application, plutôt qu'une bibliothèque cliente spécifique à la langue pour accéder à l'infrastructure. Le service est chargé en tant que side-car et offre un niveau commun pour les services d'infrastructure, y compris la journalisation, les données sur l'environnement, le magasin de configuration, la détection, les bilans d'intégrité et la surveillance. Le side-car contrôle également le processus (ou conteneur) ainsi que l'environnement hôte de l'application parente et enregistre les informations dans un service centralisé.

- Gérer NGINX/HAProxy. Déployez NGINX avec un service side-car qui contrôle l'état de l'environnement, puis met à jour le fichier de configuration NGINX et recycle le processus lorsque l'état doit être modifié.

- Ambassadeur side-car. Déployez un service ambassadeur en tant que side-car. L'application appelle par l'intermédiaire de l'ambassadeur, qui gère la journalisation des demandes, le routage, la coupure du circuit et les autres fonctionnalités liées à la connectivité.

- Proxy de déchargement. Placez un proxy NGINX devant une instance de service node.js, de façon à gérer la diffusion du contenu de fichier statique pour le service

Conseils connexes

- Patron Ambassador (Ambassadeur)

Patron Static Content Hosting (Hébergement de contenu statique)

Déployez du contenu statique dans un service de stockage basé sur le cloud qui peut les fournir directement au client. Cela peut vous permettre de réduire le nombre d'instances de calcul potentiellement onéreuses nécessaires.

Contexte et problème

En général, les applications web comprennent des éléments de contenu statique. Ce contenu statique peut inclure des pages HTML et d'autres ressources, telles que des images et des documents qui sont disponibles pour le client, soit sur une page HTML (comme les images incorporées, les feuilles de style et les fichiers JavaScript côté client), soit dans des téléchargements distincts (des documents PDF, par exemple).

Bien que les serveurs web soient configurés de manière à optimiser les demandes grâce à l'exécution efficace de code de page dynamique et à la mise en cache de sortie, ils doivent toujours traiter les demandes pour télécharger du contenu statique. Des cycles de traitement sont ainsi gaspillés, alors qu'ils pourraient être utilisés à meilleur escient.

Solution

Dans la plupart des environnements d'hébergement cloud, il est possible de réduire la nécessité de recours aux instances de calcul (par exemple, grâce à l'utilisation d'une instance plus petite ou d'un plus petit nombre d'instances), en plaçant certaines pages statiques et ressources d'une application au sein d'un service de stockage. Le coût d'un espace de stockage dans le cloud est généralement beaucoup moins élevé que celui d'instances de calcul.

Si vous hébergez des composants d'une application dans un service de stockage, les principales considérations sont liées au déploiement de l'application et à la sécurisation des ressources qui ne sont pas destinées à être accessibles aux utilisateurs anonymes.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

- Le service de stockage hébergé doit présenter un point de terminaison HTTP auquel les utilisateurs peuvent accéder pour télécharger les ressources statiques. Certains services de stockage sont également compatibles avec HTTPS. Il est donc possible d'héberger des ressources dans les services de stockage qui requièrent le protocole SSL.
- Afin d'optimiser les performances et la disponibilité, nous vous conseillons d'utiliser un réseau de distribution de contenu (CDN) pour mettre en cache les contenus du conteneur de stockage dans plusieurs datacenters du monde entier. Il est toutefois possible que l'utilisation du CDN soit payante. Les comptes de stockage sont souvent géo-répliqués par défaut à des fins de résilience pour faire face à des événements susceptibles d'avoir un impact sur un datacenter. Cela signifie que l'adresse IP peut varier, mais que l'URL reste la même.
- Lorsqu'une partie du contenu est située dans un compte de stockage et qu'une autre se trouve dans une instance de calcul hébergée, il devient plus difficile de déployer une application et de la mettre à jour. Il se peut que vous deviez procéder à des déploiements distincts et créer une nouvelle version de l'application et du contenu afin d'en faciliter la gestion, surtout si le contenu statique inclut des fichiers de script ou des composants d'interface utilisateur. Toutefois, si seules les ressources statiques doivent être mises à jour, elles peuvent simplement être téléchargées dans le compte de stockage sans nécessiter le redéploiement du package d'application.

- Les services de stockage peuvent ne pas accepter l'utilisation de noms de domaine personnalisés. Dans ce cas, il est nécessaire de spécifier l'URL complète des ressources dans les liens, car elles se trouveront dans un autre domaine que le contenu généré dynamiquement qui comporte les liens.
- Les conteneurs de stockage doivent être configurés pour un accès en lecture public, mais il est essentiel de s'assurer qu'ils ne sont pas configurés pour un accès en écriture public afin d'empêcher les utilisateurs de pouvoir télécharger du contenu. Pensez à utiliser un jeton ou une clé à accès restreint pour contrôler l'accès aux ressources qui ne devraient pas être disponibles de façon anonyme. Voir le Patron Valet Key (Clé à accès restreint) pour plus d'informations.

Quand utiliser ce patron

Ce patron est utile dans les cas suivants :

- Réduire le coût d'hébergement des sites web et applications qui contiennent des ressources statiques.
- Réduire le coût d'hébergement des sites web qui ne comportent que du contenu et des ressources statiques. Selon les capacités du système de stockage de l'hébergeur, il peut être possible d'héberger l'ensemble d'un site web entièrement statique dans un compte de stockage.
- Exposer du contenu et des ressources statiques pour les applications qui s'exécutent dans d'autres environnements d'hébergement ou sur des serveurs locaux.
- Placer le contenu dans plusieurs zones géographiques à l'aide d'un réseau de distribution de contenu qui met en cache le contenu du compte de stockage dans plusieurs datacenters du monde entier.
- Contrôler les coûts et l'utilisation de la bande passante. L'utilisation d'un compte de stockage distinct pour tout ou partie du contenu statique permet de séparer les frais plus facilement des coûts d'hébergement et d'exécution.

Ce patron n'est pas nécessairement utile dans les situations suivantes :

- L'application doit exécuter un traitement sur le contenu statique avant de le transmettre au client. Par exemple, il peut être nécessaire d'ajouter un horodatage à un document.
- Le volume du contenu statique est très faible. Les frais occasionnés par la récupération de ce contenu dans des espaces de stockage distincts peuvent dépasser les économies réalisées en le séparant de la ressource de calcul.

Exemple

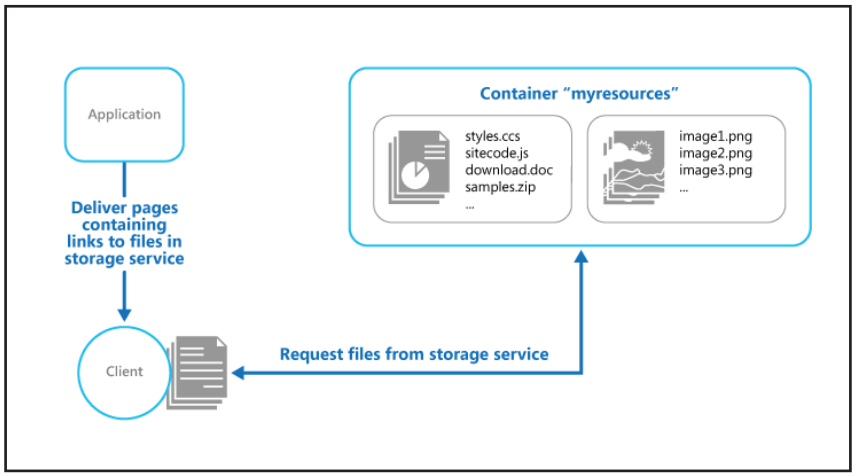
Le contenu statique situé dans le stockage Blob Azure est accessible directement à l'aide d'un navigateur web. Azure propose une interface basée sur HTTP pour le stockage accessible au public. Par exemple, le contenu d'un conteneur de stockage Blob Azure est accessible à l'aide d'une URL présentant la forme suivante :

http://[nom-compte-stockage].blob.core.windows.net/[nom-conteneur]/[nom-fichier]

Lors du téléchargement du contenu, il est nécessaire de créer un ou plusieurs conteneurs d'objets blob pour contenir les dossiers et documents. Notez que l'autorisation par défaut pour un nouveau conteneur est Privé. Vous devez la modifier en Public pour permettre aux clients d'accéder au contenu. S'il est nécessaire de protéger le contenu contre l'accès anonyme, vous pouvez implémenter le Patron Valet Key (Clé à accès restreint) qui oblige les utilisateurs à présenter un jeton valide pour télécharger les ressources.

Les concepts de service BLOB contiennent des informations sur le stockage d'objets blob et sur les façons d'y accéder et de les utiliser.

Les liens sur chaque page spécifient l'URL de la ressource. Le client y accèdera directement depuis le service de stockage. La figure suivante illustre la transmission des parties statiques d'une application directement depuis un service de stockage.



Les liens des pages envoyées au client doivent comprendre l'URL complète de la ressource et du conteneur d'objets blob. Par exemple, une page qui contient un lien vers une image dans un conteneur public peut comporter le code HTML suivant.

```
<img src="http://mystorageaccount.blob.core.windows.net/myresources/image1.png"
     alt="My image" />
```

Si les ressources sont protégées à l'aide d'une clé à accès restreint, telle qu'une signature d'accès partagé Azure, cette signature doit figurer dans les URL des liens.

Une solution nommée StaticContentHosting, qui illustre l'utilisation du stockage externe pour les ressources statiques, est disponible sur GitHub. Le projet StaticContentHosting.Cloud contient des fichiers de configuration qui spécifient le compte de stockage et le conteneur où se trouve le contenu statique.

```
<Setting name="StaticContent.StorageConnectionString"
         value="UseDevelopmentStorage=true" />
<Setting name="StaticContent.Container" value="static-content" />
```

La classe Settings dans le fichier Settings.cs du projet StaticContentHosting.Web contient des méthodes pour extraire ces valeurs et générer une valeur de chaîne comportant l'URL de conteneur du compte de stockage dans le cloud.

```
public class Settings
{
  public static string StaticContentStorageConnectionString {
    get
    {
      return RoleEnvironment.GetConfigurationSettingValue(
                            “StaticContent.StorageConnectionString”);
    }
  }

  public static string StaticContentContainer
  {
    get
    {
      return RoleEnvironment.GetConfigurationSettingValue(“StaticContent.Container”);
    }
  }

  public static string StaticContentBaseUrl
  {
    get
    {
      var account = CloudStorageAccount.Parse(StaticContentStorageConnectionString);

      return string.Format(“{0}/{1}”, account.BlobEndpoint.ToString().TrimEnd(‘/’),
                                      StaticContentContainer.TrimStart(‘/’));
    }
  }
}
```

La classe StaticContentUrlHtmlHelper du fichier StaticContentUrlHtmlHelper.cs contient une méthode nommée StaticContentUrl qui génère une URL comportant le chemin d'accès au compte de stockage dans le cloud si l'URL qui lui est transmise commence par le caractère de chemin d'accès racine ASP.NET (~).

```
public static class StaticContentUrlHtmlHelper
{
  public static string StaticContentUrl(this HtmlHelper helper, string contentPath)
  {
    if (contentPath.StartsWith(“~”))
    {
      contentPath = contentPath.Substring(1);
    }

    contentPath = string.Format(“{0}/{1}”, Settings.StaticContentBaseUrl.TrimEnd(‘/’),
                                contentPath.TrimStart(‘/’));

    var url = new UrlHelper(helper.ViewContext.RequestContext);

    return url.Content(contentPath);
  }
}
```

Le fichier Index.cshtml du dossier Views\Home contient un élément image qui utilise la méthode StaticContentUrl pour créer l'URL de son attribut src.

```
<img src="@Html.StaticContentUrl("~/media/orderedList1.png")" alt="Test Image" />
```

Patrons et informations connexes

- Un exemple illustrant ce patron est disponible sur GitHub.
- Patron Valet Key (Clé à accès restreint). Si les ressources cibles ne sont pas censées être accessibles aux utilisateurs anonymes, il est nécessaire d'implémenter la sécurité dans le magasin où se trouve le contenu statique. Explique l'utilisation d'un jeton ou d'une clé qui permet aux clients de bénéficier d'un accès direct limité à une ressource ou à un service spécifique, comme un service de stockage hébergé dans le cloud.
- Méthode de déploiement efficace d'un site web statique sur Azure sur le blog d'Infosys.
- Concepts de service BLOB

Patron Strangler (Arrêt)

Migrez progressivement un système hérité en remplaçant progressivement des éléments spécifiques de fonctionnalité par de nouveaux services et applications. Toutes les fonctionnalités de l'ancien système finissent par être remplacées par celles du nouveau, ce qui aboutit à l'arrêt de l'ancien système et vous permet de le désactiver.

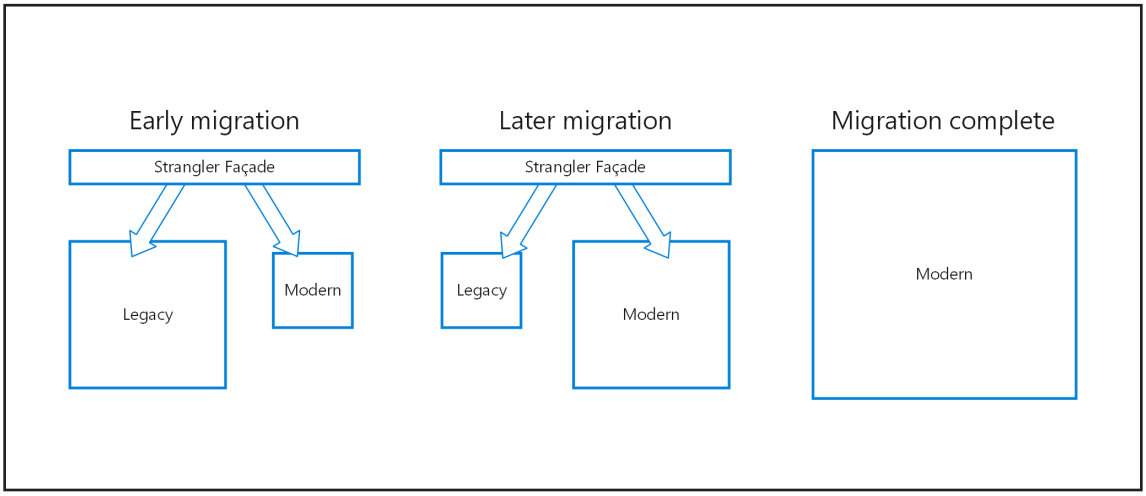
Contexte et problème

À mesure que les systèmes vieillissent, les outils de développement, la technologie d'hébergement et même les architectures système sont susceptibles de devenir obsolètes. Lorsque de nouvelles fonctionnalités sont ajoutées, la complexité de ces applications peut augmenter considérablement, ce qui les rend plus difficiles à gérer et complique l'ajout de nouvelles fonctionnalités.

Remplacer complètement un système complexe peut être une entreprise énorme. En général, vous devez effectuer une migration progressive vers un nouveau système, tout en conservant l'ancien système pour gérer les fonctionnalités qui n'ont pas encore fait l'objet d'une migration. Toutefois, l'utilisation de deux versions distinctes d'une même application signifie que les clients doivent savoir où se trouvent les différentes fonctionnalités. À chaque migration d'une fonctionnalité ou d'un service, vous devez tenir les clients informés du nouvel emplacement.

Solution

Remplacez progressivement des parties spécifiques de fonctionnalités par de nouveaux services et de nouvelles applications. Créez une façade qui intercepte les demandes destinées à l'ancien système principal. La façade achemine ces demandes soit vers l'ancienne application, soit vers les nouveaux services. Les fonctionnalités existantes peuvent être migrées petit à petit vers le nouveau système, et les clients peuvent continuer à utiliser la même interface, sans se douter qu'une migration a eu lieu.



Ce patron permet de limiter les risques liés à la migration et de répartir les efforts de développement sur une période plus longue. Étant donné que la façade redirige les utilisateurs en toute sécurité vers l'application appropriée, vous pouvez ajouter des fonctionnalités au nouveau système au rythme qui vous convient, tout en garantissant le bon fonctionnement de l'ancienne application. À mesure que les fonctionnalités sont migrées vers le nouveau système, l'ancien système finit par ne plus être nécessaire. Une fois ce processus terminé, l'ancien système peut être désactivé sans risque.

Problèmes et considérations

- Réfléchissez à la manière de gérer les services et les magasins de données qui sont potentiellement utilisés par les nouveaux et les anciens systèmes. Assurez-vous que les deux peuvent accéder à ces ressources en même temps.
- Structurez les nouvelles applications et les nouveaux services de manière à ce qu'ils puissent être facilement interceptés et remplacés dans le cadre de migrations futures.
- Une fois la migration terminée, la façade d'arrêt disparaît ou se transforme en adaptateur pour les clients hérités.
- Assurez-vous que la façade parvient à suivre le rythme de la migration.
- Assurez-vous que la façade ne devienne pas un point de défaillance unique ou un goulot d'étranglement des performances.

Quand utiliser ce patron

Utilisez ce patron lors de la migration progressive d'une application principale vers une nouvelle architecture.

Ce patron ne conviendra pas nécessairement dans les cas suivants :

- Lorsque les demandes destinées au système principal ne peuvent pas être interceptées.
- Pour les systèmes plus petits où il n'est pas difficile de procéder à un remplacement global.

Conseils connexes

- Patron Anti-Corruption Layer (niveau de lutte contre la corruption)
- Patron Gateway Routing (routage de passerelle)

Patron Throttling (Limitation)

Contrôlez la consommation de ressources utilisée par une instance d'une application, un locataire individuel ou un service complet. Ce patron peut permettre au système de continuer à fonctionner et à respecter les contrats de niveau de service, même si une augmentation du nombre de demandes fait peser une charge importante sur les ressources.

Contexte et problème

La charge à laquelle une application cloud est soumise varie généralement selon la période, en fonction du nombre d'utilisateurs actifs ou des types d'activités qu'ils exécutent. Par exemple, il est probable qu'un plus grand nombre d'utilisateurs soient actifs pendant les heures de bureau ou que le système doive effectuer des analyses nécessitant une grande puissance de calcul à la fin de chaque mois. Il peut également y avoir des pics d'activité soudains et imprévus. Si les exigences de traitement du système dépassent la capacité des ressources disponibles, il peut enregistrer de mauvaises performances, voire connaître une panne. Si le système doit respecter un contrat de niveau de service, cette panne peut être inacceptable.

Diverses stratégies existent pour gérer la variabilité de la charge dans le cloud, en fonction des objectifs commerciaux associés à l'application. Une de ces stratégies consiste à utiliser la mise à l'échelle automatique afin de faire correspondre les ressources fournies aux besoins de l'utilisateur, et ce, à tout moment. Elle offre la possibilité de répondre en permanence à la demande de l'utilisateur, tout en optimisant les coûts de fonctionnement. Toutefois, bien que la mise à l'échelle automatique permette de déclencher l'approvisionnement de ressources supplémentaires, cet approvisionnement n'est pas immédiat. Si la demande croît rapidement, il peut y avoir une période pendant laquelle il manque des ressources.

Solution

Une autre stratégie consiste à permettre aux applications d'utiliser les ressources jusqu'à un certain seuil, puis à les limiter une fois le seuil atteint. Le système doit surveiller l'utilisation des ressources de sorte que, lorsque le seuil est dépassé, il puisse limiter les demandes d'un ou de plusieurs utilisateurs. Cela permet au système de continuer à fonctionner et à respecter les contrats de niveau de service (SLA) qui sont en place. Pour plus d'informations sur le pilotage de l'utilisation des ressources, voir le guide d'instrumentation et de télémétrie.

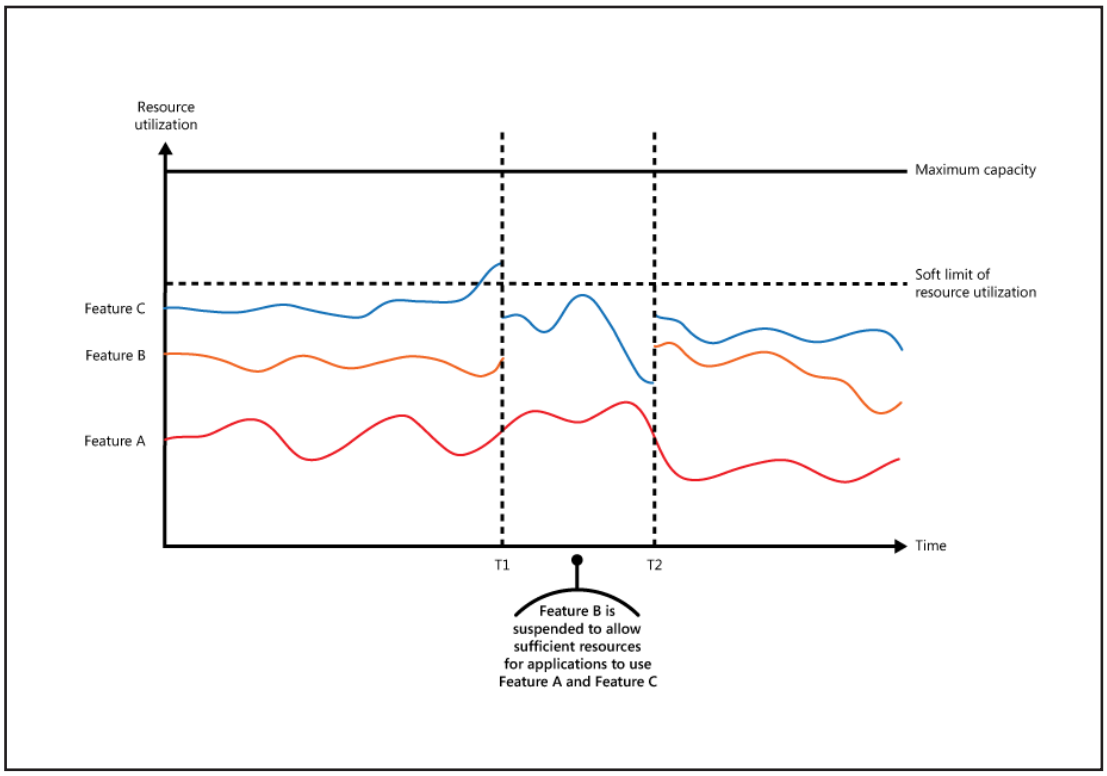
Le système pourrait mettre en œuvre plusieurs stratégies de limitation, par exemple :

- Le rejet des requêtes d'un utilisateur individuel qui a déjà accédé aux API du système plus de n fois par seconde pendant une période donnée. Pour ce faire, le système doit évaluer l'utilisation des ressources pour chaque client ou utilisateur qui exécute une application. Pour de plus amples informations, consultez l'article Conseils pour l'évaluation de l'utilisation des services.
- La désactivation ou la mise à niveau vers une version antérieure des fonctionnalités de certains services accessoires, de sorte que les services essentiels puissent s'exécuter sans encombre avec des ressources suffisantes. Par exemple, si l'application diffuse un signal vidéo en continu, elle pourrait passer à une résolution inférieure.

- Le recours au nivellement de charge pour équilibrer le volume d'activité (cette méthode est traitée plus en détail dans l'article Patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente). Dans un environnement multiclient, cette méthode entraînera une diminution des performances pour chaque client. Si le système doit prendre en charge plusieurs types de clients avec des contrats de niveau de service différents, les tâches associées aux clients importants peuvent être traitées immédiatement. Les requêtes associées aux autres clients peuvent être différées, pour ensuite être traitées lorsque le retard aura été résorbé. Le patron Priority Queue (File d'attente prioritaire) pourrait être utilisé pour faciliter la mise en œuvre de cette méthode.

- Le report des opérations exécutées pour le compte d'applications ou de clients de priorité inférieure. Ces opérations peuvent être suspendues ou limitées. Une exception est alors générée pour informer le client que le système est occupé et que l'opération doit être retentée ultérieurement.

Le graphique ci-dessous illustre l'utilisation des ressources (une combinaison de mémoire, d'unité centrale, de bande passante et d'autres facteurs) par rapport au temps pour les applications qui sollicitent trois fonctions. Une fonction est une zone de fonctionnalité, par exemple un composant qui réalise un ensemble spécifique de tâches, un extrait de code qui effectue un calcul complexe ou un élément qui fournit un service tel qu'un cache en mémoire. Ces fonctions sont nommées A, B et C.

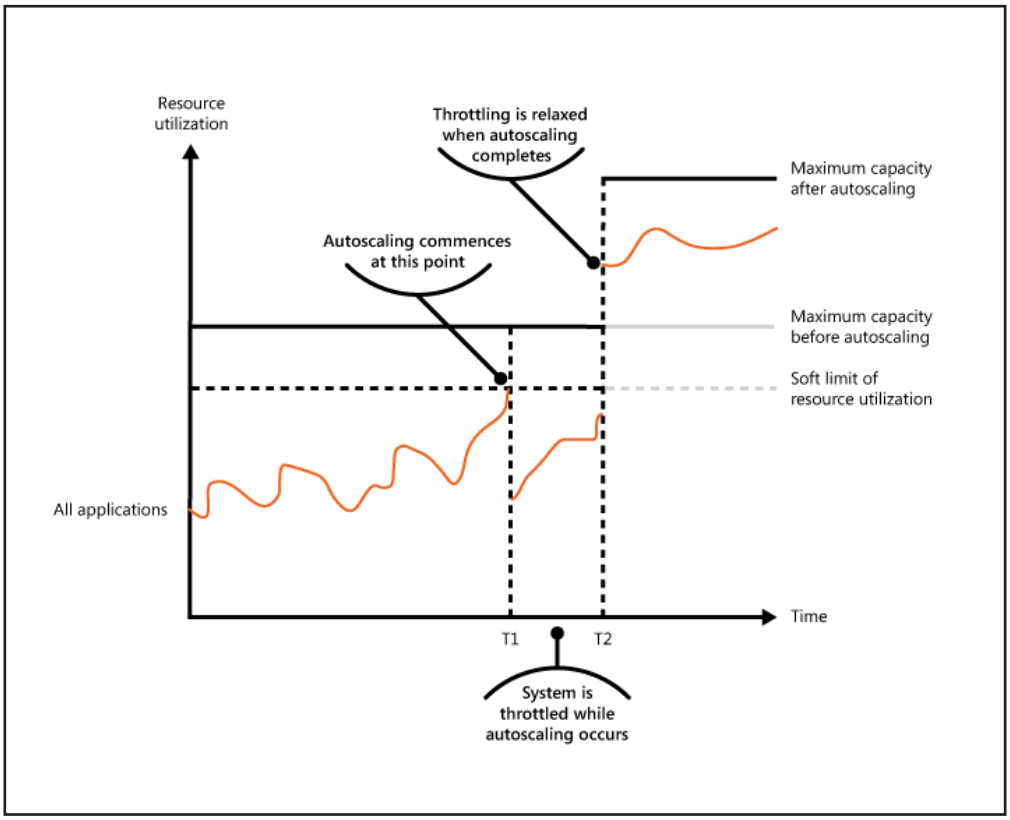


La zone qui se situe juste en dessous de la ligne d'une fonction indique les ressources utilisées par les applications quand elles sollicitent cette fonction. Par exemple, la zone située sous la ligne de la fonction A indique les ressources utilisées par les applications qui sollicitent la fonction A. La zone située entre les lignes de la fonction A et de la fonction B indique les ressources utilisées par les applications qui sollicitent la fonction B. L'addition des zones des différentes fonctions correspond à l'utilisation de ressources totale du système.

La figure ci-dessus illustre les effets du report des opérations. Juste avant le temps T1, le total des ressources attribuées aux différentes applications qui sollicitent ces fonctions atteint un seuil (la limite d'utilisation des ressources). À ce stade, les applications risquent d'épuiser les ressources disponibles. Dans ce système, la fonction B est moins critique que les fonctions A et C. Elle est donc désactivée temporairement et les ressources qu'elle utilisait sont libérées. Entre les temps T1 et T2, les applications qui utilisent les fonctions A et C continuent à s'exécuter comme d'habitude. Enfin, l'utilisation des ressources de ces deux fonctions diminue jusqu'à ce qu'il y ait, au temps T2, une capacité suffisante pour réactiver la fonction B.

Les méthodes de mise à l'échelle automatique et de limitation peuvent également être combinées pour préserver la réactivité des applications et leur conformité aux contrats de niveau de service. Si la demande est censée rester élevée, la limitation constitue une solution temporaire pendant la mise à l'échelle du système. À ce stade, toutes les fonctionnalités du système peuvent être restaurées.

Le graphique en aires ci-dessous illustre l'utilisation globale des ressources par les différentes applications exécutées dans un système par rapport au temps. Il montre également comment la limitation peut être combinée avec la mise à l'échelle automatique.



Au temps T1, le seuil correspondant à la limite logicielle d'utilisation des ressources est atteint. À ce stade, le système peut entamer la mise à l'échelle. Toutefois, si les nouvelles ressources ne sont pas disponibles suffisamment vite, les ressources existantes risquent d'être épuisées et le système risque de ne pas fonctionner correctement. Pour éviter cette situation, le système est limité temporairement, comme décrit plus haut. Lorsque la mise à l'échelle automatique est terminée et que les ressources supplémentaires sont disponibles, la limitation peut être atténuée.

Problèmes et considérations

Prenez en compte les points suivants lorsque vous choisissez le mode d'implémentation de ce patron :

- La limitation d'une application et la stratégie à mettre en œuvre constituent une décision fonctionnelle qui a une incidence sur l'ensemble de l'architecture d'un système. L'ajout d'une étape de limitation étant particulièrement complexe une fois le système mis en œuvre, il doit être envisagé au début du processus de création de l'application.
- La limitation doit s'effectuer rapidement. Le système doit être capable de détecter toute augmentation au niveau de l'activité et de réagir en conséquence. Le système doit aussi pouvoir revenir rapidement à son état initial une fois la charge réduite. Il convient alors que les données de performance appropriées soient capturées et contrôlées en continu.
- Si un service doit temporairement refuser une requête d'un utilisateur, il doit renvoyer un code d'erreur spécifique pour que l'application cliente comprenne que le refus d'exécuter l'opération est dû à une limitation. L'application cliente peut attendre un certain temps avant de réitérer la requête.
- La limitation peut servir de mesure provisoire pendant la mise à l'échelle automatique d'un système. Dans certains cas, il est préférable de simplifier la limitation plutôt que de procéder à une mise à l'échelle en cas de pic d'activité soudain momentané, car la mise à l'échelle peut entraîner des frais de fonctionnement supplémentaires considérables.
- Si la limitation est utilisée comme mesure provisoire pendant la mise à l'échelle automatique d'un système et si les demandes de ressources augmentent très rapidement, le système risque de ne pas pouvoir continuer à fonctionner, même en mode limité. Si une telle situation n'est pas acceptable, pensez à constituer des réserves de capacités plus importantes et à configurer une mise à l'échelle automatique plus agressive.

Quand utiliser ce patron

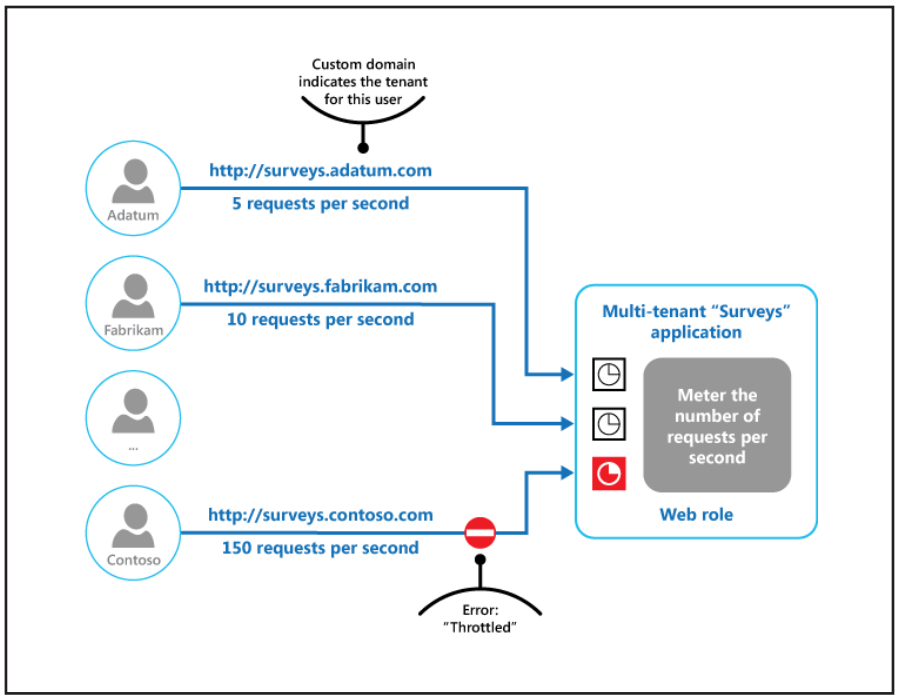
Utilisez ce patron :

- Pour s'assurer qu'un système continue de répondre aux contrats de niveau de service.
- Pour empêcher un client unique de monopoliser les ressources fournies par une application.
- Pour gérer les pics d'activité.
- Pour aider à optimiser les coûts d'un système en limitant le niveau maximal des ressources nécessaires pour maintenir son fonctionnement.

Exemple

La dernière figure montre comment la limitation peut être mise en œuvre dans un système multiclient. Les utilisateurs de chaque organisation cliente accèdent à une application hébergée dans le cloud, où ils remplissent et soumettent des enquêtes. L'application intègre des outils qui surveillent la vitesse à laquelle ces utilisateurs soumettent des requêtes à l'application.

Pour éviter que les utilisateurs d'un client spécifique n'affectent la réactivité et la disponibilité de l'application pour tous les autres utilisateurs, le nombre de requêtes pouvant être soumises par seconde par les utilisateurs d'un même client est limité. L'application bloque les requêtes qui dépassent cette limite.



Patrons et informations connexes

Les patrons et les lignes directrices suivantes peuvent également être pertinents lors de l'implémentation de ce patron :

- Conseils pour l'instrumentation et la télémétrie. La limitation dépend de la collecte d'informations sur le niveau de sollicitation d'un service. Décrit comment générer et de capturer des informations de surveillance personnalisées.
- Conseils pour l'évaluation de l'utilisation des services. Décrit comment mesurer l'utilisation des services afin de mieux comprendre leur utilisation. Ces informations peuvent être utiles pour déterminer comment limiter un service.
- Conseils sur la mise à l'échelle automatique. La limitation peut servir de mesure provisoire pendant la mise à l'échelle automatique d'un système ou pour éliminer la nécessité de mise à l'échelle automatique d'un système. Contient des informations sur les stratégies de mise à l'échelle automatique.
- Patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente). Le nivellement de charge basé sur les files d'attente est une technique couramment utilisée pour la mise en œuvre de limitations. Une file d'attente peut agir comme un tampon qui permet d'uniformiser la vitesse à laquelle les requêtes sont transmises par une application à un service.
- Patron Priority Queue (File d'attente prioritaire). Un système peut utiliser des files d'attentes prioritaires dans le cadre de sa stratégie de limitation afin de maintenir les performances des applications critiques ou de valeur supérieure, tout en réduisant les performances des applications moins importantes.

Patron Valet Key (Clé à accès restreint)

Utilisez un jeton qui fournit aux clients un accès direct restreint à une ressource spécifique pour décharger le transfert de données de l'application. Cette méthode est particulièrement utile dans les applications qui utilisent des files d'attente ou des systèmes de stockage hébergés dans le cloud, et peut réduire les coûts et optimiser l'évolutivité et les performances.

Contexte et problème

Les programmes clients et les navigateurs web doivent souvent lire et écrire des flux de données ou des fichiers dans et depuis l'espace de stockage d'une application. En règle générale, l'application gère la circulation des données, en recherchant celles-ci dans les espaces de stockage et en les transmettant en continu au client, ou en lisant le flux chargé depuis le client et en le stockant dans la banque de données. Cette méthode consomme toutefois des ressources précieuses telles que des ressources de calcul, de la mémoire et de la bande passante.
Les banques de données ont la capacité de gérer le chargement et le téléchargement direct des données sans que l'application ne doive effectuer aucun traitement pour déplacer ces données. Il est toutefois souvent nécessaire que le client ait accès aux informations d'identification de sécurité de la banque. Cette technique peut être utile pour réduire les coûts de transfert des données et la nécessité de faire évoluer l'application, et pour optimiser les performances. Cela signifie cependant que l'application n'est plus en mesure de gérer la sécurité des données. Une fois que le client dispose d'une connexion à la banque de données pour un accès direct, l'application ne peut pas agir comme contrôleur d'accès. Elle échappe au contrôle du processus et ne peut pas empêcher les chargements ou téléchargements ultérieurs depuis la banque de données.

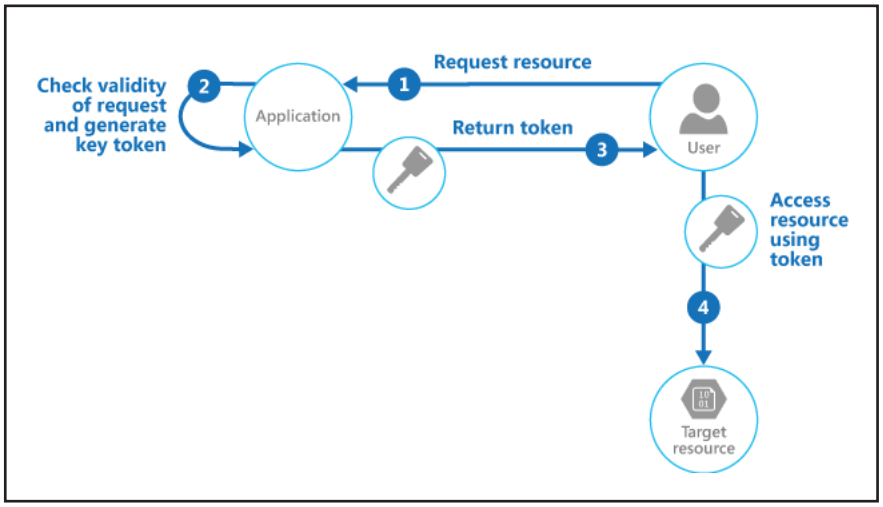
Cette approche n'est pas réaliste dans les systèmes distribués qui doivent servir des clients non fiables. Au lieu de cela, les applications doivent être en mesure de contrôler l'accès aux données de manière précise. Elles doivent également réduire la charge sur le serveur en configurant cette connexion, puis en autorisant le client à communiquer directement avec la banque de données pour exécuter les opérations de lecture ou d'écriture requises.

Solution

Vous devez résoudre le problème du contrôle d'accès à une banque de données qui ne peut pas gérer l'authentification et l'autorisation des clients. Une solution typique consiste à limiter l'accès à la connexion publique de la banque de données, et à fournir au client une clé ou un jeton que la banque de données peut valider.

Cette clé ou ce jeton est généralement appelé « clé à accès restreint ». Il offre un accès limité dans le temps à des ressources spécifiques et permet uniquement les opérations prédéfinies telles que la lecture et l'écriture dans l'espace de stockage ou les files d'attente, ou le chargement et le téléchargement dans un navigateur web. Les applications peuvent créer et générer rapidement et facilement des clés à accès restreint pour les navigateurs Web et les périphériques clients, afin de permettre aux clients d'effectuer les opérations requises sans que l'application ne doive gérer directement le transfert de données. Cette technique élimine de l'application et du serveur la surcharge de traitement et l'impact sur les performances et l'évolutivité.

Le client utilise ce jeton pour accéder à une ressource spécifique dans la banque de données pendant une période spécifique et avec les restrictions spécifiques applicables aux autorisations d'accès, comme illustré sur la figure. Après la période spécifiée, la clé n'est plus valide et ne permet plus d'accéder à la ressource.



Il est également possible de configurer une clé qui a d'autres dépendances, telles que la portée des données. Par exemple, selon les capacités de la banque de données, la clé peut désigner une table complète dans une banque de données ou uniquement des lignes spécifiques dans une table. Dans les systèmes de stockage dans le cloud, la clé peut désigner un conteneur ou uniquement un élément spécifique d'un conteneur.

La clé peut aussi être invalidée par l'application. Cette méthode est utile si le client informe le serveur que l'opération de transfert de données est terminée. Le serveur peut alors invalider cette clé pour éviter toute utilisation ultérieure.

L'utilisation de ce patron peut simplifier la gestion de l'accès aux ressources, car il n'est pas nécessaire de créer et d'authentifier un utilisateur, de lui octroyer des autorisations, puis de supprimer à nouveau l'utilisateur. Cette technique facilite également la limitation de l'emplacement, de l'autorisation et de la durée de validité, tout cela en gérant simplement une clé lors de l'exécution. Il est essentiel de limiter la durée de validité et, surtout, l'emplacement de la ressource, aussi étroitement que possible afin que le destinataire ne puisse l'utiliser qu'aux fins prévues.

Problèmes et considérations

Considérez les points suivants lorsque vous décidez comment implémenter ce patron :

Gestion de l'état et de la durée de validité de la clé. En cas de fuite ou de violation de la sécurité, la clé déverrouille l'élément cible et permet son utilisation à des fins malveillantes pendant la période de validité. Une clé peut généralement être révoquée ou désactivée, selon la façon dont elle a été générée. Les politiques côté serveur peuvent être modifiées ou la clé serveur utilisée pour la signature peut être invalidée. Indiquez une durée de validité courte pour réduire le risque que la banque de données ne fasse l'objet d'opérations non autorisées. Toutefois, si la durée de validité est trop courte, il se peut que le client ne soit pas en mesure d'achever l'opération avant l'expiration de la clé. Donnez aux utilisateurs autorisés la possibilité de renouveler la clé avant l'expiration de la période de validité si plusieurs accès à la ressource protégée sont requis.

Contrôle du niveau d'accès offert par la clé. En règle générale, la clé doit permettre à l'utilisateur d'effectuer uniquement les actions nécessaires pour mener à bien l'opération, par exemple l'accès en lecture seule si le client ne doit pas avoir la possibilité de charger des données dans la banque de données. Pour les chargements de fichiers, il est courant de spécifier une clé qui fournit une autorisation en écriture seule, ainsi que l'emplacement et la durée de validité. Il est essentiel de préciser avec exactitude la ressource ou l'ensemble des ressources auxquelles s'applique la clé.

Détermination du mode de contrôle du comportement des utilisateurs. La mise en œuvre de ce patron implique une certaine perte de contrôle sur les ressources auxquelles les utilisateurs ont le droit d'accéder. Le niveau de contrôle qui peut être exercé est limité par les capacités des politiques et des autorisations disponibles pour le service ou la banque de données cible. Par exemple, il n'est généralement pas possible de créer une clé qui limite la taille des données à écrire dans l'espace de stockage ou le nombre de fois que la clé peut être utilisée pour accéder à un fichier. Cela peut entraîner des coûts imprévus considérables pour le transfert de données, même si cette clé est utilisée par les clients visés. Cette situation peut provenir d'une erreur dans le code qui provoque des chargements ou téléchargements répétitifs. Pour limiter le nombre de fois qu'un fichier peut être chargé, forcez dans la mesure du possible le client à signaler à l'application qu'une opération est terminée. Par exemple, certaines banques de données déclenchent des événements que le code de l'application peut utiliser pour surveiller les opérations et contrôler le comportement des utilisateurs. Il est toutefois difficile d'appliquer des quotas pour des utilisateurs individuels dans un scénario multiclient où la même clé est utilisée par tous les utilisateurs d'un même client.

Validation et désinfection éventuelle de toutes les données chargées. Un utilisateur malveillant qui accède à la clé pourrait charger des données conçues pour compromettre le système. Il est également possible que des utilisateurs autorisés puissent charger des données non valides qui, lors de leur traitement, pourraient entraîner une erreur ou une panne au niveau du système. Pour vous protéger contre ces risques, assurez-vous que toutes les données chargées sont validées et vérifiez avant leur utilisation qu'elles ne comportent pas de contenu malveillant.

Vérification de toutes les opérations. De nombreuses techniques basées sur des clés peuvent consigner des opérations telles que les chargements, téléchargements et échecs. Ces journaux peuvent généralement être intégrés dans un processus de vérification et être utilisés pour la facturation si l'utilisateur est facturé selon le volume des données ou la taille des fichiers. Utilisez ces journaux pour détecter les échecs d'authentification qui pourraient provenir de problèmes au niveau du fournisseur de clé ou de la suppression accidentelle d'une politique d'accès stockée.

Remise de la clé en toute sécurité. La clé peut être intégrée dans une URL que l'utilisateur active sur une page web. Elle peut également être utilisée dans le cadre d'une redirection du serveur pour que le téléchargement s'effectue automatiquement. Utilisez toujours le protocole HTTPS pour diffuser la clé sur un canal sécurisé.
Protégez les données sensibles pendant le transfert. Les données sensibles transmises par le biais de l'application sont généralement acheminées à l'aide du protocole SSL ou TLS. Cette méthode doit être appliquée pour les clients qui accèdent directement à la banque de données.

Autres problèmes à prendre en compte lors de la mise en œuvre de ce patron :

- Si le client ne signale pas ou ne peut pas signaler au serveur la fin de l'opération, et si la seule limite est la période d'expiration de la clé, l'application ne sera pas capable d'effectuer des opérations de contrôle telles que la comptabilisation des chargements ou téléchargements ou d'empêcher les chargements ou téléchargements multiples.
- La flexibilité des politiques basées sur des clés pouvant être générées peut être limitée. Par exemple, certains mécanismes permettent uniquement l'utilisation d'une période d'expiration temporisée. D'autres ne peuvent pas spécifier une granularité suffisante des autorisations de lecture/écriture.
- Si l'heure de début de la période de validité du jeton ou de la clé est spécifiée, veillez à ce qu'elle soit légèrement antérieure à l'heure actuelle du serveur afin d'accepter les horloges des clients qui pourraient être légèrement désynchronisées. Lorsqu'il n'est pas défini, ce paramètre est généralement réglé par défaut sur l'heure actuelle du serveur.
- L'URL qui contient la clé est enregistrée dans les fichiers journaux du serveur. Tandis que la clé aura généralement expiré avant l'utilisation des fichiers journaux pour l'analyse, veillez à limiter l'accès à ces journaux. Si les données des journaux sont transmises à un système de surveillance ou sont stockées à un autre emplacement, pensez à paramétrer un décalage pour éviter les fuites de clés jusqu'à l'expiration de leur période de validité.
- Si le code client s'exécute dans un navigateur web, le navigateur devra peut-être prendre en charge le partage de ressources d'origine croisée (CORS) afin d'activer le code qui s'exécute dans le navigateur web pour accéder aux données dans un autre domaine que celui qui a diffusé la page. Certains anciens navigateurs et certaines banques de données ne prennent pas en charge les CORS. Le code qui s'exécute dans ces navigateurs peut être en mesure d'utiliser une clé à accès restreint pour fournir l'accès aux données dans un domaine différent, par exemple un compte de stockage dans le cloud.

Quand utiliser ce patron

Ce patron est utile dans les situations suivantes :

- Pour réduire le chargement de ressources et optimiser les performances et l'évolutivité. L'utilisation d'une clé à accès restreint ne nécessite pas le verrouillage de la ressource, ni aucun appel de serveur distant, ni aucune limite sur le nombre de clés à accès restreint pouvant être générées. Elle évite également un point de défaillance unique résultant du transfert de données par le biais du code de l'application. La création d'une clé à accès restreint est généralement une opération cryptographique simple de signature d'une chaîne avec une clé.
- Pour réduire les coûts d'exploitation. Permettre un accès direct aux banques de données et aux files d'attente est efficace en termes de ressources et de coûts, peut réduire le nombre d'allers-retours sur le réseau et peut diminuer le nombre de ressources de calcul nécessaires.
- Lorsque les clients chargent ou téléchargent régulièrement des données, notamment si le volume est important ou si chaque opération implique des fichiers volumineux.

- Lorsque les ressources de calcul disponibles de l'application sont limitées, en raison de limitations d'hébergement ou pour des considérations liées aux coûts. Dans ce scénario, le patron est encore plus utile si le nombre de chargements ou téléchargements simultanés est élevé, car il évite à l'application de gérer le transfert de données.

- Lorsque les données sont stockées dans une banque de données distante ou dans un autre centre de données. Si l'application était tenue d'agir comme contrôleur d'accès, il pourrait y avoir des frais pour la bande passante supplémentaire nécessaire au transfert des données entre les centres de données ou sur les réseaux publics ou privés entre le client et l'application, puis entre l'application et la banque de données.

Ce patron n'est pas nécessairement utile dans les situations suivantes :

- Si l'application doit exécuter des tâches sur les données avant leur stockage ou leur transmission au client. Par exemple, si l'application doit effectuer une validation, consigner le succès des accès ou soumettre les données à une transformation. Cependant, certaines banques de données et clients sont en mesure de négocier et de réaliser des transformations simples, telles que la compression et la décompression (par exemple, un navigateur web peut généralement gérer les formats GZip).

- Si la conception d'une application existante rend difficile l'intégration du patron. L'utilisation de ce patron requiert généralement une approche architecturale différente pour la transmission et la réception de données.

Exemple

Azure prend en charge les signatures d'accès partagé sur Azure Storage pour le contrôle d'accès granulaire sur les données contenues dans des objets blob, des tables et des files d'attente, et pour les rubriques et files d'attente Service Bus. Un jeton de signature d'accès partagé peut être configuré pour fournir des droits d'accès spécifiques tels que la lecture, l'écriture, la mise à jour et la suppression sur une table spécifique, une place de clés dans une table, une file d'attente, un objet blob ou un conteneur d'objets blob. La validité peut correspondre à une période spécifique ou à aucune limite temporelle.

Les signatures d'accès partagé Azure prennent également en charge les stratégies d'accès stockées sur des serveurs qui peuvent être associées à une ressource spécifique comme une table ou un objet blob. Cette fonctionnalité offre plus de contrôle et de flexibilité par rapport aux jetons de signature d'accès partagé générés par l'application et doit être utilisée chaque fois que possible. Les paramètres définis dans une stratégie stockée sur un serveur peuvent être modifiés et sont reflétés dans le jeton sans nécessiter l'émission d'un nouveau jeton, alors que les paramètres définis dans le jeton ne peuvent pas être modifiés sans émettre un nouveau jeton. Cette méthode permet aussi de révoquer un jeton de signature d'accès partagé valide avant son expiration.

Pour de plus amples informations, consultez les articles Présentation des SAS (Shared Access Signature) de table et de file d'attente et mise à jour du SAS d'objet blob et Utilisation de signatures d'accès partagé sur MSDN.

Le code ci-dessous montre comment créer un jeton de signature d'accès partagé valide pendant cinq minutes. La méthode « GetSharedAccessReferenceForUpload » renvoie un jeton de signature d'accès partagé qui peut être utilisé pour charger un fichier sur Azure Blob Storage.

```
public class ValuesController : ApiController
{
  private readonly CloudStorageAccount account;
  private readonly string blobContainer;
  ...
  /// <summary>
  /// Renvoyer une clé d'accès limité qui permet à l'appelant de charger un fichier
  /// vers cette destination spécifique pendant une période définie.
  /// </summary>
  private StorageEntitySas GetSharedAccessReferenceForUpload(string blobName)
  {
    var blobClient = this.account.CreateCloudBlobClient();
    var container = blobClient.GetContainerReference(this.blobContainer);

    var blob = container.GetBlockBlobReference(blobName);

    var policy = new SharedAccessBlobPolicy
    {
      Permissions = SharedAccessBlobPermissions.Write,

      // Indiquer une heure de début cinq minutes plus tôt pour permettre un décalage de
         l'horloge du client.
      SharedAccessStartTime = DateTime.UtcNow.AddMinutes(-5),

      // Indiquer une durée de validité de cinq minutes à partir de maintenant.
      SharedAccessExpiryTime = DateTime.UtcNow.AddMinutes(5)
    };

    // Créer la signature.
    var sas = blob.GetSharedAccessSignature(policy);

    return new StorageEntitySas
    {
      BlobUri = blob.Uri,
      Credentials = sas,
      Name = blobName
    };
  }

  public struct StorageEntitySas
  {
    public string Credentials;
    public Uri BlobUri;
    public string Name;
  }
}
```

L'exemple intégral est disponible dans la solution ValetKey accessible en téléchargement sur GitHub. Le projet ValetKey.Web dans cette solution contient une application web qui inclut la classe ValuesController illustrée ci-dessus. Le projet ValetKey.Client contient un exemple d'application cliente qui utilise cette application web pour récupérer une clé de signature d'accès partagé et charger un fichier dans l'espace de stockage blob.

Étapes suivantes

Les patrons et les lignes directrices suivantes peuvent également être pertinents lors de l'implémentation de ce patron :

- Un exemple illustrant ce patron est disponible sur GitHub.

- Patron Gatekeeper (Contrôleur d'accès). Ce patron peut être utilisé conjointement avec le patron Valet Key (Clé à accès restreint) pour protéger les applications et les services à l'aide d'une instance dédiée de l'hôte qui agit comme un intermédiaire entre les clients et l'application ou le service. Le contrôleur d'accès valide et assainit les requêtes et transmet celles-ci ainsi que les données entre le client et l'application. Il peut fournir une couche de sécurité supplémentaire et réduire la surface d'attaque du système.

- Patron Static Content Hosting (Hébergement de contenu statique). Décrit comment déployer des ressources statiques sur un service de stockage dans le cloud capable de fournir ces ressources directement au client afin de réduire les exigences en termes d'instances de calcul coûteuses. Lorsque les ressources ne sont pas conçues pour être accessibles au public, le patron Valet Key (Clé à accès restreint) peut permettre leur sécurisation.

- Présentation des SAS (Shared Access Signature) de table et de file d'attente et mise à jour du SAS d'objet blob

- Utilisation de signatures d'accès partagé

- Authentification des signatures d'accès partagé avec Service Bus

7

Listes de vérification de la conception

Liste de vérification DevOps

DevOps est l'intégration du développement, de l'assurance qualité et des opérations informatiques dans une culture unifiée et l'ensemble des processus de distribution de logiciels.

Utilisez cette liste de vérification comme point de départ pour évaluer votre culture et votre processus DevOps.

Culture

☐ **Veiller à l'harmonisation commerciale entre les organisations et les équipes**

- Veillez à ce que les équipes commerciales et les équipes en charge du développement et des opérations soient toutes en phase.

☐ **S'assurer que l'ensemble de l'équipe comprend le cycle de vie des logiciels**

- Assurez-vous que votre équipe comprend le cycle de vie de l'application et quelle partie de ce cycle de vie est actuellement en cours.

☐ **Réduire la durée des cycles**

- Réduisez le temps nécessaire pour passer des idées aux logiciels développés utilisables.
- Limitez la taille et la portée des publications individuelles pour maintenir la charge des tests à un niveau peu élevé.
- Automatisez les processus de construction, de test, de configuration et de déploiement chaque fois que possible.
- Éliminez tous les obstacles à la communication entre les développeurs et les autres membres du personnel.

☐ **Réviser et améliorer les processus.**

- Mettez en place des contrôles réguliers des flux de production, des procédures et de la documentation actuels dans une perspective d'amélioration continue.

☐ **Établir une planification proactive.**

- Prévoyez les pannes de manière proactive.
- Mettez en place des processus pour identifier rapidement les problèmes lorsqu'ils se produisent.
- Transmettez les problèmes aux membres compétents de l'équipe pour les résoudre et confirmer leur résolution.

☐ **Apprendre des échecs**

- En cas de défaillance opérationnelle, identifiez le problème, documentez sa cause et sa solution et partagez tous les enseignements tirés.
- Mettez à jour vos processus de construction de sorte qu'ils détectent automatiquement ce genre de défaillance à l'avenir.

☐ **Optimiser la vitesse et la collecte des données**

- Utilisez les plus petits incréments possibles.
- Traitez les nouvelles idées comme des expérimentations.
- Exploitez les expérimentations afin de pouvoir recueillir des données de production pour évaluer leur efficacité.
- Préparez-vous à un échec rapide si l'hypothèse est erronée.

☐ **Prévoir le temps d'apprentissage**

- Avant de passer à de nouveaux projets, prévoyez suffisamment de temps pour tirer les leçons importantes et vous assurer que votre équipe les assimile correctement. Donnez à l'équipe le temps de développer des compétences, d'expérimenter et de se familiariser avec les nouveaux outils et techniques.

☐ **Documenter les opérations**

- Documentez tous les outils, processus et tâches automatisées avec le même niveau de qualité que votre code produit.
- Documentez la conception et l'architecture actuelles de tous les systèmes pris en charge, ainsi que les processus de récupération et les autres procédures de maintenance.
- Mettez l'accent sur les étapes que vous effectuez réellement et non sur les processus optimaux en théorie.
  Révisez et mettez à jour régulièrement la documentation.
- Concernant le code, pensez à inclure des commentaires significatifs, en particulier dans les API publiques, et à utiliser des outils permettant de générer automatiquement la documentation sur le code.

☐ **Partager les connaissances**

- Assurez-vous que la documentation est organisée et facilement détectable.
- Utilisez des réunions casse-croûte (présentations informelles), des vidéos ou des bulletins d'information pour partager les connaissances.

Développement

☐ **Fournir aux développeurs des environnements similaires à l'environnement de production**

- Maintenez les environnements de développement et de test aussi proches que possible de l'environnement de production.
- Assurez-vous que les données de test sont compatibles avec les données utilisées dans la production, même s'il s'agit d'exemples de données et non de données de production réelles (pour des raisons de confidentialité ou de conformité).
- Prévoyez la génération et l'anonymisation des exemples de données de test.

☐ **S'assurer que tous les membres autorisés de l'équipe peuvent alimenter l'infrastructure et déployer l'application**

- Toute personne possédant les autorisations appropriées devrait être en mesure de créer ou de déployer des ressources similaires aux ressources de production sans s'adresser à l'équipe en charge des opérations.

☐ **Exploiter l'application pour obtenir des renseignements**

- Incluez toujours les outils comme une exigence de la conception et intégrez-les dans l'application dès le début.
- Les outils doivent inclure la journalisation des événements pour l'analyse des causes profondes, la télémétrie et la métrologie pour surveiller l'état général et l'utilisation de l'application.

☐ **Suivre votre dette technique**

- Faites le suivi des moments où les cycles de lancement peuvent obtenir la priorité sur la qualité du code.
- Documentez toute lacune ou mise en œuvre qui ne se révèle pas optimale et prévoyez du temps plus tard pour vérifier ces problèmes.

☐ **Envisager de pousser les mises à jour directement à la production**

- Pour réduire la durée globale du cycle de lancement, pensez à pousser les codes testés adéquatement directement à la production.
- Utilisez des fonctions à bascule pour gérer les fonctions qui sont actives.

Test

☐ **Automatiser les tests**

- Automatisez les tests courants et intégrez-les à vos processus de conception.
- Des tests intégrés de l'interface utilisateur devraient également être effectués par un outil automatique.

☐ **Effectuer des tests pour identifier les défaillances**

- Exécutez toujours des tests par injection de défaillance pendant la révision des environnements d'essai et de simulation.
- Lorsque vos pratiques et processus de test sont au point, pensez à les exécuter dans l'environnement de production.

☐ **Effectuer des tests dans l'environnement de production**

- Prévoyez des tests pour vous assurer que le code déployé fonctionne comme prévu.
- Dans le cas des déploiements qui ne sont pas souvent mis à jour, planifiez dans le cadre de la maintenance des tests dans l'environnement de production.

☐ **Automatiser les tests de performance pour repérer rapidement les problèmes de performance**

- Définissez des objectifs de performance acceptables pour des mesures comme la latence, les temps de chargement et l'utilisation des ressources.
- Intégrez des tests de performance automatisés à votre canal de publication pour vous assurer que l'application satisfait à ces objectifs.

☐ **Effectuer des tests de capacité**

- Définissez toujours la capacité prévue et les limites d'utilisation maximales.
- Effectuez des tests pour confirmer que l'application peut gérer ces limites, mais également pour voir ce qui survient en cas de dépassement de ces limites.
- Testez la capacité à intervalles réguliers.
- Après la publication initiale, effectuez des tests de performance et de capacité lorsque le code de production est mis à jour.
- Servez-vous des données historiques pour mettre les tests au point et déterminer les types de tests qu'il faut exécuter.

☐ **Effectuer des tests d'intrusion automatisés**

- Intégrez toujours des tests d'intrusion automatisés dans le cadre du processus de création et de développement.
- Planifiez régulièrement des tests de sécurité et d'analyse des vulnérabilités sur les applications déployées en vérifiant si des ports sont ouverts, les points de terminaison et les attaques.

☐ **Effectuer des tests automatisés de continuité des activités**

- Développer des tests pour la continuité des activités à grande échelle, y compris la récupération et le basculement.
- Configurez des processus automatisés pour exécuter régulièrement ces tests.

Publication

☐ **Automatiser les déploiements**

- Automatisez le déploiement de l'application pour tester les environnements de simulation et de production.

☐ **Utiliser une intégration continue**

- Fusionnez régulièrement tous les codes des développeurs dans une base de données centrale, puis exécutez automatiquement les processus standard de conception et de test.
- Exécutez le processus d'intégration continue chaque fois qu'un code est validé ou vérifié et au moins une fois par jour.
  Pensez à adopter un modèle de développement basé sur le tronc.

☐ **Envisager la distribution continue**

- Veillez à ce que le code soit toujours prêt à être déployé en concevant, testant et déployant automatiquement le code dans des environnements comparables à un environnement de production.
- Le déploiement continu est un processus supplémentaire qui récupère automatiquement les mises à jour acheminées par le canal d'intégration continue et de distribution continue, puis le déploie dans l'environnement de production.
- Il convient de planifier efficacement les tests automatisés et les processus évolués.

☐ **Apporter de modestes changements progressifs**

- Apportez des changements modestes.

☐ **Gérer l'exposition aux changements**

- Utilisez des fonctions à bascule pour gérer l'activation des fonctions pour les utilisateurs finaux.

☐ **Mettre en œuvre des stratégies de gestion des publications pour réduire les risques liés au déploiement**

- Adoptez des stratégies comme le déploiement de canari ou le déploiement bleu-vert pour déployer les mises à jour à un sous-ensemble d'utilisateurs.
- Vérifiez que la mise à jour fonctionne comme prévu, puis déployez-la sur le reste du système.

☐ **Documentez toutes les modifications**

- Gardez toujours un historique clair des modifications, aussi minime soient-elles.
- Consignez tous les changements dans un journal, notamment les correctifs appliqués, les modifications de stratégies et les modifications de la configuration.
- N'incluez pas de données sensibles dans ces journaux. Par exemple, consignez dans le journal qu'une information d'identification a été mise à jour et indiquez l'auteur de la modification, mais n'enregistrez pas les informations d'identification mises à jour.

☐ **Automatiser les déploiements**

- Automatisez tous les déploiements et mettez en place des systèmes pour détecter tout problème au cours du déploiement.
- Utilisez une procédure d'atténuation pour préserver le code existant et les données en production, avant que la mise à jour ne les remplace dans toutes les instances de production.
- Automatisez la restauration par progression des correctifs et l'annulation des modifications.

☐ **Envisagez de rendre votre infrastructure immuable**

- Vous ne devez pas modifier l'infrastructure après son déploiement en production.

Pilotage

☐ **Rendez vos systèmes observables**

- Prévoyez des points de terminaison externes d'intégrité pour surveiller le statut et garantir que les applications sont codées pour instrumenter les métriques des opérations.
- Utilisez un schéma commun et cohérent qui vous permet de mettre en corrélation des événements entre les systèmes.

☐ **Regroupez et mettez en corrélation les journaux et les métriques**

- Assurez-vous que les données de télémétrie et de journaux sont traitées et mises en corrélation dans un court laps de temps, de manière que le personnel d'exploitation dispose toujours d'une image à jour de l'intégrité du système.
- Organisez et affichez les données de façon à obtenir une vue cohérente de tous les problèmes et, dans la mesure du possible, à distinguer clairement la corrélation entre les événements.
- Consultez votre stratégie de rétention d'entreprise pour connaître les exigences relatives au traitement des données et à leur durée de stockage.

☐ **Implémentez des notifications et des alertes automatiques**

- Mettez en place des outils de pilotage, tels que Azure Monitor, pour détecter les patrons ou les conditions indiquant des problèmes actuels ou potentiels et envoyer des alertes aux membres de votre équipe disposant des compétences pour résoudre ces problèmes.
- Réglez les alertes pour éviter les faux positifs.

☐ **Surveillez les dates d'expiration de vos ressources**

- Assurez-vous d'inclure dans le suivi les noms des ressources concernées, leurs dates d'expiration, ainsi que les services et fonctionnalités dépendant de ces ressources.
- Automatisez le pilotage de ces ressources.
- Envoyez une notification à l'équipe des opérations avant l'expiration d'une ressource et procédez à un transfert en escalade si l'expiration menace de perturber l'application.

Gestion

☐ **Automatisez les tâches**

- Automatisez autant que possible les processus opérationnels répétitifs pour garantir une qualité et une exécution homogènes.
- Les versions du code qui implémente l'automatisation doivent être gérées dans le contrôle de code source.
  Comme avec tout autre code, les outils d'automatisation doivent être testés.

☐ **Traitez l'infrastructure sous forme de code dans votre approche de la configuration**

- Utilisez des scripts et des modèles Azure Resource Manager.
- Conservez les scripts et les modèles dans le contrôle de code source, comme tout autre code que vous gérez.

☐ **Envisagez d'utiliser des conteneurs**

☐ **Implémentez la résilience et l'auto-régénération**

- Instrumentez vos applications de façon que les problèmes soient signalés immédiatement et que vous puissiez gérer les pannes ou autres défaillances du système.

☐ **Utilisez un manuel des opérations**

- Prévoyez la tenue d'un manuel des opérations ou d'un runbook pour documenter les procédures et les informations de gestion dont le personnel d'exploitation a besoin pour assurer la maintenance d'un système.
- Documentez tous les scénarios d'opérations et les plans d'atténuation susceptibles d'entrer en jeu en cas de défaillance ou autre perturbation de votre service.
- Créez cette documentation pendant le processus développement et tenez-la à jour par la suite.
- Examinez, testez et améliorer régulièrement.

Encouragez les membres de votre équipe à contribuer et à partager leurs connaissances.

- Faites en sorte que chaque membre de l'équipe puisse contribuer facilement à la mise à jour des documents.

☐ **Documentez les procédures d'astreinte**

- Assurez-vous que les tâches, calendriers et procédures d'astreinte sont documentés et partagés entre tous les membres de l'équipe.
- Tenez ces informations à jour en permanence.

☐ **Documentez les procédures d'escalade pour les dépendances tierces**

- Définissez un plan pour traiter les pannes si vous faites appel à des services tiers.
- Créez la documentation pour vos processus d'atténuation planifiés.
- Incluez les contacts de support et les chemins de réaffectation.

☐ **Utilisez la gestion de la configuration**

- Planifiez et enregistrez vos modifications de configuration.
- Réalisez régulièrement des audits pour vous assurer que le niveau de service attendu est effectivement en place.

☐ **Bénéficiez d'un plan de support Azure et comprenez le processus**

- Déterminez le plan adapté à vos besoins et assurez-vous que toute l'équipe sait comment l'utiliser.
- Les membres de l'équipe doivent comprendre les détails du plan, connaître le fonctionnement du processus d'assistance et savoir ouvrir un ticket de support avec Azure.
- Si vous prévoyez un événement à grande échelle, le support Azure vous aide à augmenter les limites de votre service.

☐ **Suivez les principes des privilèges minimaux lors de l'octroi de l'accès aux ressources.**

- Gérez attentivement l'accès aux ressources.
- Octroyez uniquement aux utilisateurs l'accès aux ressources dont ils ont besoin pour effectuer leurs tâches.

☐ **Utilisez le contrôle d'accès en fonction du rôle.**

- Utilisez le contrôle d'accès en fonction du rôle (RBAC) pour octroyer l'accès en fonction des identités et des groupes Azure Active Directory.

☐ **Utilisez un système de suivi des bogues pour suivre les problèmes.**

- Utilisez un outil de suivi des bogues pour enregistrer les détails des problèmes, affecter des ressources pour y remédier et fournir une piste d'audit de la progression et de l'état.

☐ **Gérez toutes les ressources dans un système de gestion des changements.**

- Traitez tous ces types de ressources sous forme de code tout au long du processus de test/conception/révision.

☐ **Utilisez les listes de contrôle.**

- Gérez les listes de contrôle et recherchez en permanence des moyens d'automatiser les tâches et de rationaliser les processus.

Liste de vérification de la disponibilité

La disponibilité définit la proportion de temps durant laquelle le système est fonctionnel et en fonctionnement. Passez en revue les éléments de cette liste de contrôle pour améliorer la disponibilité de votre application.

☐ **Évitez tout point de défaillance unique**

- Déployez tous les composants, services, ressources et instances de calcul sur plusieurs instances.
- Concevez l'application de façon qu'elle soit configurable pour l'utilisation de plusieurs instances.
- Concevez l'application de manière qu'elle détecte automatiquement les échecs et redirige les demandes vers des instances qui ne sont pas en échec.

☐ **Décomposez le scénario d'usage par contrat de niveau de service**

- Gérez différemment les scénarios d'usage critiques et moins critiques.
- Spécifiez les fonctionnalités du service et le nombre d'instances pour répondre à leurs exigences de disponibilité.

☐ **Réduisez et comprenez les dépendances de services**

- Réduisez le nombre de services différents utilisés, dans la mesure du possible.
- Comprenez les dépendances et l'impact d'une défaillance ou de la diminution des performances dans chacun des services sur l'application.

☐ **Concevez des tâches et des messages idempotents (répétables en toute sécurité)**

- Rendez idempotents les consommateurs de messages et les opérations qu'ils effectuent.
- Détectez les messages dupliqués ou utilisez une approche optimiste pour traiter les conflits.

☐ **Utilisez un courtier de messagerie hautement disponible pour les transactions critiques**

- Utilisez une messagerie asynchrone dans laquelle l'expéditeur continue le traitement sans attendre la réponse.
- Utilisez un système de messagerie qui offre une haute disponibilité et garantit la sémantique « at-least-once » (au moins une fois).
- Accordez de l'importance au traitement des messages (voir article précédent).

☐ **Concevez des applications capables de passer en douceur au mode dégradé**

- Concevez l'application de façon qu'elle puisse passer automatiquement et en douceur au mode dégradé.
- Lorsque les limites de ressources sont atteintes, prenez les mesures appropriées pour minimiser l'impact de la disponibilité réduite et des échecs de connexion pour l'utilisateur.
- Reportez les demandes à acheminer vers un sous-système en échec, dans la mesure du possible.

☐ **Gérez sans heurts les événements en rafale rapide**

- Concevez des applications permettant de gérer différents scénarios d'usage, tels que les pics de début de matinée ou lors du lancement d'un nouveau produit sur un site de commerce électronique.
- Utilisez la mise à l'échelle automatique, dans la mesure du possible.
- Mettez en file d'attente les demandes des services et passez en douceur au mode dégradé lorsque les files d'attente sont proches de la pleine capacité.
- Assurez-vous que les performances et la capacité disponibles dans des conditions de non-rafale sont suffisantes pour vider les files d'attente et traiter les demandes en suspens. Pour de plus amples informations, consultez le Patron Queue-Based Load Leveling (Nivellement de charge basé sur la file d'attente).

Déploiement et maintenance

☐ **Déployez plusieurs instances de rôles pour chaque service**

- Déployez au moins deux instances de chaque rôle sur le service. Ainsi, un rôle peut être indisponible pendant que l'autre reste actif.

☐ **Hébergez des applications dans plusieurs régions**

- Hébergez vos applications métier vitales dans plusieurs régions pour assurer une disponibilité maximale.

☐ **Automatisez et testez les tâches de déploiement et de maintenance**

- Automatisez le déploiement à l'aide de mécanismes testés et éprouvés, tels que les scripts
- et les modèles Resource Manager.
- Automatisez toutes les mises à jour de vos applications.
- Testez entièrement vos processus automatisés pour vous assurer qu'ils ne contiennent aucune erreur.
- Utilisez des restrictions de sécurité sur les outils d'automatisation.
- Définissez soigneusement et mettez en œuvre des stratégies de déploiement.

☐ **Envisagez d'utiliser les fonctionnalités de gestion intermédiaire et de production de la plateforme**

- Azure App Service prend en charge le basculement entre l'environnement de tests ou de préproduction et l'environnement de production, sans temps d'arrêt des applications.
- Si vous préférez stade local, ou déployez simultanément les différentes versions de l'application et progressivement migrez les utilisateurs, vous ne serez peut-être pas en mesure d'utiliser un swap de VIP.

☐ **Appliquez les modifications de configuration sans recyclage**

- Les paramètres de configuration d'une application ou d'un service Azure peuvent être modifiés sans qu'un redémarrage du rôle ne soit nécessaire.
- Concevez une application qui accepte les modifications apportées aux paramètres de configuration sans qu'un redémarrage de l'application entière ne soit nécessaire.

☐ **Utilisez les domaines de mise à niveau pour garantir des mises à jour sans aucune interruption de service**

- Indiquez le nombre de domaines de mise à niveau à créer pour un service lors du déploiement du service.

☐ **Remarque**

Les rôles sont aussi répartis sur des domaines d'erreur, dont chacun est raisonnablement indépendant des autres domaines d'erreur en termes de rack de serveur, d'alimentation et de système de refroidissement, afin de minimiser le risque qu'une défaillance affecte toutes les instances de rôle. Cette distribution s'effectue automatiquement et vous ne pouvez pas la contrôler.

☐ **Configurez des groupes à haute disponibilité pour les machines virtuelles Azure**

- Placez au moins deux machines virtuelles dans le même groupe à haute disponibilité pour garantir qu'elles ne seront pas déployées sur le même domaine d'erreur.
- Pour maximiser la disponibilité, créez plusieurs instances de chaque machine virtuelle critique utilisée par votre système, puis placez ces instances dans le même groupe à haute disponibilité.
- Si vous exécutez plusieurs machines virtuelles ayant chacune un usage spécifique, créez un groupe à haute disponibilité pour chaque machine virtuelle.
- Ajoutez des instances de chaque machine virtuelle à chaque groupe à haute disponibilité.

Gestion des données

☐ **Géo-répliquez vos données dans Azure Storage**

- Utilisez Read-Access Geographically Redundant Storage (RA-GRS) pour une plus grande disponibilité.

☐ **Géo-répliquez vos bases de données**

- Utilisez Azure SQL Database et Cosmos DB pour la prise en charge de la géo-réplication.
- Configurez des répliques de bases de données secondaires dans d'autres régions.
- Si une panne régionale se produit ou si vous ne parvenez pas à vous connecter à la base de données primaire, basculez vers le réplica secondaire.

  Pour de plus amples informations, consultez Comment distribuer des données mondialement avec Azure Cosmos DB.

☐ **Utilisez l'accès concurrentiel optimiste et la cohérence éventuelle**

- Utilisez le partitionnement pour minimiser les risques de mises à jour conflictuelles.

☐ **Utilisez la sauvegarde périodique et la limite de restauration dans le temps**

- Assurez-vous que la sauvegarde et la restauration répondent à votre objectif de point de récupération (RPO).
- Sauvegardez régulièrement et automatiquement les données qui ne sont pas conservées ailleurs.
- Vérifiez que vous pouvez restaurer de manière fiable les données et l'application elle-même cas de défaillance.
- Sécurisez votre processus de sauvegarde pour protéger les données en transit et en stockage.

☐ **Activez l'option de haute disponibilité pour conserver une copie secondaire d'un cache Redis Azure**

- Lors de l'utilisation de Cache Redis Azure, choisissez le niveau Standard ou Premium pour conserver une copie secondaire du contenu. Pour de plus amples informations, consultez Create a cache in Azure Redis Cache (Créez un cache dans Cache Redis Azur).

Erreurs et échecs

☐ **Introduisez le concept de délai d'attente**

- Assurez-vous que les délais d'attente que vous appliquez sont appropriés à chaque service ou ressource ainsi qu'au client qui y accède.
- Il peut s'avérer opportun d'autoriser un délai d'attente plus long pour une instance particulière d'un client, selon le contexte et les autres actions effectuées par le client.

☐ **Recommencez les opérations ayant échoué en raison de défaillances transitoires**

- Concevez une stratégie de nouvelle tentative pour l'accès à tous les services et toutes les ressources qui ne prennent pas en charge intrinsèquement la nouvelle tentative automatique de connexion.
- Utilisez une stratégie qui inclut un délai croissant entre les tentatives à mesure que le nombre d'échecs augmente.

☐ **Arrêtez d'envoyer des demandes pour éviter les échecs en cascade**

- Au lieu de recommencer sans cesse une opération qui a peu de chances de réussir, l'application doit rapidement accepter que l'opération a échoué et gérer sans heurts cet échec.
- Vous pouvez utiliser le patron Circuit Breaker (Disjoncteur) pour rejeter les demandes d'opérations spécifiques sur des périodes définies. Pour plus d'informations, consultez Patron Circuit Breaker (Disjoncteur).

☐ **Composez ou utilisez plusieurs composants**

- Si possible, concevez vos applications de manière à pouvoir tirer parti de plusieurs instances sans perturber le fonctionnement ni affecter les connexions existantes.
- Distribuez les demandes entre plusieurs instances, détectez les instances en échec et évitez de leur envoyer les demandes. Vous maximiserez la disponibilité.

☐ **Basculez vers un autre service ou workflow lorsque cela est possible**

- Mettez à disposition une installation permettant de relire les écritures du stockage d'objets blob dans SQL Database lorsque le service sera disponible.
- Détectez les échecs et redirigez les demandes vers d'autres services susceptibles d'offrir des fonctionnalités de substitution, ou vers des instances de sauvegarde capables de maintenir les opérations essentielles pendant que le service principal est hors ligne.

Pilotage et récupération d'urgence

☐ **Prévoyez une instrumentation riche pour les défaillances probables et les événements d'échec**

- Pour les défaillances qui surviendront probablement mais qui n'ont pas encore eu lieu, fournissez suffisamment de données au personnel d'exploitation pour lui permettre de déterminer la cause, d'atténuer la situation et de garantir la disponibilité du système.
- Pour les défaillances qui ont déjà eu lieu, l'application doit renvoyer un message d'erreur à l'utilisateur mais tenter de continuer à s'exécuter avec des fonctionnalités réduites.
- Dans tous les cas, le système de pilotage doit capturer les détails complets pour pouvoir activer la récupération rapide et modifier le système de manière à éviter que la situation ne se reproduise.

☐ **Surveillez l'intégrité du système en implémentant des fonctions de contrôle**

- Implémentez des sondes et des fonctions de contrôle exécutées régulièrement depuis l'extérieur de l'application.

☐ **Testez régulièrement tous les systèmes de basculement et de secours**

- Testez les systèmes de basculement et de secours avant qu'ils ne soient requis pour pallier un problème réel lors de l'exécution.

☐ **Testez les systèmes de pilotage**

- Assurez-vous que le pilotage et l'instrumentation fonctionnent correctement.

☐ **Suivez la progression des workflows de longue durée et effectuez une nouvelle tentative en cas d'échec**

- Assurez-vous que chaque étape est indépendante et peut faire l'objet d'une nouvelle tentative.
- Surveillez et gérez la progression des workflows de longue durée en implémentant un patron tel que le Patron Scheduler Agent Supervisor (Planificateur-agent-superviseur).

☐ **Planifiez la récupération d'urgence**

- Créez un plan accepté, entièrement testé, qui assure la récupération après tout type de défaillance susceptible d'affecter la disponibilité du système.
- Choisissez une architecture de récupération d'urgence multisite pour toutes les applications critiques.
- Identifiez un propriétaire spécifique pour le plan de récupération d'urgence, notamment pour l'automatisation et les tests.
- Assurez-vous que le plan est bien documenté et automatisez le processus autant que possible.
- Mettez en place une stratégie de sauvegarde pour toutes les données de référence et les données transactionnelles, et testez la restauration de ces sauvegardes régulièrement.
- Formez le personnel d'exploitation à l'exécution du plan et réalisez régulièrement des simulations de sinistre pour valider et améliorer le plan.

Liste de contrôle de l'évolutivité

Conception des services

Partitionnez le scénario d'usage

- ☐ Concevez des composants de processus discrets et décomposables.
- ☐ Réduisez la taille de chaque composant, tout en respectant les règles habituelles de séparation des préoccupations et le principe de responsabilité unique.

Intégrez l'évolutivité

- ☐ Concevez vos applications de façon qu'elles réagissent à la variation de charge en augmentant et en diminuant le nombre d'instances de rôles, de files d'attente et autres services qu'elles utilisent.
- ☐ Implémentez la configuration ou la détection automatique des instances ajoutées et supprimées, de sorte que le code de l'application puisse effectuer le routage nécessaire.

Dimensionnez en tant qu'unité

- ☐ Planifiez des ressources supplémentaires pour vous adapter à la croissance.
- ☐ Pour chaque ressource, identifiez les limites supérieures d'évolutivité et utilisez le partitionnement ou la décomposition pour dépasser ces limites.
- ☐ Déterminez les unités d'échelle du système en termes d'ensembles de ressources bien définis.
- ☐ Concevez l'application de façon à pouvoir la faire évoluer facilement en ajoutant une ou plusieurs unités d'échelle.

Évitez l'affinité du client

- ☐ Si possible, faites en sorte que l'application ne nécessite pas d'affinité.
- ☐ Acheminez les demandes vers toutes les instances. Le nombre d'instances n'est pas pertinent.

Tirez parti des fonctionnalités de mise à l'échelle automatique de votre plateforme

- ☐ Préférez une capacité de mise à l'échelle automatique, telle que Azure Autoscale, à des mécanismes personnalisés ou tiers, à moins que le mécanisme intégré ne réponde pas à vos besoins.
- ☐ Utilisez si possible des règles de mise à l'échelle planifiée pour garantir la disponibilité des ressources sans délai de démarrage. Toutefois, privilégiez des règles de mise à l'échelle réactive pour faire face aux variations imprévues de la demande, le cas échéant.

Exécutez en arrière-plan les tâches intensives de processeur ou d'E/S

- ☐ Si vous vous attendez à ce qu'une demande de service absorbe des ressources considérables ou que son exécution prenne du temps, déportez le traitement de cette demande vers une tâche séparée.
- ☐ Utilisez des rôles de travail ou des tâches en arrière-plan (en fonction de la plateforme d'hébergement) pour exécuter ces tâches.

Distribuez le scénario d'usage pour les tâches en arrière-plan

- ☐ Lorsque le nombre de tâches en arrière-plan est élevé ou que les tâches nécessitent beaucoup de temps ou de ressources, répartissez le travail entre plusieurs unités de calcul (telles que des rôles de travail ou des tâches en arrière-plan).

Liste de contrôle de résilience

Concevoir votre application pour la résilience exige de prévoir et de prévenir un grand nombre de modes d'échec susceptibles de survenir. Examinez les éléments de cette liste de contrôle au regard de la conception de votre application pour améliorer sa résilience.

Exigences

☐ **Définissez les exigences de disponibilité de votre client**

- Obtenez l'accord de votre client pour les objectifs de disponibilité de chaque partie de vos applications. Pour de plus amples informations, consultez Définition de vos besoins de résilience.

Conception des applications

☐ **Effectuez une analyse du mode d'échec (FMA) pour votre application**

- Identifiez les types d'échecs susceptibles de se produire dans une application.
- Évaluez les effets potentiels et l'impact de chaque type d'échec sur l'application.
- Identifiez les stratégies de récupération.

☐ **Déployez plusieurs instances de services**

- Configurez plusieurs instances pour améliorer la résilience et l'évolutivité.
- Pour Azure App Service, sélectionnez un Plan App Service proposant plusieurs instances.
- Pour Azure Cloud Services, configurez chacun de vos rôles pour l'utilisation de plusieurs instances.
- Pour Azure Virtual Machines (VM), assurez-vous que votre architecture de machine virtuelle comprend plusieurs machines virtuelles et que chaque machine virtuelle est incluse dans un groupe à haute disponibilité.

☐ **Utilisez la mise à l'échelle automatique pour répondre aux augmentations de charge**

- Configurez votre application pour qu'elle puisse évoluer automatiquement à mesure que la charge augmente.

☐ **Utilisez l'équilibrage de charge pour distribuer les demandes**

- Si votre application utilise des machines virtuelles Azure, configurez un équilibreur de charge.

☐ **Configurez les passerelles Azure Application Gateway pour l'utilisation de plusieurs instances**

- Configurez plusieurs instances Application Gateway moyennes ou grandes pour garantir la disponibilité du service en vertu des conditions du SLA.

☐ **Utilisez des groupes à haute disponibilité pour chaque couche Application**

☐ **Envisagez le déploiement de votre application dans plusieurs régions**

- Utilisez un patron actif-actif (qui distribue les demandes sur plusieurs instances actives) ou un patron actif-passif (qui garde une instance en réserve, en cas d'échec de l'instance principale).
- Déployez plusieurs instances des services de votre application sur des paires régionales.

☐ **Utilisez Azure Traffic Manager pour acheminer le trafic de votre application vers différentes régions**

- Spécifiez une méthode de routage du trafic pour votre application.

☐ **Configurez et testez des sondes d'intégrité pour vos équilibreurs de charge et gestionnaires de trafic**

- Assurez-vous que votre logique d'intégrité vérifie les composants critiques du système et répond de manière appropriée aux sondes d'intégrité.
- Pour une sonde Traffic Manager, votre point de terminaison d'intégrité doit vérifier les dépendances critiques qui sont déployées au sein d'une même région et dont l'échec déclenche un basculement vers une autre région.
- Pour un équilibreur de charge, le point de terminaison d'intégrité doit signaler l'intégrité de la machine virtuelle.
- N'incluez pas d'autres niveaux ou services externes. Pour découvrir comment mettre en œuvre la surveillance de l'intégrité dans votre application, consultez Patron Health Endpoint Monitoring (Point de terminaison pour la surveillance de fonctionnement).

☐ **Surveillez les services tiers**

- Si votre application possède des dépendances sur des services tiers, déterminez où et comment ils peuvent échouer et les effets potentiels de ces échecs sur votre application.
- Consignez vos invocations de pilotage et de diagnostic, et mettez-les en corrélation avec la journalisation d'intégrité et les journaux de diagnostics de votre application à l'aide d'un identifiant unique.

☐ **Assurez-vous que tous les services tiers que vous utilisez fournissent un SLA**

☐ **Implémentez des patrons de résilience pour les opérations distantes le cas échéant**

- Si votre application dépend de la communication entre des services distants, conformez-vous à des patrons de conception permettant de traiter les échecs passagers, tels que le Patron Retry (Nouvelle tentative) et le Patron Circuit Breaker (Disjoncteur).

☐ **Mettez en œuvre des opérations asynchrones lorsque cela est possible**

- Concevez chaque composant de votre application pour permettre des opérations asynchrones lorsque cela est possible.

Gestion des données

☐ **Comprenez les méthodes de réplication pour les sources de données de votre application**

- Évaluez les méthodes de réplication pour chaque type de stockage de données dans Azure, notamment Azure Storage Replication et SQL Database Active Geo-Replication afin de garantir que les exigences relatives aux données de votre application sont satisfaites.

☐ **Vérifiez qu'aucun compte utilisateur unique n'a accès à la fois aux données de production et aux données de sauvegarde**

- Concevez votre application pour pouvoir limiter les autorisations des comptes utilisateurs, de sorte que seuls les utilisateurs qui ont besoin d'un accès en écriture disposent uniquement d'un accès à la production ou à la sauvegarde, mais pas aux deux.

☐ **Documentez et testez vos processus de basculement et de restauration automatique**

- Testez régulièrement les étapes documentées pour vérifier qu'un opérateur qui les suit peut basculer et restaurer avec succès la source de données.

☐ **Validez vos sauvegardes de données**

- Vérifiez régulièrement que vos données de sauvegarde correspondent à vos attentes en exécutant un script permettant de valider l'intégrité des données, le schéma et les demandes.
- Consignez et signalez toutes les incohérences afin de pouvoir réparer le service de sauvegarde.

☐ **Envisagez d'utiliser un type de compte de stockage géo-redondant**

- Lorsque vous configurez un compte de stockage, choisissez une stratégie de réplication.
- Sélectionnez Azure Read-Access Geo Redundant Storage (RA-GRS) pour protéger les données de votre application contre les cas rares d'indisponibilité d'une région entière.
- Pour les machines virtuelles, ne comptez pas sur la réplication RA-GRS pour restaurer les disques virtuels (fichiers VHD). Utilisez plutôt Azure Backup.

Sécurité

☐ **Mettez en œuvre une protection de niveau application contre les attaques distribuées de déni de service (DDoS)**

☐ **Mettez en œuvre le principe du privilège minimum pour l'accès aux ressources de l'application**

- La valeur par défaut pour l'accès aux ressources de l'application doit être aussi restrictive que possible.
- Accordez des autorisations de niveau plus élevé sur la base d'une approbation.
- Vérifiez les autorisations à privilège minimum pour les autres ressources disposant de leurs propres systèmes d'octroi d'autorisations, telles que SQL Server.

Tests

☐ **Effectuez des tests de basculement et de restauration automatique**

- Assurez-vous que les services dépendants de votre application basculent et sont restaurés automatiquement dans le bon ordre.

☐ **Effectuez des tests avec injection d'erreurs**

- Testez votre application dans un environnement aussi proche que possible de la production, en simulant ou en déclenchant des échecs réels.
- Vérifiez la capacité de votre application à récupérer après tous les types de pannes, individuellement ou en association.
- Vérifiez que les échecs ne se propagent pas ou ne provoquent pas des échecs en cascade dans votre système.

☐ **Exécutez des tests en production à l'aide de données utilisateur réelles et synthétiques**

- Utilisez un déploiement bleu/vert ou un déploiement de contrôle de validité (canari), et testez votre application en production.

Déploiement

☐ **Documentez le processus de mise en production de votre application**

- Définissez clairement et documentez votre processus de mise en production, et assurez-vous qu'il est disponible pour toute l'équipe des opérations.

☐ **Automatisez le processus de déploiement de votre application**

☐ **Concevez votre processus de mise en production de manière à maximiser la disponibilité des applications**

- Déployez votre application en production à l'aide de la technique de déploiement bleu/vert ou de déploiement de contrôle de validité (canari).

☐ **Consignez et auditez les déploiements de votre application**

- Mettez en œuvre une stratégie de journalisation fiable pour capturer le plus d'informations possible sur la version.

☐ **Prévoyez un plan de restauration pour le déploiement**

- Concevez un processus de restauration permettant de revenir à une dernière version valide connue et de minimiser les temps d'arrêt.

Opérations

☐ **Mettez en œuvre les meilleures pratiques en matière de pilotage et d'alertes dans votre application**

☐ **Mesurez les statistiques d'appels distants et mettez les informations à la disposition de l'équipe chargée des applications**

- Résumez les métriques d'appel distant comme la latence, le débit et les erreurs dans les centiles 99 et 95.
- Effectuez une analyse statistique des métriques pour déceler les erreurs qui se produisent au sein de chaque centile.

☐ **Réalisez un suivi du nombre d'exceptions passagères et effectuez une nouvelle tentative sur une période appropriée**

☐ **Mettez en place un système d'alerte précoce qui prévient un opérateur**

- Identifiez les indicateurs de performance clés de l'intégrité de votre application, tels que les exceptions passagères et la latence des appels distants, et définissez des valeurs seuils appropriées pour chacun d'eux.
- Envoyez une alerte aux opérations lorsque la valeur seuil est atteinte.
- Définissez ces seuils à des niveaux permettant d'identifier les problèmes avant qu'ils ne deviennent critiques et ne nécessite une solution de récupération.

☐ **Assurez-vous que plusieurs personnes de l'équipe sont formées au pilotage de l'application et à l'exécution des étapes de récupération manuelle**

- Formez plusieurs personnes à la détection et à la récupération, et faites en sorte de pouvoir compter sur la disponibilité de l'une d'entre elles à tout moment.

☐ **Assurez-vous que votre application ne se heurte pas aux** limites de l'abonnement Azure

- Si les besoins de votre application dépassent les limites de l'abonnement Azure, créez un autre abonnement Azure et réservez des ressources suffisantes sur celui-ci.

☐ **Assurez-vous que votre application ne se heurte pas aux** limites par service

- Montez en puissance (par exemple, en choisissant un autre niveau de tarification) ou descendez en puissance (en ajoutant de nouvelles instances) pour éviter les limites par service.

☐ **Adaptez les besoins de stockage de votre application aux objectifs d'évolutivité et de performance du stockage Azure**

- Concevez votre application pour l'utilisation du stockage prévu dans ces objectifs.
- Configurez des comptes de stockage supplémentaires si vous dépassez ces objectifs de stockage.
- Si vous vous heurtez à la limite du compte de stockage, configurez des abonnements Azure supplémentaires, puis configurez des comptes de stockage supplémentaires.

☐ **Déterminez si le scénario d'usage de votre application est stable ou s'il varie au fil du temps**

- Si votre scénario d'usage varie au fil du temps, utilisez des groupes de machines virtuelles identiques Azure pour mettre automatiquement à l'échelle le nombre d'instances de machine virtuelle.

☐ **Sélectionnez le niveau de service adéquat pour Azure SQL Database**

- Si votre application utilise Azure SQL Database, assurez-vous d'avoir sélectionné le niveau de service approprié.

☐ **Créez un processus permettant d'interagir avec le support Azure**

- Incluez dès le départ, dans le cadre de la résilience de votre application, un processus permettant de contacter le support et de transférer les problèmes en escalade.

☐ **Faites en sorte que votre application n'utilise pas plus que le nombre maximal de comptes de stockage par abonnement**

- Si votre application requiert plus de 200 comptes de stockage, vous devrez créer un nouvel abonnement et configurer des comptes de stockage supplémentaires sur celui-ci.

☐ **Veillez à ce que votre application ne dépasse pas les objectifs d'évolutivité des disques virtuels**

- Si votre application dépasse les objectifs d'évolutivité des disques virtuels, configurez des comptes de stockage supplémentaires et créez les disques virtuels sur ces comptes.

Télémétrie

☐ **Consignez les données de télémétrie dans l'environnement de production**

- Capturez des informations télémétriques fiables pendant l'exécution de l'application dans l'environnement de production.

☐ **Implémentez la journalisation à l'aide d'un patron asynchrone**

- Veillez à ce que vos opérations de journalisation soient mises en œuvre en tant qu'opérations asynchrones.

☐ **Mettez en corrélation les données des journaux au-delà des limites de service**

- Assurez-vous que votre système de journalisation met en corrélation les appels au-delà des limites de service, pour que vous puissiez suivre la demande sur l'ensemble de votre application.

Ressources Azure

☐ **Utilisez des modèles Azure Resource Manager pour réserver des ressources**

☐ **Donnez des noms explicites à vos ressources**

☐ **Utilisez le contrôle d'accès en fonction du rôle (RBAC).**

☐ **Verrouillez les ressources critiques, telles que les machines virtuelles**

☐ **Choisissez des paires régionales**

- Lors du déploiement sur deux régions, choisissez des régions dans la même paire régionale.

☐ **Organisez les groupes de ressources par fonction et cycle de vie**

- Créez des groupes de ressources distincts pour la production, le développement et les environnements de test.
- Lors d'un déploiement multirégion, placez les ressources de chaque région dans des groupes de ressources distincts. Vous pourrez ainsi effectuer facilement un redéploiement sur une région sans affecter la/les autres région(s).

Services Azure

Les éléments de liste de contrôle suivants s'appliquent à des services spécifiques dans Azure.

App Service

☐ **Utilisez le niveau Standard ou Premium**

☐ **Évitez la montée ou la descente en puissance**

- Sélectionnez un niveau et une taille d'instance qui répondent à vos exigences de performances dans des conditions de charge typiques, puis procédez à une montée en puissance des instances pour gérer les variations du volume de trafic.

☐ **Stockez la configuration en tant que paramètres de l'application**

- Utilisez les paramètres d'application pour stocker les paramètres de configuration de l'application.

  Définissez les paramètres dans vos modèles Resource Manager, ou à l'aide de PowerShell, pour pouvoir les appliquer dans le cadre d'un processus de déploiement/mise à jour automatique.

☐ **Créez des plans App Service distincts pour la production et les tests**

- N'utilisez pas d'emplacements de votre déploiement en production pour effectuer les tests.

☐ **Séparez les applications web des API Web.**

- Si votre solution comporte à la fois un serveur web frontal et une API Web, envisagez de les décomposer en applications App Service distinctes.
- Si vous n'avez pas besoin de ce niveau d'évolutivité au départ, vous pouvez commencer par déployer les applications dans le même plan et les déplacer vers des plans distincts ultérieurement.

☐ **Évitez d'utiliser la fonctionnalité de sauvegarde App Service pour sauvegarder des bases de données Azure SQL**

- Utilisez des sauvegardes automatiques SQL Database.

☐ **Déployez sur un emplacement de test**

- Créez un emplacement de déploiement pour le test. Déployez des mises à jour de l'application sur l'emplacement de test et vérifiez le déploiement avant de l'intégrer en production.

☐ **Créez un emplacement de déploiement pour conserver le dernier déploiement valide connu (LKG)**

- Lors du déploiement d'une mise à jour en production, déplacez le déploiement en production précédent vers l'emplacement LKG.

☐ **Activez la journalisation des diagnostics**

- Notamment la journalisation des applications et la journalisation du serveur web.

☐ **Consignez les données de journalisation dans le stockage d'objets blob**

☐ **Créez un compte de stockage distinct pour les journaux**

- N'utilisez pas le même compte de stockage pour les journaux et les données d'application.

☐ **Surveillez les performances**

- Utilisez un service de pilotage des performances tel que New Relic ou Application Insights pour surveiller les performances de l'application et son comportement sous charge.

Application Gateway

☐ **Configurez au moins deux instances**

- Déployez au moins deux instances de la passerelle Application Gateway. Pour être éligible pour le SLA, vous devez configurer plusieurs instances moyennes ou grandes.

Azure Search

☐ **Configurez plusieurs réplicas**

- Utilisez au moins deux réplicas pour la haute disponibilité en lecture, ou trois pour la haute disponibilité en lecture-écriture.

☐ **Configurez des indexeurs pour les déploiements sur plusieurs régions**

- Si la source de données est géo-répliquée, faites pointer chaque indexeur de chaque service régional Azure Search vers son réplica local de source de données.
- Par contre, pour les grands jeux de données stockés dans Azure SQL Database, faites pointer tous les indexeurs vers le réplica principal. Après un basculement, faites pointer les indexeurs Azure Search vers le nouveau réplica principal.
- Si la source de données n'est pas géo-répliquée, faites pointer plusieurs indexeurs vers cette même source de données pour que les services Azure Search dans plusieurs régions effectuent une indexation continue et indépendante à partir de la source de données.

Azure Storage

☐ **Pour les données d'application, utilisez le stockage géo-redondant avec accès en lecture (RA-GRS)**

☐ **Pour les disques virtuels, utilisez Managed Disks**

☐ **Pour les files d'attente de stockage, créez une file d'attente de sauvegarde dans une autre région**

- Créez une file d'attente de sauvegarde sur un compte de stockage dans une autre région.

Cosmos DB

☐ **Répliquez la base de données dans toutes les régions.**

SQL Database

☐ **Utilisez le niveau Standard ou Premium**

☐ **Activez l'audit SQL Database**

☐ **Utilisez la géo-réplication active**

- Si votre base de données primaire échoue ou, tout simplement, si vous devez la mettre hors ligne, effectuez un basculement manuel vers la base de données secondaire.

☐ **Utilisez le partitionnement**

☐ **Utilisez la limite de restauration dans le temps pour assurer la récupération d'urgence en cas d'erreur humaine**

☐ **Utilisez la géo-restauration pour assurer une récupération d'urgence après une interruption du service**

SQL Server (exécuté sur une machine virtuelle)

☐ **Répliquez la base de données**

- Utilisez les groupes de disponibilité AlwaysOn SQL Server pour répliquer la base de données.

☐ **Sauvegardez la base de données**

- Si vous utilisez déjà Azure Backup pour sauvegarder vos machines virtuelles, envisagez d'utiliser Azure Backup pour les scénarios d'usage SQL Server à l'aide de DPM.
- Sinon, utilisez les opérations de sauvegarde gérées de SQL Server vers Microsoft Azurew

Traffic Manager

☐ **Effectuez une restauration manuelle**

- Après un basculement Traffic Manager, préférez la restauration manuelle à la restauration automatique.
- Avant d'effectuer la restauration, vérifiez que tous les sous-systèmes d'application sont sains.

☐ **Créez un point de terminaison à sonde d'intégrité**

- Créez un point de terminaison personnalisé qui signale l'état d'intégrité général de l'application.
- Toutefois, ne signalez pas les erreurs pour les services non critiques.

Machines virtuelles

☐ **Évitez d'exécuter un scénario d'usage de production sur une machine virtuelle unique**

- Placez plusieurs machines virtuelles dans un groupe à haute disponibilité ou un groupe de machines virtuelles, avec un équilibreur de charge frontal.

☐ **Spécifiez un groupe à haute disponibilité lors de la configuration de la machine virtuelle**

- Lorsque vous ajoutez une machine virtuelle à un groupe à haute disponibilité existant, veillez à créer une carte d'interface réseau pour la machine virtuelle et à ajouter cette carte au pool d'adresses frontales sur l'équilibreur de charge.

☐ **Placez chaque couche Application dans un groupe à haute disponibilité distinct**

- Dans une application multicouche, ne placez pas des machines virtuelles issues de différentes couches dans le même groupe à haute disponibilité.
- Pour bénéficier des avantages de la redondance des FD (domaines d'erreur) et des UD (domaines de mise à jour), chaque machine virtuelle du groupe à haute disponibilité doit être capable de gérer les mêmes demandes du client.

☐ **Choisissez la taille de machine virtuelle adaptée à vos exigences de performances**

- Lors du transfert d'un scénario d'usage existant vers Azure, commencez par la taille de machine virtuelle la plus adaptée à vos serveurs sur site.
- Mesurez ensuite la performance de votre scénario d'usage réel en matière d'UC, de mémoire et d'IOPS de disque, puis ajustez la taille en fonction des besoins.
- Si vous avez besoin de plusieurs cartes réseau, tenez compte de la limite relative aux cartes réseau pour chaque taille.

☐ **Utilisez des disques gérés pour les disques durs virtuels.**

☐ **Installez les applications sur un disque de données et non sur le disque du système d'exploitation.**

☐ **Utilisez Sauvegarde Azure pour sauvegarder des machines virtuelles.**

☐ **Activez les journaux de diagnostic.**

- Incluez des mesures d'intégrité de base, des journaux d'infrastructure et un diagnostic d'amorçage.

☐ **Utilisez l'extension AzureLogCollector.**

☐ **Pour inscrire sur une liste blanche ou bloquer des adresses IP publiques, ajoutez un groupe de sécurité réseau (NSG) au sous-réseau.**

- Bloquez l'accès aux utilisateurs malveillants ou autorisez l'accès uniquement aux utilisateurs qui ont l'autorisation d'accéder à l'application.

☐ **Créez une sonde d'intégrité personnalisée.**

- Pour une sonde HTTP, utilisez un point de terminaison personnalisé qui rend compte de l'intégrité globale de l'application, y compris toutes les dépendances critiques.

☐ **Ne bloquez pas la sonde d'intégrité.**

- Ne bloquez pas le trafic vers ou à partir de cette adresse IP par toute stratégie de pare-feu ou règle de groupe de sécurité réseau (NSG).

☐ **Activez la journalisation de l'équilibrage de charge.**

Résumé

Dans ce guide, vous avez appris à choisir le style d'architecture correspondant à l'application, à sélectionner les technologies de calcul et de stockage de données les plus appropriées et à appliquer des principes et des piliers de conception lors de la création de vos applications.

À l'avenir, de nouvelles tendances, les demandes des utilisateurs et les capacités continueront à créer encore plus d'occasions d'améliorer vos architectures. Pour rester à l'avant-garde, nous vous encourageons à demeurer informés grâce aux ressources et conseils ci-dessous :

- Ajoutez le centre d'Architecture à aka.ms/architecturecenter à vos favoris.
- Consultez le Centre de documentation d'Azur pour y trouver des conseils détaillés, des guides de démarrage rapide et des téléchargements.
- Obtenez gratuitement une formation Azure en ligne gratuite, y compris une formation Pluralsight et des parcours d'apprentissage guidés.

Commencez à créer grâce à un compte Azure gratuit.

Si vous ne l'avez pas déjà fait, créez un compte Azure gratuit pour profiter d'un certain nombre d'avantages, notamment les suivants :

- Crédit de 200 $ à utiliser sur n'importe quel produit Azure pendant 30 jours.
- Accès gratuit à nos produits les plus populaires du marché pendant 12 mois, y compris le calcul, le stockage en réseau et la base de données.
- Plus de 25 produits disponibles en permanence gratuitement.

Obtenez l'aide d'experts.

Contactez-nous à l'adresse aka.ms/azurespecialist

Architectures Azure de référence

Nos architectures de référence sont organisées par scénario, en regroupant les architectures associées. Chaque architecture inclut des pratiques recommandées ainsi que des considérations en matière d'évolutivité, de disponibilité, de facilité de gestion et de sécurité. La plupart comprennent également une solution pouvant être déployée.

Gestion des identités

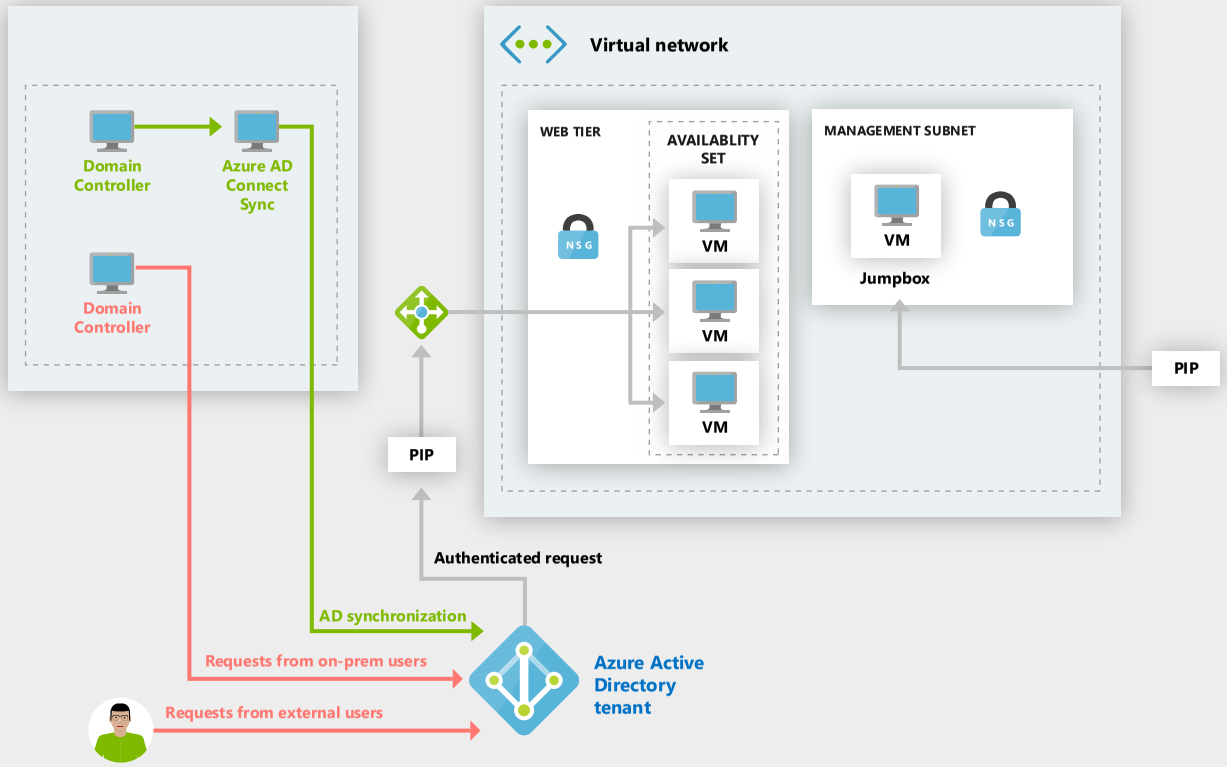
Ces architectures de référence présentent des options d'intégration de votre environnement Active Directory (AD) sur site à un réseau Azure.

For this scenario	Consider this architecture
Votre application est hébergée en partie sur site et en partie dans Azure.	Intégrez votre environnement AD sur site avec l'AD d'Azur.
Vous devez utiliser des fonctionnalités AD DS qui ne sont actuellement pas mises en œuvre par Azure AD.	Étendez les services AD DS à Azure.
Vous devez maintenir la séparation en matière de sécurité pour les objets et les identités se trouvant dans le cloud ou migrer des domaines individuels sur site vers le cloud.	Créez une forêt AD DS dans Azure.
Vous devez : • Authentifier et autoriser les utilisateurs d'organisations partenaires. • Permettre aux utilisateurs de s'authentifier sur des navigateurs Web fonctionnant hors du pare-feu de l'organisation. • Permettre aux utilisateurs de se connecter à partir de périphériques externes autorisés, tels que des appareils mobiles.	Étendre les services AD FS à Azure.

Microsoft

Integrate on-premises Active Directory domains with Azure Active Directory

This architecture integrates Azure Active Directory (Azure AD) with an on-premises Active Directory domain. Your on-premises AD directories are replicated to Azure AD. This allows your applications that run in Azure to authenticate users with their on-premises identities, and enables a single sign-on (SSO) experience.



Recommendations

- If you have multiple on-premises domains in a forest, store and synchronize information for the entire forest to a single Azure AD tenant. Filter identities to avoid duplication.
- To achieve high availability for the AD Connect sync service, run a secondary staging server.
- If you are likely to synchronize more than 100,000 objects from your local directory, use a production version of SQL Server, and use SQL clustering to achieve high availability.
- Protect on-premises applications that can be accessed externally. Use the Azure AD Application Proxy to provide controlled access to on-premises web applications for external users.
- Actively monitor Azure AD for signs of suspicious activity.
- Use conditional access control to deny authentication requests from unexpected sources.

Azure

Architecture Components

Azure AD tenant

An instance of Azure AD created by your organization. It acts as a directory service for cloud applications by storing objects copied from the on-premises Active Directory and provides identity services.

Web tier subnet

This subnet holds VMs that run a web application. Azure AD can act as an identity broker for this application.

On-premise AD DS server

An on-premise directory and identity service. The AD DS directory can be synchronized with Azure AD to enable it to authenticate on-premise users.

Azure AD Connect sync server

An on-premises computer that runs the Azure AD Connect sync service. This service synchronizes information held in the on-premises Active Directory to Azure AD. For example, if you provision or deprovision groups and users on-premises, these changes propagate to Azure AD.

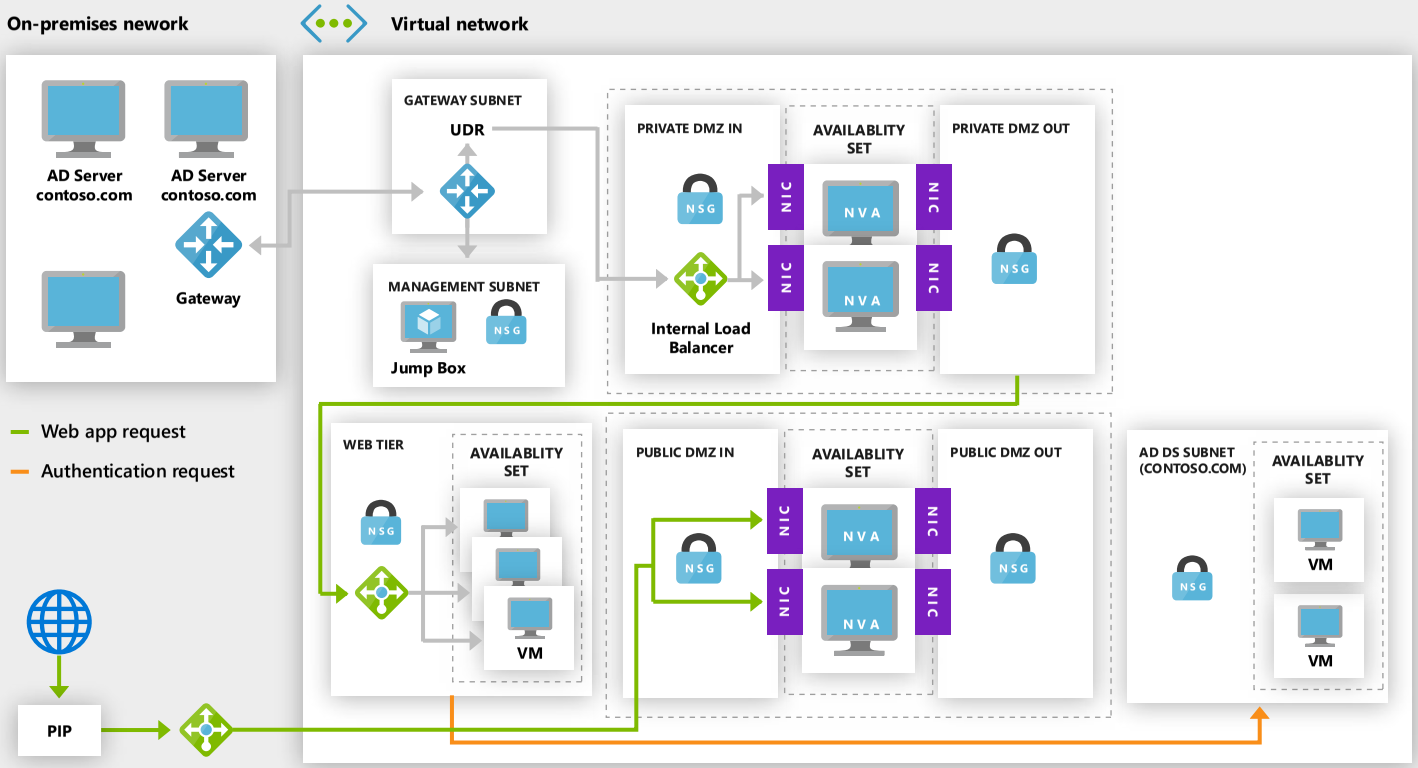
Note:

For simplicity, this diagram only shows the connections directly related to Azure AD, and does not show protocol-related traffic that may occur as part of authentication and identity federation. For example, a web application may redirect the web browser to authenticate the request through Azure AD. Once authenticated, the request can be passed back to the web application, with the appropriate identity information.

Microsoft

Extend Active Directory Domain Services (AD DS) to Azure

This architecture extends an on-premises Active Directory environment to Azure using Active Directory Domain Services (AD DS). This architecture can reduce the latency caused by sending authentication and local authorization requests from the cloud back to AD DS running on-premises. Consider this option if you need to use AD DS features that are not currently implemented by Azure AD.



Recommendations

- Deploy at least two VMs running AD DS as domain controllers and add them to an availability set.
- For VM size, use the on-premises AD DS machines as a starting point, and pick the closest Azure VM sizes.
- Create a separate virtual data disk for storing the database, logs, and SYSVOL for Active Directory.
- Configure the VM network interface (NIC) for each AD DS server with a static private IP address for full domain name service (DNS) support.
- Monitor the resources of the domain controller VMs as well as the AD DS Services and create a plan to quickly correct any problems.
- Perform regular AD DS backups. Don't simply copy the VHD files of domain controllers, because the AD DS database file on the VHD may not be in a consistent state when it's copied.
- Do not shut down a domain controller VM using Azure portal. Instead, shut down and restart from the guest operating system.
- Use either BitLocker or Azure disk encryption to encrypt the disk hosting the AD DS database.
- Azure disk encryption to encrypt the disk hosting the AD DS database.

Azure

Architecture Components

On-premises network

The on-premises network includes local Active Directory servers that can perform authentication and authorization for components located on-premises.

Active Directory servers

These are domain controllers implementing directory services (AD DS) running as VMs in the cloud. These servers can provide authentication of components running in your Azure virtual network.

Active Directory subnet

The AD DS servers are hosted in a separate subnet. Network security group (NSG) rules protect the AD DS servers and provide a firewall against traffic from unexpected sources.

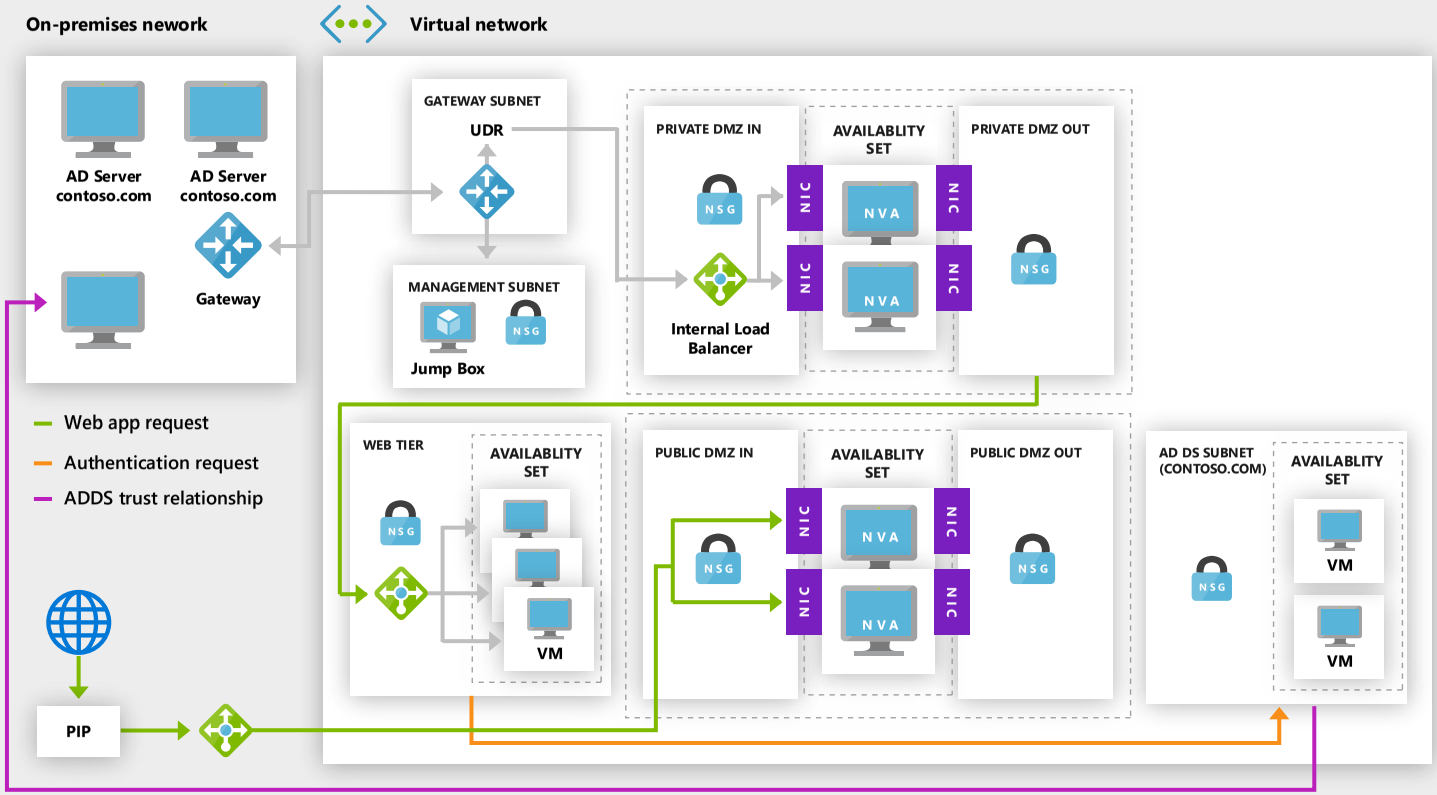
Azure Gateway and Active Directory synchronization

The Azure gateway provides a connection between the on-premises network and the Azure VNet. This can be a VPN connection or Azure ExpressRoute. All synchronization requests between the Active Directory servers in the cloud and on-premises pass through the gateway. User-defined routes (UDRs) handle routing for on-premises traffic that passes to Azure. Traffic to and from the Active Directory servers does not pass through the network virtual appliances (NVAs) used in this scenario.

Microsoft

Create an Active Directory Domain Services (AD DS) resource forest in Azure

This architecture shows an AD DS forest in Azure with a one-way trust relationship with an on-premises AD domain. The forest in Azure contains a domain that does not exist on-premises. This architecture maintains security separation for objects and identities held in the cloud, while allowing on-premises identities to access your applications running in Azure.



Recommendations

- Provision at least two domain controllers for each domain. This enables automatic replication between servers.
- Create an availability set for the VMs acting as Active Directory servers handling each domain. Put at least two servers in this availability set.
- Consider designating one or more servers in each domain as standby operations masters in case connectivity to a server acting as a flexible single master operation (FSMO) role fails.

Azure

Architecture Components

On-premises network

The on-premises network contains its own Active Directory forest and domains.

Active Directory servers

These are domain controllers implementing domain services running as VMs in the cloud. These servers host a forest containing one or more domains, separate from those located on-premises.

One-way trust relationship

The example in the diagram shows a one-way trust from the domain in Azure to the on-premises domain. This relationship enables on-premises users to access resources in the domain in Azure, but not the other way around. It is possible to create a two-way trust if cloud users also require access to on-premises resources.

Active Directory subnet

The AD DS servers are hosted in a separate subnet. Network security group (NSG) rules protect the AD DS servers and provide a firewall against traffic from unexpected sources.

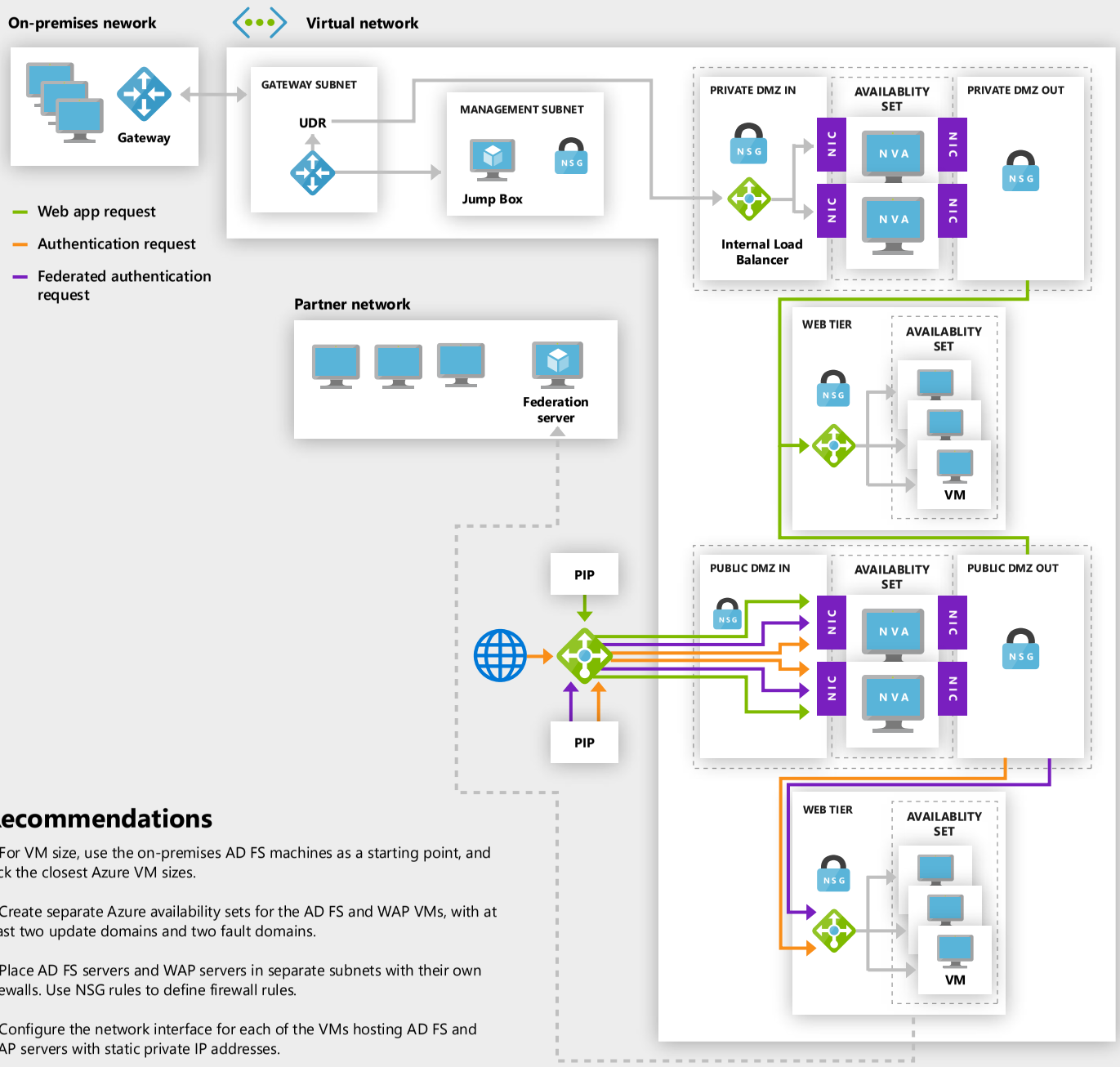
Azure gateway

The Azure gateway provides a connection between the on-premises network and the Azure VNet. This can be a VPN connection or Azure ExpressRoute. For more information, see Implementing a secure hybrid network architecture in Azure.

Microsoft

Active Directory Federation Services (AD FS)

This architecture extends an on-premises network to Azure and uses Active Directory Federation Services (AD FS) to perform federated authentication and authorization. AD FS can be hosted on-premises, but for applications running in Azure, it may be more efficient to replicate AD FS in the cloud.
Use this architecture to authenticate users from partner organizations, allow users to authenticate from outside of the organizational firewall, or allow users to connect from authorized mobile devices.



Recommendations

- For VM size, use the on-premises AD FS machines as a starting point, and pick the closest Azure VM sizes.
- Create separate Azure availability sets for the AD FS and WAP VMs, with at least two update domains and two fault domains.
- Place AD FS servers and WAP servers in separate subnets with their own firewalls. Use NSG rules to define firewall rules.
- Configure the network interface for each of the VMs hosting AD FS and WAP servers with static private IP addresses.
- Prevent direct exposure of the AD FS servers to the Internet.
- Do not join the WAP servers to the domain.

Azure

Architecture Components

AD DS subnet

The AD DS servers are contained in their own subnet with network security group (NSG) rules acting as a firewall.

AD DS servers

Domain controllers running as VMs in Azure. These servers provide authentication of local identities within the domain.

AD FS subnet

The AD FS servers are located within their own subnet with NSG rules acting as a firewall.

AD FS servers

The AD FS servers provide federated authorization and authentication. In this architecture, they perform the following tasks:

Receiving security tokens containing claims made by a partner federation server on behalf of a partner user. AD FS verifies that the tokens are valid before passing the claims to the web application running in Azure to authorize requests.

The web application running in Azure is the relying party. The partner federation server must issue claims that are understood by the web application. The partner federation servers are referred to as account partners, because they submit access requests on behalf of authenticated accounts in the partner organization. The AD FS servers are called resource partners because they provide access to resources (the web application).

Authenticating and authorizing incoming requests from external users running a web browser or device that needs access to web applications, by using AD DS and the Active Directory Device Registration Service.

The AD FS servers are configured as a farm accessed through an Azure load balancer. This implementation improves availability and scalability. The AD FS servers are not exposed directly to the Internet. All Internet traffic is filtered through AD FS web application proxy servers and a DMZ (also referred to as a perimeter network).

AD FS proxy subnet

The AD FS proxy servers can be contained within their own subnet, with NSG rules providing protection. The servers in this subnet are exposed to the Internet through a set of network virtual appliances that provide a firewall between your Azure virtual network and the Internet.

AD FS web application proxy (WAP) servers

These VMs act as AD FS servers for incoming requests from partner organizations and external devices. The WAP servers act as a filter, shielding the AD FS servers from direct access from the Internet. As with the AD FS servers, deploying the WAP servers in a farm with load balancing gives you greater availability and scalability than deploying a collection of stand-alone servers.

Partner organization

A partner organization running a web application that requests access to a web application running in Azure. The federation server at the partner organization authenticates requests locally, and submits security tokens containing claims to AD FS running in Azure. AD FS in Azure validates the security tokens, and if valid can pass the claims to the web application running in Azure to authorize them.

Réseau hybride

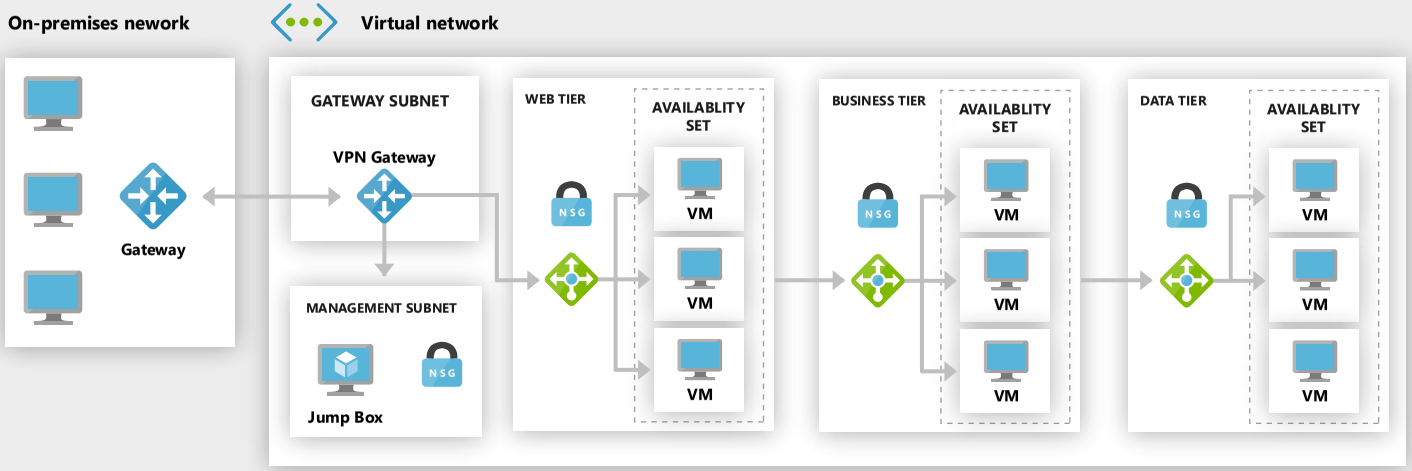
Ces architectures de référence présentent des pratiques éprouvées permettant de créer une connexion réseau fiable entre un réseau sur site et Azur.

For this scenario	Consider this architecture
Vous disposez d'applications hybrides à faible trafic entre le matériel sur site et le cloud.	VPN
Vos applications hybrides exécutent des scénarios d'usage essentiels à grande échelle qui requièrent un haut degré d'évolutivité.	ExpressRoute
Vous disposez d'applications hybrides qui nécessitent la bande passante supérieure d'ExpressRoute ainsi qu'une connectivité réseau hautement disponible.	ExpressRoute avec basculement VPN

Microsoft

Connect an on-premises network to Azure using a VPN gateway

This architecture extends an on-premises network to Azure, using a site-to-site virtual private network (VPN). Traffic flows between the on-premises network and the Azure Virtual Network (VNet).



Recommendations

- Create an Azure VNet with an address space large enough for all of your required resources, with room for growth in case more VMs are needed in the future. The address space of the VNet must not overlap with the on-premises network.
- The virtual network gateway requires a subnet named GatewaySubnet. Do not deploy any VMs to the gateway subnet. Also, do not assign an NSG to this subnet, as it will cause the gateway to stop functioning.
- Create a policy-based gateway if you need to closely control how requests are routed based on policy criteria such as address prefixes. Create a route-based gateway if you connect to the on-premises network using RRAS, support multi-site or cross-region connections, or implement VNet-to-VNet connections.
- Ensure that the on-premises routing infrastructure is configured to forward requests intended for addresses in the Azure VNet to the VPN device.
- Open any ports required by the cloud application in the on-premises network.
- If you need to ensure that the on-premises network remains available to the Azure VPN gateway, implement a failover cluster for the on-premises VPN gateway.
- If your organization has multiple on-premises sites, create multi-site connections to one or more Azure VNets. This approach requires dynamic (route-based) routing, so make sure that the on-premises VPN gateway supports this feature.
- Generate a different shared key for each VPN gateway. Use a strong shared key to help resist brute-force attacks.
- If you need higher bandwidth than a VPN connection supports, consider using an Azure ExpressRoute connection instead.

Azure

Architecture Components

On-premises network

A private local-area network running within an organization.

VPN appliance

A device or service that provides external connectivity to the on-premises network. The VPN appliance may be a hardware device, or it can be a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring them to connect to an Azure VPN gateway, see the instructions for the selected device in the article About VPN devices for Site-to-Site VPN Gateway connections.

Virtual network (VNet)

The cloud application and the components for the Azure VPN gateway reside in the same VNet.

Azure VPN gateway

The VPN gateway service enables you to connect the VNet to the on-premises network through a VPN appliance. For more information, see Connect an on-premises network to a Microsoft Azure virtual network. The VPN gateway includes the following elements:

Virtual network gateway

A resource that provides a virtual VPN appliance for the VNet. It is responsible for routing traffic from the on-premises network to the VNet.

Local network gateway

An abstraction of the on-premises VPN appliance. Network traffic from the cloud application to the on-premises network is routed through this gateway.

Connection

The connection has properties that specify the connection type (IPSec) and the key shared with the on-premises VPN appliance to encrypt traffic.

Gateway subnet

The virtual network gateway is held in its own subnet, which is subject to various requirements, described in the Recommendations section below.

Cloud Application

The application hosted in Azure. It might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see Running Windows VM workloads and Running Linux VM workloads.

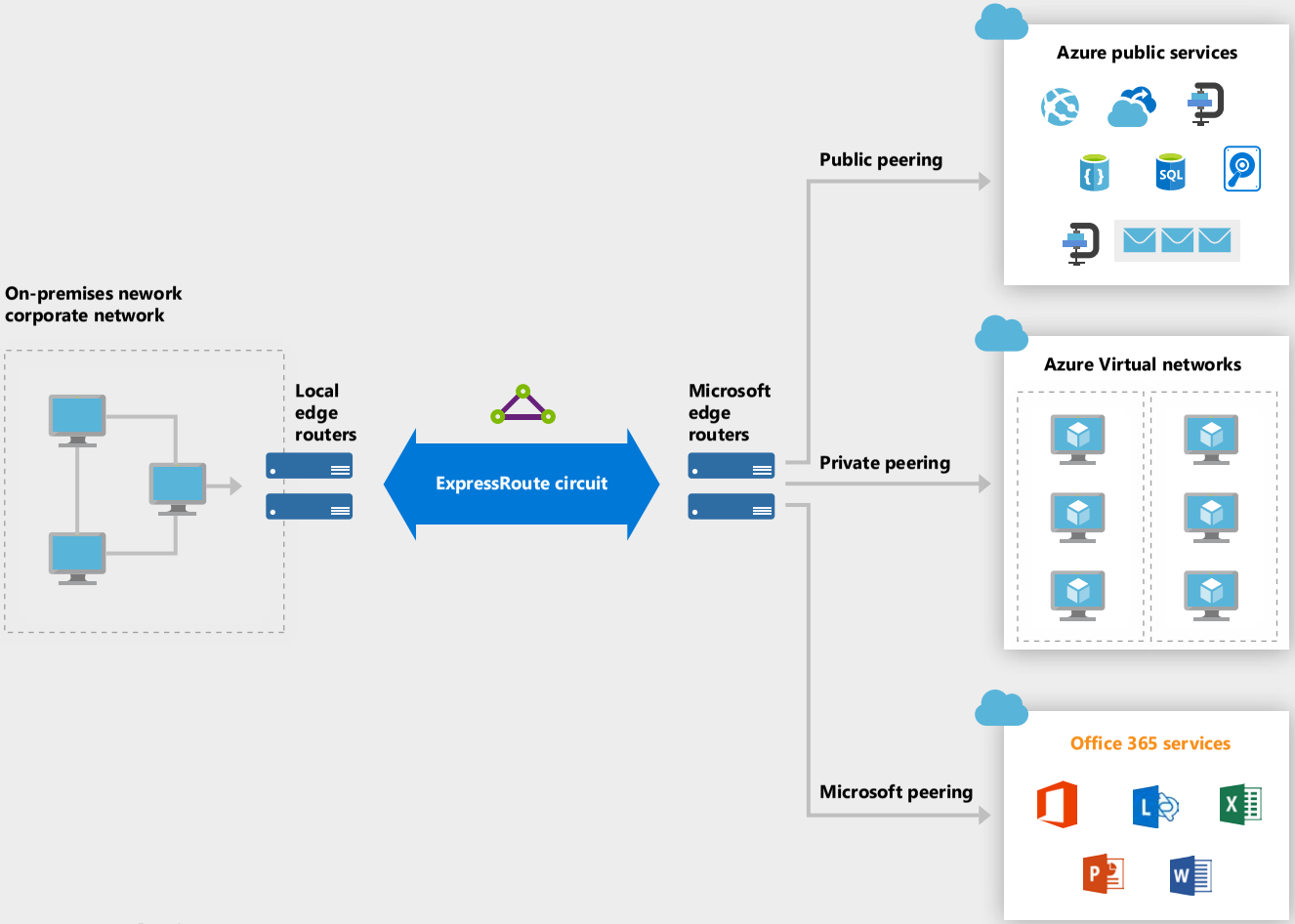
Internal load balancer

Network traffic from the VPN gateway is routed to the cloud application through an internal load balancer. The load balancer is located in the front-end subnet of the application.

Microsoft

Connect an on-premises network to Azure using ExpressRoute

This architecture extends an on-premises network to Azure, using Azure ExpressRoute. ExpressRoute connections use a private, dedicated connection through a third-party connectivity provider. The private connection extends your on-premises network into Azure.



Recommendations

- Ensure that your organization has met the ExpressRoute prerequisite requirements for connecting to Azure. See ExpressRoute prerequisites & checklist.
- Create an Azure VNet with an address space large enough for all of your required resources, with room for growth in case more VMs are needed in the future. The address space of the VNet must not overlap with the on-premises network.
- The virtual network gateway requires a subnet named GatewaySubnet. Do not deploy any VMs to the gateway subnet. Also, do not assign an NSG to this subnet, as it will cause the gateway to stop functioning.
- Although some providers allow you to change your bandwidth, make sure you pick an initial bandwidth that surpasses your needs and provides room for growth.
- Consider the following options for high availability:

  If you're using a layer 2 connection, deploy redundant routers in your on-premises network in an active-active configuration. Connect the primary circuit to one router, and the secondary circuit to the other.

  If you're using a layer 3 connection, verify that it provides redundant BGP sessions that handle availability for you.

  Connect the VNet to multiple ExpressRoute circuits, supplied by different service providers. This strategy provides additional high-availability and disaster recovery capabilities.

  Configure a site-to-site VPN as a failover path for ExpressRoute. This option only applies to private peering. For Azure and Office 365 services, the Internet is the only failover path.

Azure

Architecture Components

On-premises network

A private local-area network running within an organization.

ExpressRoute circuit

A layer 2 or layer 3 circuit supplied by the connectivity provider that joins the on-premises network with Azure through the edge routers. The circuit uses the hardware infrastructure managed by the connectivity provider.

Local edge routers

Routers that connect the on-premises network to the circuit managed by the provider. Depending on how your connection is provisioned, you may need to provide the public IP addresses used by the routers.

Microsoft edge routers

Two routers in an active-active highly available configuration. These routers enable a connectivity provider to connect their circuits directly to their datacenter. Depending on how your connection is provisioned, you may need to provide the public IP addresses used by the routers.

Azure virtual networks (VNets)

Each VNet resides in a single Azure region, and can host multiple application tiers. Application tiers can be segmented using subnets in each VNet.

Azure public services

Azure services that can be used within a hybrid application. These services are also available over the Internet, but accessing them using an ExpressRoute circuit provides low latency and more predictable performance, because traffic does not go through the Internet. Connections are performed using public peering, with addresses that are either owned by your organization or supplied by your connectivity provider.

Office 365 services

The publicly available Office 365 applications and services provided by Microsoft. Connections are performed using Microsoft peering, with addresses that are either owned by your organization or supplied by your connectivity provider. You can also connect directly to Microsoft CRM Online through Microsoft peering.

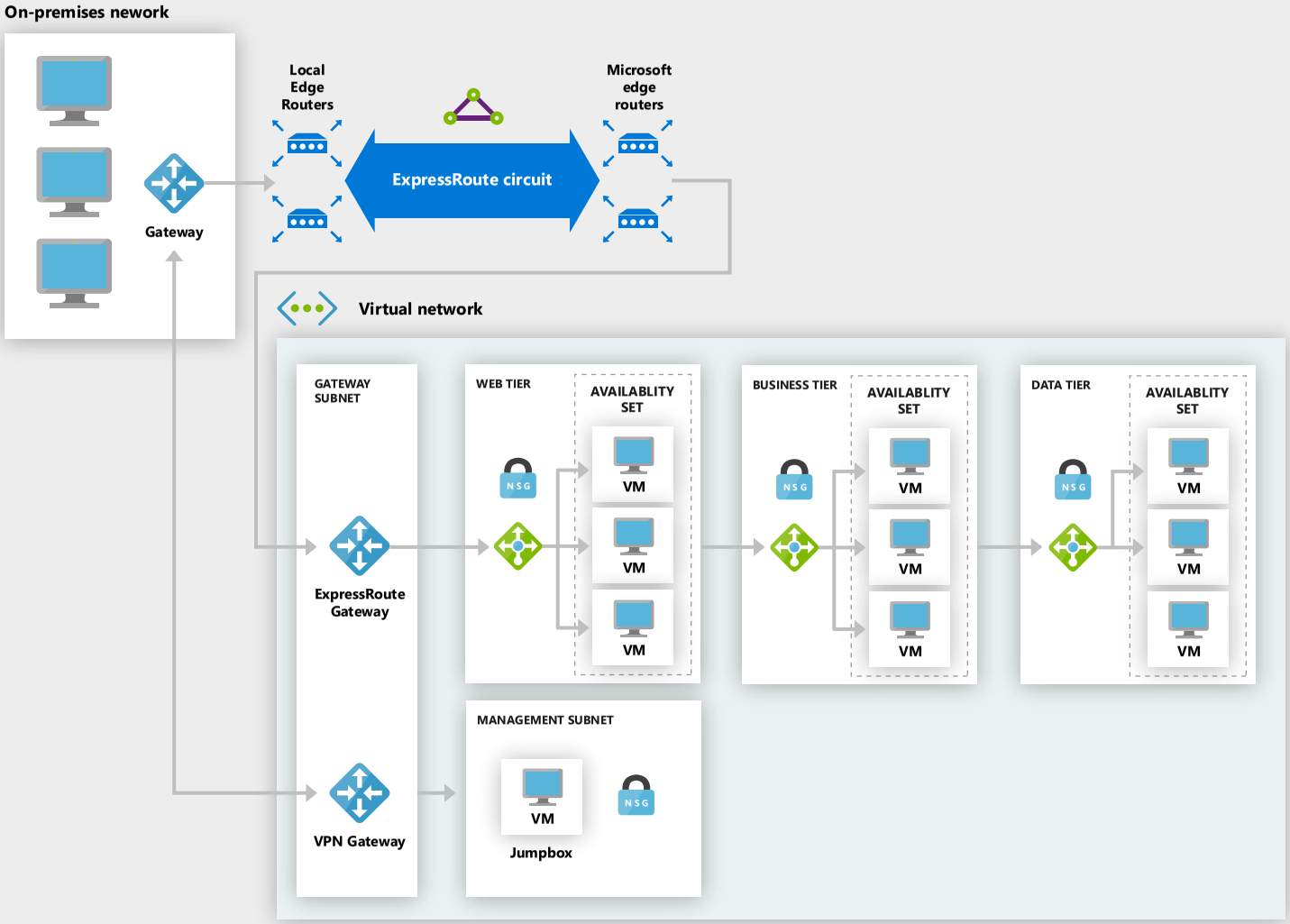
Connectivity providers (not shown)

Companies that provide a connection either using layer 2 or layer 3 connectivity between your datacenter and an Azure datacenter.

Microsoft

Connect an on-premises network to Azure using ExpressRoute with VPN failover

This architecture extends an on-premises network to Azure by using ExpressRoute, with a site-to-site virtual private network (VPN) as a failover connection. Traffic flows between the on-premises network and the Azure VNet through an ExpressRoute connection. If there is a loss of connectivity in the ExpressRoute circuit, traffic is routed through an IPSec VPN tunnel.



Recommendations

- The recommendations from the previous two architectures apply to this architecture.

- After you establish the virtual network gateway connections, test the environment. First make sure you can connect from your on-premises network to your Azure VNet. This connection will use ExpressRoute. Then contact your provider to stop ExpressRoute connectivity for testing, and verify that you can still connect using the VPN connection.

Azure

Architecture Components

On-premises network

A private local-area network running within an organization.

VPN appliance

A device or service that provides external connectivity to the on-premises network. The VPN appliance may be a hardware device, or it can be a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring selected VPN appliances for connecting to Azure, see About VPN devices for Site-to-Site VPN Gateway connections.

ExpressRoute circuit

A layer 2 or layer 3 circuit supplied by the connectivity provider that joins the on-premises network with Azure through the edge routers. The circuit uses the hardware infrastructure managed by the connectivity provider.

ExpressRoute virtual network gateway

The ExpressRoute virtual network gateway enables the VNet to connect to the ExpressRoute circuit used for connectivity with your on-premises network.

VPN virtual network gateway

The VPN virtual network gateway enables the VNet to connect to the VPN appliance in the on-premises network. The VPN virtual network gateway is configured to accept requests from the on-premises network only through the VPN appliance. For more information, see Connect an on-premises network to a Microsoft Azure virtual network.

VPN connection

The connection has properties that specify the connection type (IPSec) and the key shared with the on-premises VPN appliance to encrypt traffic.

Azure Virtual Network (VNet)

Each VNet resides in a single Azure region, and can host multiple application tiers. Application tiers can be segmented using subnets in each VNet.

Gateway subnet

The virtual network gateways are held in the same subnet.

Cloud application

The application hosted in Azure. It might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see Running Windows VM workloads and Running Linux VM workloads.

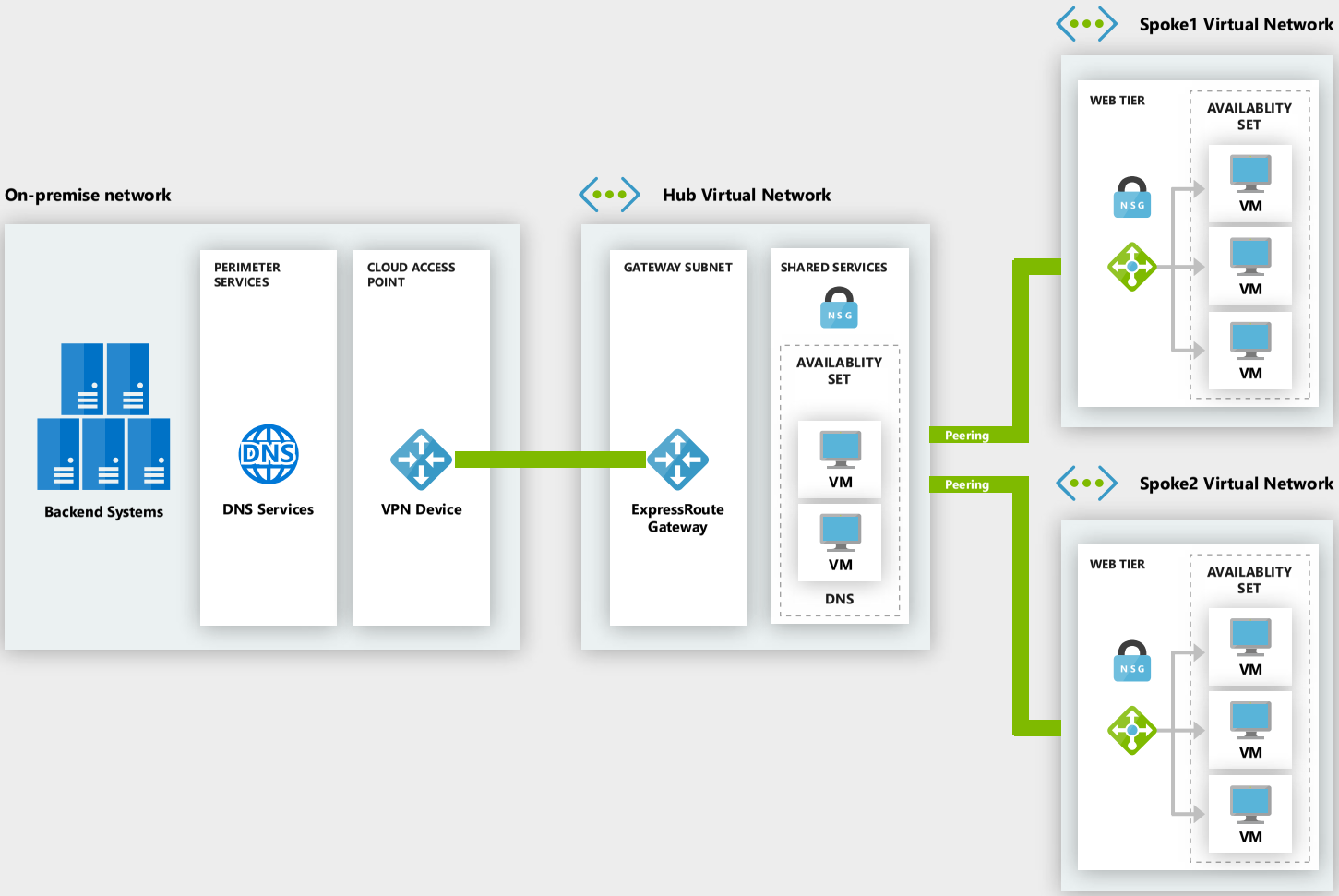
Microsoft

Implement a hub-spoke network topology in Azure

In this architecture, the hub is an Azure virtual network (VNet) that acts as a central point of connectivity to your on-premises network. The spokes are VNets that peer with the hub, and can be used to isolate workloads.

Reasons to consider this architecture :

Reduce costs by centralizing services shared services such as network virtual appliances (NVAs) and DNS servers.
Overcome subscriptions limits by peering VNets from different subscriptions to the central hub.
Separate concerns between central IT (SecOps, InfraOps) and workloads (DevOps).



Recommendations

- The hub VNet, and each spoke VNet, can be implemented in different resource groups, and even different subscriptions, as long as they belong to the same Azure Active Directory (Azure AD) tenant in the same Azure region. This allows for a decentralized management of each workload, while sharing services maintained in the hub VNet.
- A hub-spoke topology can be used without a gateway, if you don't need connectivity with your on-premises network.
- If you require connectivity between spokes, consider implementing an NVA for routing in the hub, and using user defined routes (UDRs) in the spoke to forward traffic to the hub.
- Make sure to consider the limitation on the number of VNet peerings per VNet in Azure. If you need more spokes than this limit, consider creating a hub-spoke-hub-spoke topology, where the first level of spokes also act as hubs.
- Consider what services are shared in the hub, to ensure the hub scales to the number of spokes.

Azure

Architecture Components

On-premises network

A private local-area network running within an organization.

VPN Device

A device or service that provides external connectivity to the on-premises network. The VPN device may be a hardware device, or a software solution such as the Routing and Remote Access Service (RRAS) in Windows Server 2012. For a list of supported VPN appliances and information on configuring selected VPN appliances for connecting to Azure, see About VPN devices for Site-to-Site VPN Gateway connections.

VPN virtual network gateway or ExpressRoute gateway

The virtual network gateway enables the VNet to connect to the VPN device, or ExpressRoute circuit, used for connectivity with your on-premises network. For more information, see Connect an on-premises network to a Microsoft Azure virtual network.

Note:

The deployment scripts for this reference architecture use a VPN gateway for connectivity, and a VNet in Azure to simulate your on-premises network.

Hub Vnet

Azure VNet used as the hub in the hub-spoke topology. The hub is the central point of connectivity to your on-premises network, and a place to host services that can be consumed by the different workloads hosted in the spoke VNets.

Gateway subnet

The virtual network gateways are held in the same subnet.

Shared services subnet

A subnet in the hub VNet used to host services that can be shared among all spokes, such as DNS or AD DS.

Spoke Vnets

One or more Azure VNets that are used as spokes in the hub-spoke topology. Spokes can be used to isolate workloads in their own VNets, managed separately from other spokes. Each workload might include multiple tiers, with multiple subnets connected through Azure load balancers. For more information about the application infrastructure, see Running Windows VM workloads and Running Linux VM workloads.

Vnet peering

Two VNets in the same Azure region can be connected using a peering connection. Peering connections are non-transitive, low latency connections between VNets. Once peered, the VNets exchange traffic by using the Azure backbone, without the need for a router. In a hub-spoke network topology, you use VNet peering to connect the hub to each spoke.

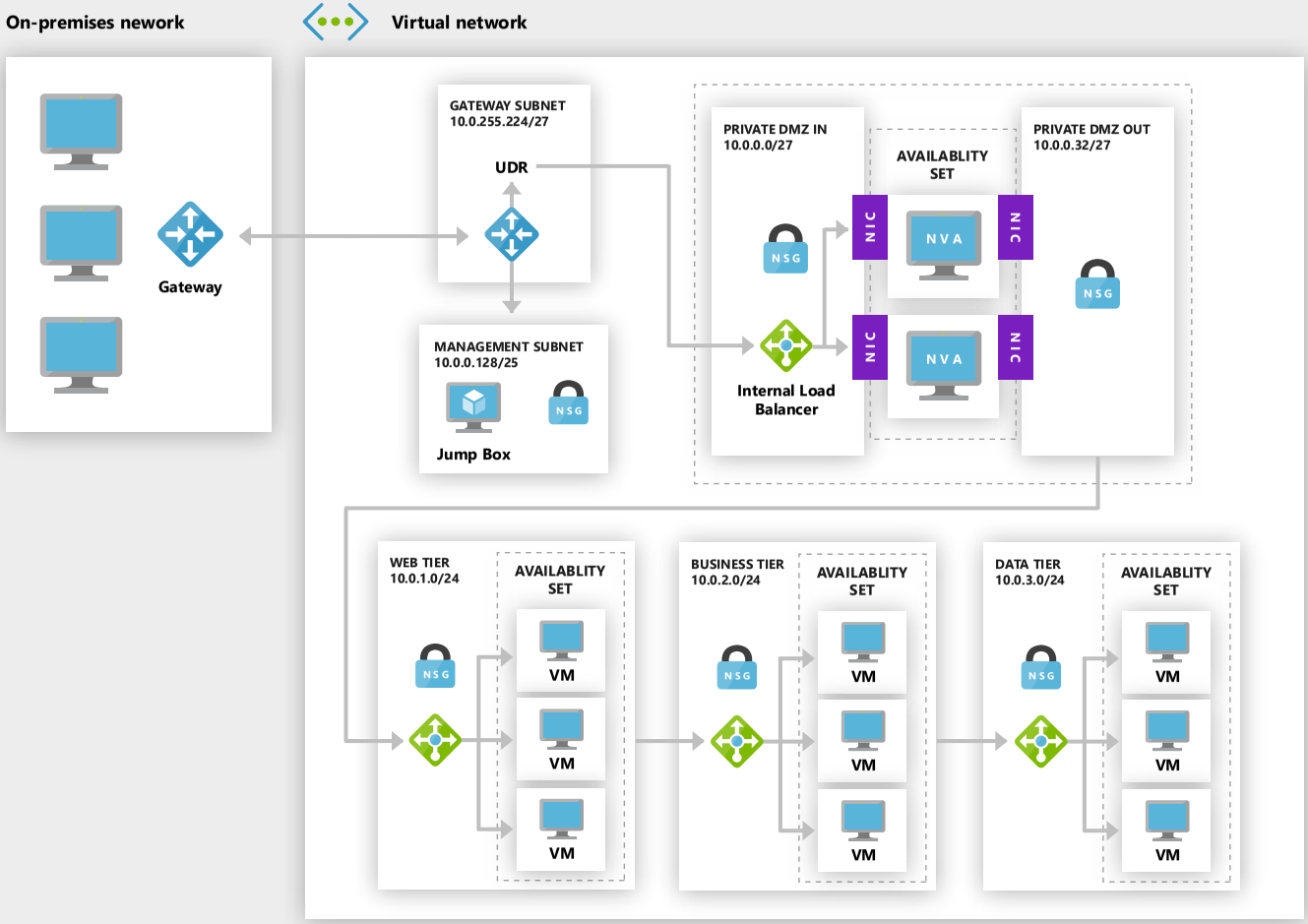
Réseau DMZ

Ces architectures de référence présentent des pratiques éprouvées permettant de créer un réseau DMZ qui protège la frontière entre un réseau virtuel Azure et un réseau sur site ou Internet.

Microsoft

DMZ between Azure and your on-premises datacenter

This architecture implements a DMZ (also called a perimeter network) between an on-premises network and an Azure virtual network. The DMZ includes highly available network virtual appliances (NVAs) to implement security functionality such as firewalls and packet inspection. All outgoing traffic from the VNet is force-tunneled to the Internet through the on-premises network, so that it can be audited.



Recommendations

- Use Role-Based Access Control (RBAC) to manage the resources in your application.
- On-premises traffic passes to the VNet through a virtual network gateway. We recommend an Azure VPN gateway or an Azure ExpressRoute gateway.
- Create a network security group (NSG) for the inbound NVA subnet, and only allow traffic originating from the on-premises network.
- Create NSGs for each subnet to provide a second level of protection in case of an incorrectly configured or disabled NVA.
- Force-tunnel all outbound Internet traffic through your on-premises network using the site-to-site VPN tunnel, and route to the Internet using network address translation (NAT).
- Don't completely block Internet traffic from the application tiers, as this will prevent these tiers from using Azure PaaS services that rely on public IP addresses, such as VM diagnostics logging.
- Perform all application and resource monitoring through the jumpbox in the management subnet.

Azure

Architecture Components

On-premise Nework

A private local-area network implemented in an organization.

Azure Virtual Network (VNet)

The VNet hosts the application and other resources running in Azure.

Gateway

The gateway provides connectivity between the routers in the on-premises network and the VNet.

Network Virtual Appliance (NVA)

NVA is a generic term that describes a VM performing tasks such as allowing or denying access as a firewall, optimizing wide area network (WAN) operations (including network compression), custom routing, or other network functionality.

Web Tier, Business Tier, and Data Tier Subnets

Subnets hosting the VMs and services that implement an example 3-tier application running in the cloud. See Running Windows VMs for an N-tier architecture on Azure for more information.

User Defined Routes

User defined routes define the flow of IP traffic within Azure VNets.

NOTE: Depending on the requirements of your VPN connection, you can configure Border Gateway Protocol (BGP) routes instead of using UDRs to implement the forwarding rules that direct traffic back through the on-premises network.

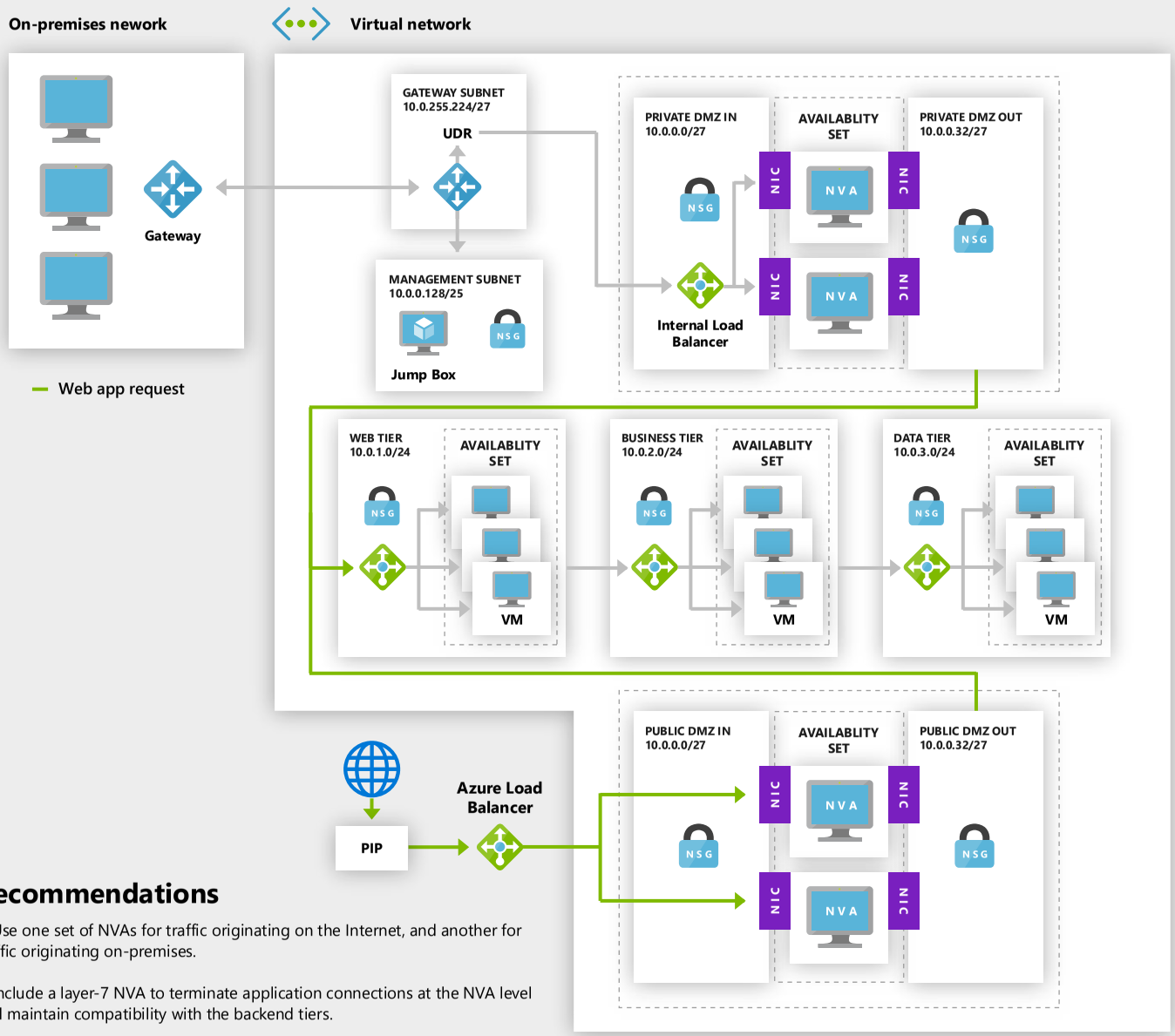
Management Subnet

This subnet contains VMs that implement management and monitoring capabilities for the components running in the VNet.

Microsoft

DMZ between Azure and the Internet

This architecture implements a DMZ (also called a perimeter network) that accepts Internet traffic to an Azure virtual network. It also includes a DMZ that handles traffic from an on-premises network. Network virtual appliances (NVAs) implement security functionality such as firewalls and packet inspection.



Recommendations

- Use one set of NVAs for traffic originating on the Internet, and another for traffic originating on-premises.
- Include a layer-7 NVA to terminate application connections at the NVA level and maintain compatibility with the backend tiers.
- For scalability and availability, deploy the public DMZ NVAs in an availability set with a load balancer to distribute requests across the NVAs.
- Even if your architecture initially requires a single NVA, we recommend putting a load balancer in front of the public DMZ from the beginning. That makes it easier to scale to multiple NVAs in the future.
- Log all incoming requests on all ports. Regularly audit the logs.

Azure

Architecture Components

Public IP Address (PIP)

The IP address of the public endpoint. External users connected to the Internet can access the system through this address.

Network Virtual Appliance (NVA)

This architecture includes a separate pool of NVAs for traffic originating on the Internet.

Azure Load Balancer

All incoming requests from the Internet pass through the load balancer and are distributed to the NVAs in the public DMZ.

Public DMZ Inbound Subnet

This subnet accepts requests from the Azure load balancer. Incoming requests are passed to one of the NVAs in the public DMZ.

Public DMZ Outbound Subnet

Requests that are approved by the NVA pass through this subnet to the internal load balancer for the web tier.

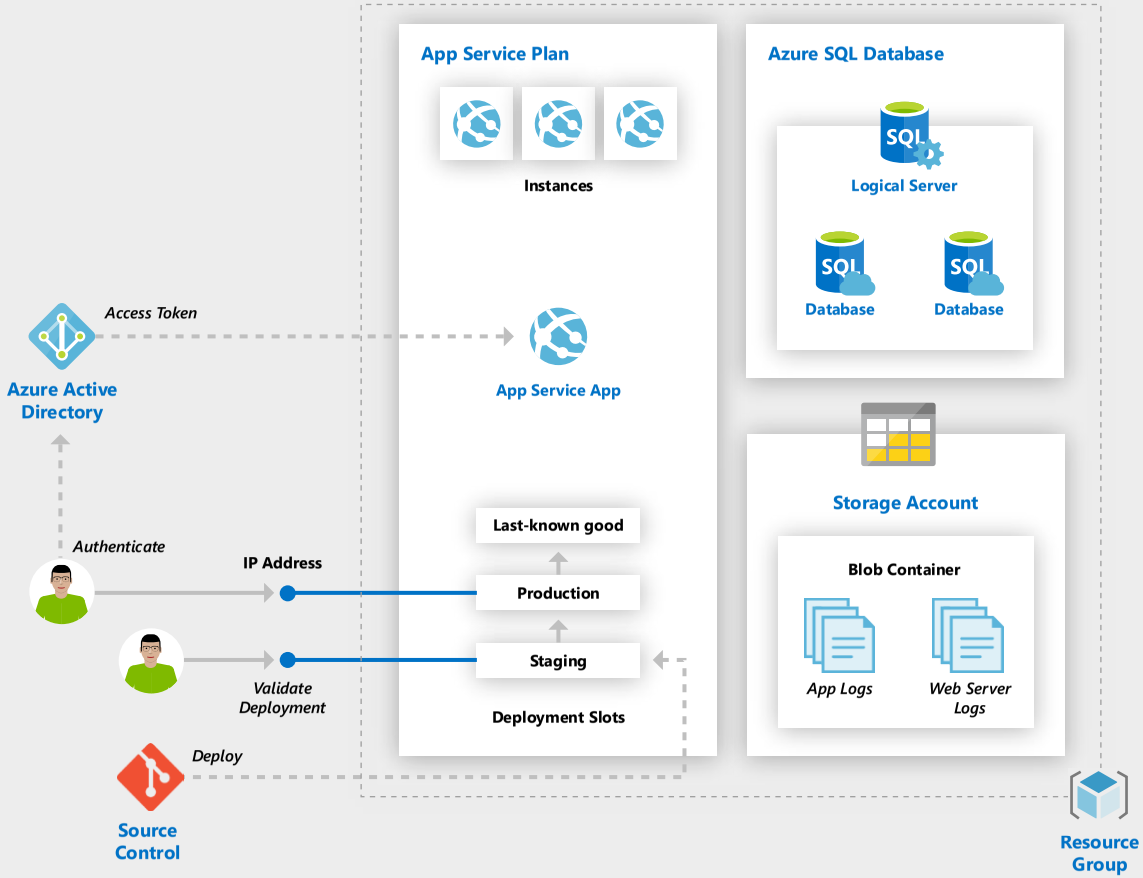
Application web gérée

Ces architectures de référence présentent des pratiques éprouvées quant aux applications web qui ont recours à Azure App Service et à d'autres services gérés dans Azure.

Microsoft

Basic web application

This architecture shows a baseline deployment for a web application that uses Azure App Service and Azure SQL Database.



Recommendations

- Use the Standard or Premium tiers, because they support scale out, autoscale, and secure sockets layer (SSL).
- Provision the App Service plan and the SQL Database in the same region to minimize network latency.
- Enable autoscaling. If your application has a predictable, regular workload, schedule the instance counts ahead of time. If the workload is not predictable, use rule-based autoscaling to react to changes in load.
- Create a staging slot to deploy updates. By using a staging slot, you can verify the deployment succeeded, before swapping it into production. Using a staging slot also ensures that all instances are warmed up before being swapped into production.
- Don't use slots on your production deployment for testing, because all apps within the same App Service plan share the same VM instances. Instead, put test deployments into a separate App Service plan to isolate them from the production version.
- Store configuration settings as app settings. Define the app settings in your Resource Manager templates, or using PowerShell. Never check passwords, access keys, or connection strings into source control. Instead, pass these as parameters to a deployment script that stores these values as app settings.
- Consider using App Service authentication to implement authentication with an identity provider such as Azure Active Directory, Facebook, Google, or Twitter.
- Use SQL Database point-in-time restore to recover from human error by returning the database to an earlier point in time. Use geo-restore to recover from a service outage by restoring a database from a geo-redundant backup.

Azure

Architecture Components

Resource Group

A resource group is a logical container for Azure resources.

App Service App

Azure App Service is a fully managed platform for creating and deploying cloud applications.

App Service Plan

An App Service plan provides the managed virtual machines (VMs) that host your app. All apps associated with a plan run on the same VM instances.

Deployment Slot

A deployment slot lets you stage a deployment and then swap it with the production deployment. That way, you avoid deploying directly into production. See the Manageability section for specific recommendations.

IP Address

The App Service app has a public IP address and a domain name. The domain name is a subdomain of azurewebsites.net, such as contoso.azurewebsites.net. To use a custom domain name, such as contoso.com, create domain name service (DNS) records that map the custom domain name to the IP address. For more information, see Configure a custom domain name in Azure App Service.

Azure SQL Database

SQL Database is a relational database-as-a-service in the cloud.

Logical Server

In Azure SQL Database, a logical server hosts your databases. You can create multiple databases per logical server.

Azure Storage

Create an Azure storage account with a blob container to store diagnostic logs.

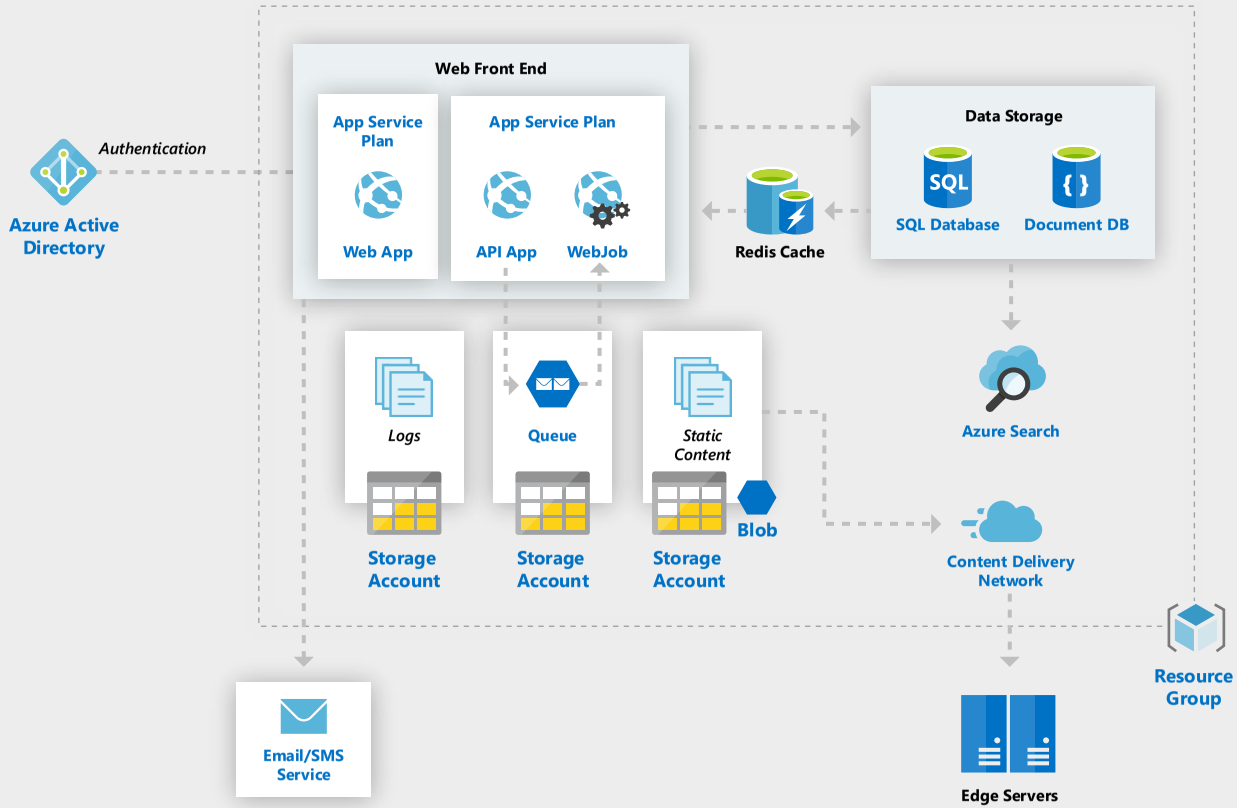
Azure Active Directory (Azure AD)

Use Azure AD or another identity provider for authentication.

Microsoft

Improved scalability in a web application

This architecture builds on the one shown in "Basic web application" and adds elements to improve scalability and performance.



Recommendations

- Use Azure WebJobs to run long-running tasks in the background. WebJobs can run on a schedule, continuously, or in response to a trigger, such as putting a message on a queue.
- Consider deploying resource intensive WebJobs to a separate App Service plan. This provides dedicated instances for the WebJob.
- Use Azure Redis Cache to cache semi-static transaction data, session state, and HTML output.
- Use Azure CDN to cache static content. The main benefit of a CDN is to reduce latency for users, because content is cached at an edge server that is geographically close to the user. CDN can also reduce load on the application, because that traffic is not being handled by the application.
- Choose a data store based on application requirements. Depending on the scenario, you might use multiple stores. For more guidance, see Choose the right data store.
- If you are using Azure SQL Database, consider using elastic pools. Elastic pools enable you to manage and scale multiple databases that have varying and unpredictable usage demands.
- Also consider using Elastic Database tools to shard the database. Sharding allows you to scale out the database horizontally.
- Use Transparent Data Encryption to encrypt data at rest in Azure SQL Database.

Azure

Architecture Components

Resource Group

A resource group is a logical container for Azure resources.

Web App and API App

A typical modern application might include both a website and one or more RESTful web APIs. A web API might be consumed by browser clients through AJAX, by native client applications, or by server-side applications. For considerations on designing web APIs, see API design guidance.

WebJob

Use Azure WebJobs to run long-running tasks in the background. WebJobs can run on a schedule, continously, or in response to a trigger, such as putting a message on a queue. A WebJob runs as a background process in the context of an App Service app.

Queue

In the architecture shown here, the application queues background tasks by putting a message onto an Azure Queue storage queue. The message triggers a function in the WebJob. Alternatively, you can use Service Bus queues. For a comparison, see Azure Queues and Service Bus queues - compared and contrasted.

Cache

Store semi-static data in Azure Redis Cache.

CDN

Use Azure Content Delivery Network (CDN) to cache publicly available content for lower latency and faster delivery of content.

Data Storage

Use Azure SQL Database for relational data. For non-relational data, consider a NoSQL store, such as Cosmos DB.

Azure Search

Use Azure Search to add search functionality such as search suggestions, fuzzy search, and language-specific search. Azure Search is typically used in conjunction with another data store, especially if the primary data store requires strict consistency. In this approach, store authoritative data in the other data store and the search index in Azure Search. Azure Search can also be used to consolidate a single search index from multiple data stores.

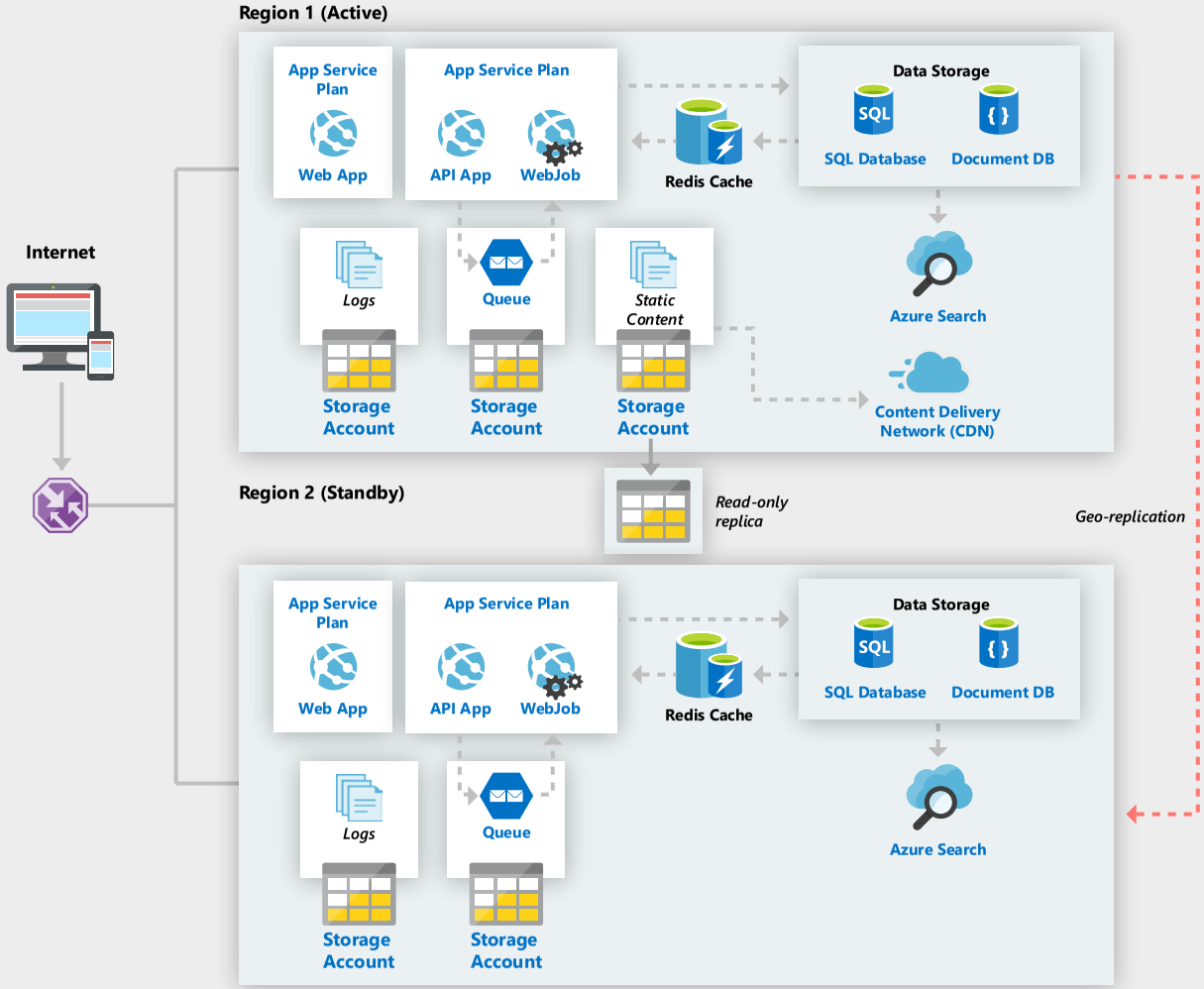
Email/SMS

Use a third-party service such as SendGrid or Twilio to send email or SMS messages instead of building this functionality directly into the application.

Microsoft

Run a web application in multiple regions

This architecture shows a web application running on Azure App Service in two regions to achieve high availability. If an outage occurs in the primary region, the application can fail over to the secondary region.



Recommendations

- Each Azure region is paired with another region within the same geography. In general, choose regions from the same regional pair. If there is a broad outage, recovery of at least one region out of every pair is prioritized.
- Configure Traffic Manager to use priority routing, which routes all requests to the primary region. If the primary region becomes unreachable, Traffic Manager automatically fails over to the secondary region.
- Traffic Manager uses an HTTP (or HTTPS) probe to monitor the availability of each region. Create a health probe endpoint that reports the overall health of the application.
- Traffic Manager is a possible failure point in the system. Review the Traffic Manager SLA, and determine whether using Traffic Manager alone meets your business requirements for high availability. If not, consider adding another traffic management solution as a failback.
- For Azure SQL Database, use Active Geo-Replication to create a readable secondary replica in a different region. Fail over to a secondary database if your primary database fails or needs to be taken offline.
- Cosmos DB also supports geo-replication across regions. One region is designated as writable and the others are read-only replicas. If there is a regional outage, you can fail over by selecting another region to be the write region.
- For Azure Storage, use read-access geo-redundant storage (RA-GRS).

Azure

Architecture Components

Primary and Secondary Regions

This architecture uses two regions to achieve higher availability. The application is deployed to each region. During normal operations, network traffic is routed to the primary region. If the primary region becomes unavailable, traffic is routed to the secondary region.

Azure Traffice Manager

Traffic Manager routes incoming requests to the primary region. If the application running that region becomes unavailable, Traffic Manager fails over to the secondary region.

Geo-replication

of SQL Database and Cosmos DB.

should content below be included?

A multi-region architecture can provide higher availability than deploying to a single region. If a regional outage affects the primary region, you can use Traffic Manager to fail over to the secondary region. This architecture can also help if an individual subsystem of the application fails.

There are several general approaches to achieving high availability across regions:

Active/passive with hot standby. Traffic goes to one region, while the other waits on hot standby. Hot standby means the VMs in the secondary region are allocated and running at all times.

Active/passive with cold standby. Traffic goes to one region, while the other waits on cold standby. Cold standby means the VMs in the secondary region are not allocated until needed for failover. This approach costs less to run, but will generally take longer to come online during a failure.

Active/active. Both regions are active, and requests are load balanced between them. If one region becomes unavailable, it is taken out of rotation.

This reference architecture focuses on active/passive with hot standby, using Traffic Manager for failover.

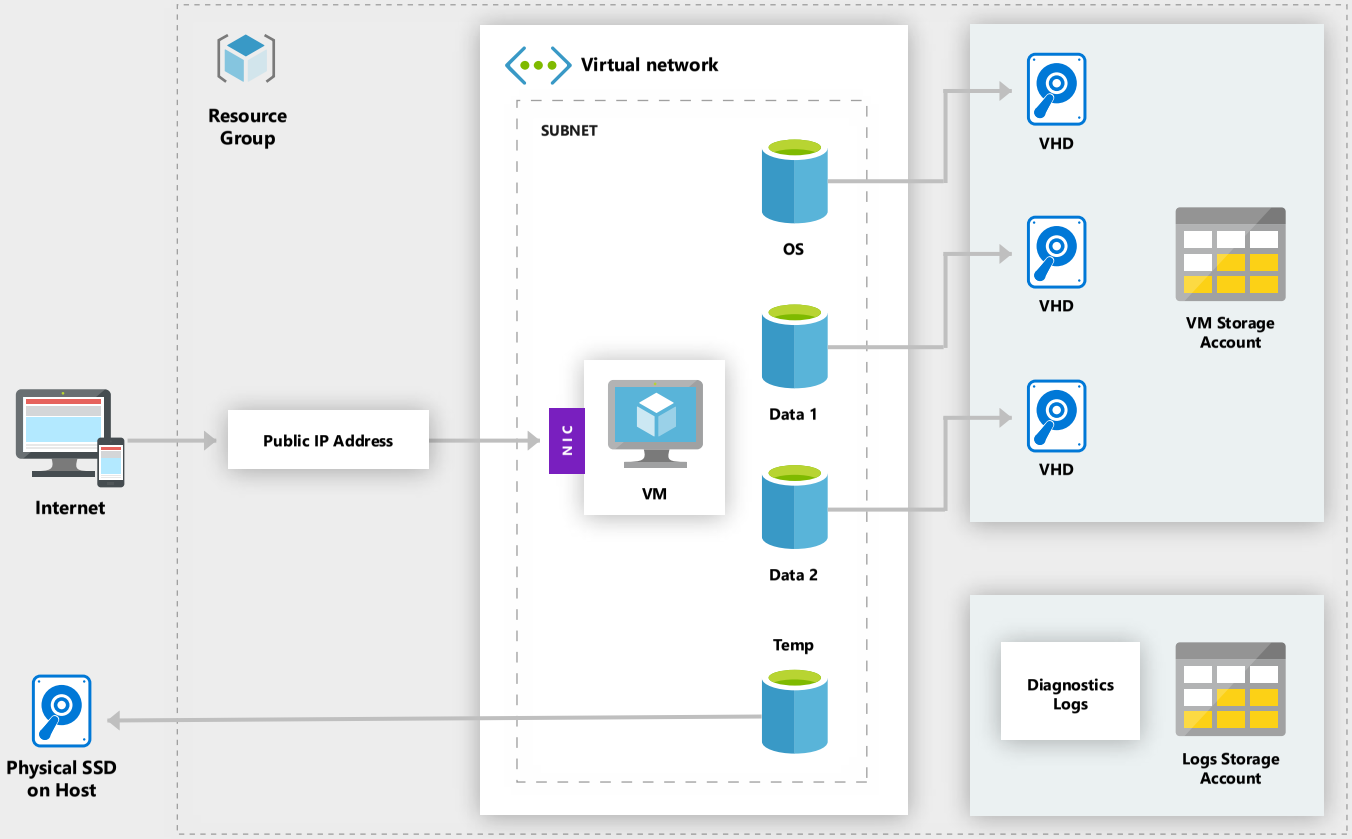
Exécution de scénarios d'usage de machine virtuelle Linux

Ces architectures de référence présentent des pratiques éprouvées permettant l'exécution de machines virtuelles Linux dans Azure.

Microsoft

Run a Linux VM on Azure

This architecture shows a Linux virtual machine (VM) running on Azure, along with associated networking and storage components. This architecture can be used to run a single instance, and is the basis for more complex architectures such as n-tier applications.



Recommendations

- For best disk I/O performance, we recommend Premium Storage, which stores data on solid-state drives (SSDs).
- Use Managed disks, which do not require a storage account. You simply specify the size and type of disk and it is deployed in a highly available way.
- Attach a data disk for persistent storage of application data.
- Enable monitoring and diagnostics, including health metrics, diagnostics infrastructure logs, and boot diagnostics.
- Add an NSG to the subnet to allow/deny network traffic to the subnet. To enable SSH, add a rule to the NSG that allows inbound traffic to TCP port 22.
- Reserve a static IP address if you need a fixed IP address that won't change — for example, if you need to create an A record in DNS, or need the IP address to be added to a safe list.
- For higher availability, deploy multiple VMs behind a load balancer. See [Load balanced VMs reference architecture]
- Use Azure Security Center to get a central view of the security state of your Azure resources. Security Center monitors potential security issues and provides a comprehensive picture of the security health of your deployment.
- Consider Azure Disk Encryption if you need to encrypt the OS and data disks.

Azure

Architecture Components

Resource Group

A resource group is a container that holds related resources. You usually create resource groups for different resources in a solution based on their lifetime, and who will manage the resources. For a single VM workload, you may create a single resource group for all resources.

VM

Azure supports running various popular Linux distributions, including CentOS, Debian, Red Hat Enterprise, Ubuntu, and FreeBSD. For more information, see Azure and Linux. You can provision a VM from a list of published images or from a virtual hard disk (VHD) file that you upload to Azure Blob storage.

OS disk

The OS disk is a VHD stored in Azure Storage. That means it persists even if the host machine goes down. The OS disk is /dev/sda1.

Temporary disk

The VM is created with a temporary disk. This disk is stored on a physical drive on the host machine. It is not saved in Azure Storage, and might be deleted during reboots and other VM lifecycle events. Use this disk only for temporary data, such as page or swap files. The temporary disk is /dev/sdb1 and is mounted at /mnt/resource or /mnt.

Data disks

A data disk is a persistent VHD used for application data. Data disks are stored in Azure Storage, like the OS disk.

Virtual network (VNet) and subnet

Every VM in Azure is deployed into a VNet that is further divided into subnets.

Public IP address

A public IP address is needed to communicate with the VM — for example over SSH.

Network interface (NIC)

The NIC enables the VM to communicate with the virtual network.

Network security group (NSG)

The NSG is used to allow/deny network traffic to the subnet. You can associate an NSG with an individual NIC or with a subnet. If you associate it with a subnet, the NSG rules apply to all VMs in that subnet.

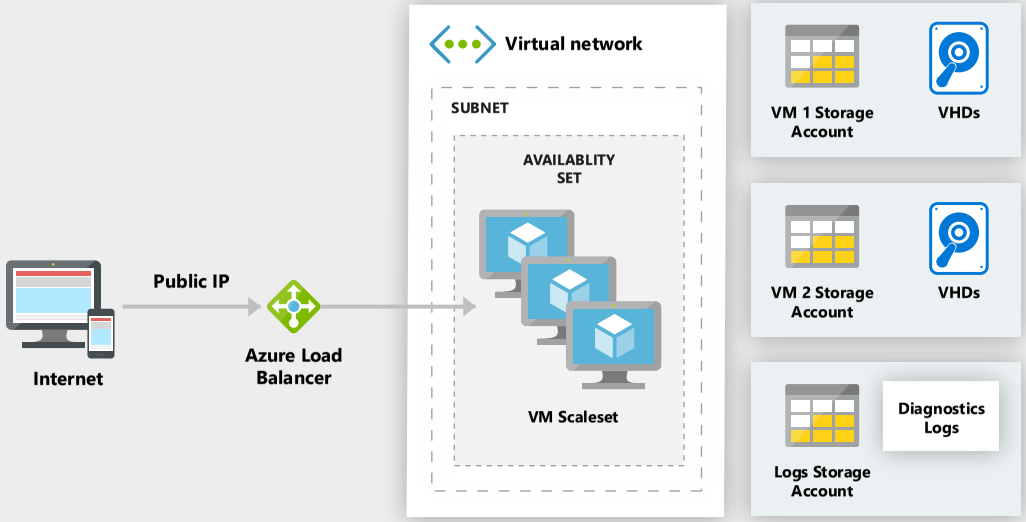
Diagnostics

Diagnostic logging is crucial for managing and troubleshooting the VM.

Microsoft

Run load-balanced VMs for scalability and availability

This architecture shows running several Linux virtual machines (VMs) running behind a load balancer, to improve availability and scalability. This architecture can be used for any stateless workload, such as a web server, and is a building block for deploying n-tier applications.



Recommendations

- Consider using a VM scale set if you need to quickly scale out VMs, or need to autoscale. If you don't use a scale set, place the VMs in an availability set.
- Use Managed disks, which do not require a storage account. You simply specify the size and type of disk and it is deployed in a highly available way.
- Place the VMs within the same subnet. Do not expose the VMs directly to the Internet, but instead give each VM a private IP address. Clients connect using the public IP address of the load balancer.
- For incoming Internet traffic, the load balancer rules define which traffic can reach the back end. However, load balancer rules don't support IP whitelisting, so if you want to add certain public IP addresses to a whitelist, add an NSG to the subnet.
- The load balancer uses health probes to monitor the availability of VM instances. If your VMs run an HTTP server, create an HTTP probe. Otherwise create a TCP probe.
- Virtual networks are a traffic isolation boundary in Azure. VMs in one VNet cannot communicate directly to VMs in a different VNet. VMs within the same VNet can communicate, unless you create network security groups (NSGs) to restrict traffic.

Azure

Architecture Components

Availability Set

The availability set contains the VMs, making the VMs eligible for the availability service level agreement (SLA) for Azure VMs. For the SLA to apply, the availability set must include a minimum of two VMs. Availability sets are implicit in scale sets. If you create VMs outside a scale set, you need to create the availability set independently.

Virtual Network (VNet) and Subnet

Every VM in Azure is deployed into a VNet that is further divided into subnets.

Azure Load Balancer

The load balancer distributes incoming Internet requests to the VM instances. The load balancer includes some related resources:

Public IP Address

A public IP address is needed for the load balancer to receive Internet traffic.

Front-end Configuration

Associates the public IP address with the load balancer.

Back-end Address Pool

Contains the network interfaces (NICs) for the VMs that will receive the incoming traffic.

Load Balancer Rules

Used to distribute network traffic among all the VMs in the back-end address pool.

Network Address Translation (NAT) Rules

Used to route traffic to a specific VM. For example, to enable remote desktop protocol (RDP) to the VMs, create a separate NAT rule for each VM.

Storage

If you are not using managed disks, storage accounts hold the VM images and other file-related resources, such as VM diagnostic data captured by Azure.

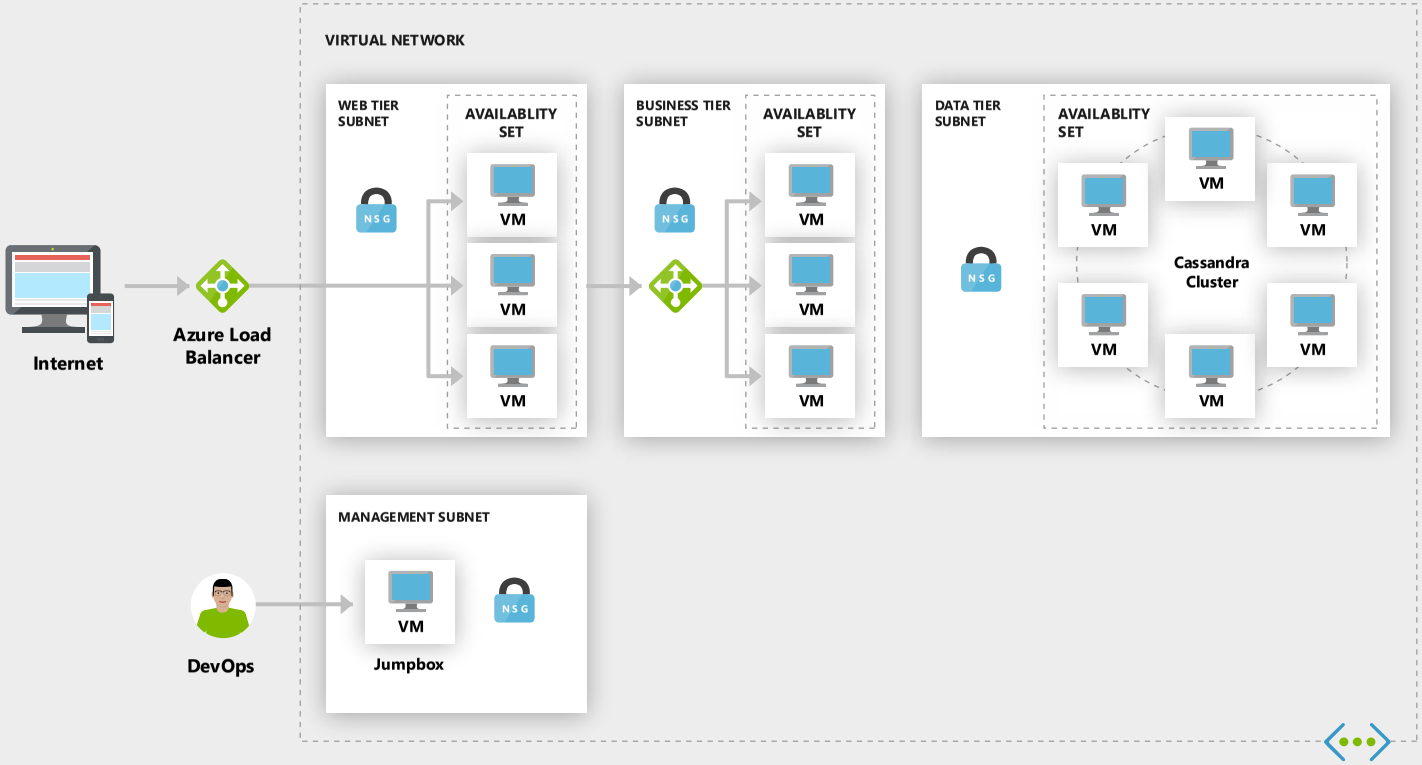
VM Scale set

A VM scale set is a set of identical VMs used to host a workload. Scale sets allow the number of VMs to be scaled in or out manually, or based on predefined rules.

Microsoft

Run Linux VMs for an N-tier Application

This architecture shows how to deploy Linux virtual machines (VMs) to run an N-tier application in Azure. For the data tier, this architecture shows Apache Cassandra, which provides replication and failover. However you could easily replace Cassandra in this architecture with another database, such as SQL Server.



Recommendations

- When you create the VNet, determine how many IP addresses your resources in each subnet require.
- Choose an address range that does not overlap with your on-premises network, in case you need to set up a gateway between the VNet and your on-premises network later.
- Design subnets with functionality and security requirements in mind. All VMs within the same tier or role should go into the same subnet, which can be a security boundary.
- Use NSG rules to restrict traffic between tiers. For example, in the 3-tier architecture shown above, the web tier should not communicate directly with the database tier.
- Do not allow SSH access from the public Internet to the VMs that run the application workload. Instead, all SSH access to these VMs must come through the jumpbox.
- The load balancers distribute network traffic to the web and business tiers. Scale horizontally by adding new VM instances. Note that you can scale the web and business tiers independently, based on load.
- At the database tier, having multiple VMs does not automatically translate into a highly available database. For a relational database, you will typically need to use replication and failover to achieve high availability.

Azure

Architecture Components

Availability Sets

Create an availability set for each tier, and provision at least two VMs in each tier. This makes the VMs eligible for a higher service level agreement (SLA) for VMs.

Subnets

Create a separate subnet for each tier. Specify the address range and subnet mask using CIDR notation.

Load Balancers

Use an Internet-facing load balancer to distribute incoming Internet traffic to the web tier, and an internal load balancer to distribute network traffic from the web tier to the business tier.

Jumpbox

Also called a bastion host. A secure VM on the network that administrators use to connect to the other VMs. The jumpbox has an NSG that allows remote traffic only from public IP addresses on a safe list. The NSG should permit secure shell (SSH) traffic.

Monitoring

Monitoring software such as Nagios, Zabbix, or Icinga can give you insight into response time, VM uptime, and the overall health of your system. Install the monitoring software on a VM that's placed in a separate management subnet.

NSGs

Use network security groups (NSGs) to restrict network traffic within the VNet. For example, in the 3-tier architecture shown here, the database tier does not accept traffic from the web front end, only from the business tier and the management subnet.

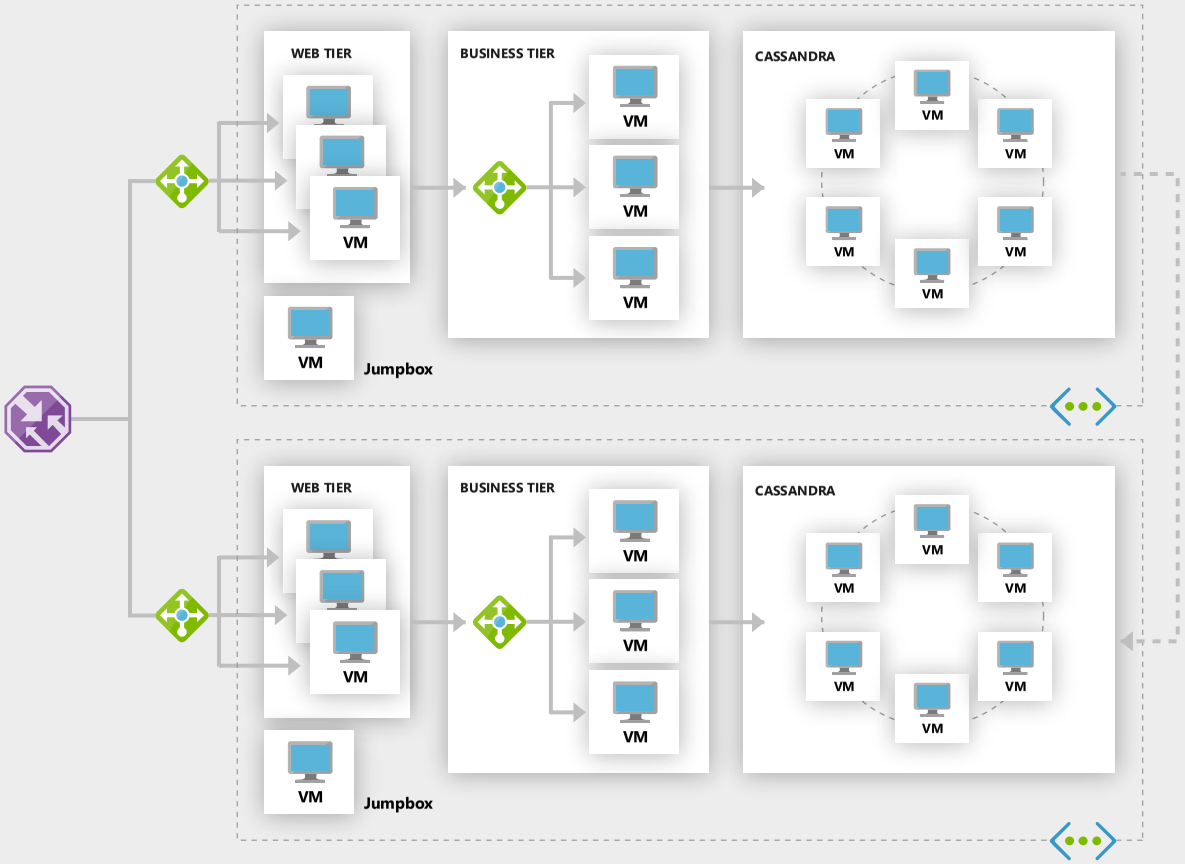
Apache Cassandra Database

Provides high availability at the data tier, by enabling replication and failover.

Microsoft

Run Linux VMs in multiple regions for high availability

This architecture shows an N-tier application deployed in two Azure regions. This architecture can provide higher availability than a single region. If an outage occurs in the primary region, the application can fail over to the secondary region. However, you must consider issues such as data replication and managing failover.



Recommendations

- Each Azure region is paired with another region within the same geography. In general, choose regions from the same regional pair. If there is a broad outage, recovery of at least one region out of every pair is prioritized.

- Configure Traffic Manager to use priority routing, which routes all requests to the primary region. If the primary region becomes unreachable, Traffic Manager automatically fails over to the secondary region.

- If Traffic Manager fails over, we recommend performing a manual failback rather than implementing an automatic failback. Verify that all application subsystems are healthy before failing back.

- Traffic Manager uses an HTTP (or HTTPS) probe to monitor the availability of each region. Create a health probe endpoint that reports the overall health of the application.

- Traffic Manager is a possible failure point in the system. Review the Traffic Manager SLA, and determine whether using Traffic Manager alone meets your business requirements for high availability. If not, consider adding another traffic management solution as a failback.

- For the data tier, this architecture shows Apache Cassandra for data replication and failover. Other database systems have similar functionality.

- When you update your deployment, update one region at a time to reduce the chance of a global failure from an incorrect configuration or an error in the application.

- Test the resiliency of the system to failures. Measure the recovery times and verify they meet your business requirements.

Azure

Architecture Components

Primary and Secondary Regions

Use two regions to achieve higher availability. One is the primary region.The other region is for failover.

Azure Traffic Manager

Traffic Manager routes incoming requests to one of the regions. During normal operations, it routes requests to the primary region. If that region becomes unavailable, Traffic Manager fails over to the secondary region. For more information, see the section Traffic Manager configuration.

Resource Group

Create separate resource groups for the primary region, the secondary region, and for Traffic Manager. This gives you the flexibility to manage each region as a single collection of resources. For example, you could redeploy one region, without taking down the other one. Link the resource groups, so that you can run a query to list all the resources for the application.

VNets

Create a separate VNet for each region. Make sure the address spaces do not overlap.

Apache Cassandra

Deploy Cassandra in data centers across Azure regions for high availability. Within each region, nodes are configured in rack-aware mode with fault and upgrade domains, for resiliency inside the region.

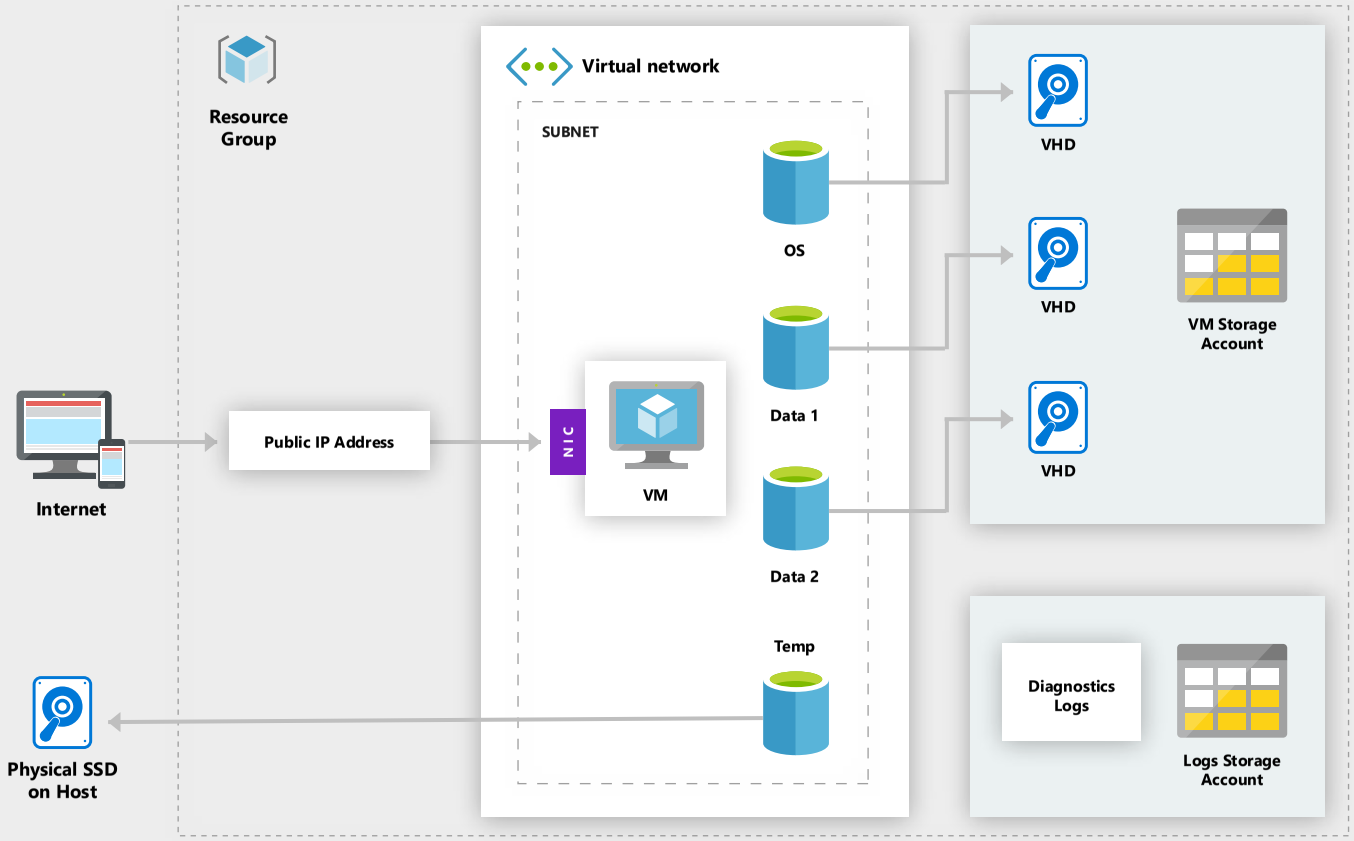
Exécution de scénarios d'usage de machine virtuelle Windows

Ces architectures de référence présentent des pratiques éprouvées permettant l'exécution de machines virtuelles Windows dans Azure.

Microsoft

Run a Windows VM on Azure

This architecture shows a Windows virtual machine (VM) running on Azure, along with associated networking and storage components. This architecture can be used to run a single instance, and is the basis for more complex architectures such as n-tier applications.



Recommendations

- For best disk I/O performance, we recommend Premium Storage, which stores data on solid-state drives (SSDs).
- Use Managed disks, which do not require a storage account. You simply specify the size and type of disk and it is deployed in a highly available way.
- Attach a data disk for persistent storage of application data.
- Enable monitoring and diagnostics, including health metrics, diagnostics infrastructure logs, and boot diagnostics.
- Add an NSG to the subnet to allow/deny network traffic to the subnet. To enable remote desktop (RDP), add a rule to the NSG that allows inbound traffic to TCP port 3389.
- Reserve a static IP address if you need a fixed IP address that won't change — for example, if you need to create an A record in DNS, or need the IP address to be added to a safe list.
- For higher availability, deploy multiple VMs behind a load balancer. See [Load balanced VMs reference architecture]
- Use Azure Security Center to get a central view of the security state of your Azure resources. Security Center monitors potential security issues and provides a comprehensive picture of the security health of your deployment.
- Consider Azure Disk Encryption if you need to encrypt the OS and data disks.

Azure

Architecture Components

Resource Group

A resource group is a container that holds related resources. Create a resource group to hold the resources for this VM.

VM

You can provision a VM from a list of published images or from a virtual hard disk (VHD) file that you upload to Azure Blob storage.

OS Disk

The OS disk is a VHD stored in Azure Storage. That means it persists even if the host machine goes down.

Temporary Disk

The VM is created with a temporary disk (the D: drive on Windows). This disk is stored on a physical drive on the host machine. It is not saved in Azure Storage, and might be deleted during reboots and other VM lifecycle events. Use this disk only for temporary data, such as page or swap files.

Data Disk

A data disk is a persistent VHD used for application data. Data disks are stored in Azure Storage, like the OS disk.

Virtual Network (VNet) and Subnet

Every VM in Azure is deployed into a VNet that is further divided into subnets.

Public IP Address

A public IP address is needed to communicate with the VM—for example over remote desktop (RDP).

Network interface (NIC)

The NIC enables the VM to communicate with the virtual network.

Network security group (NSG)

The NSG is used to allow/deny network traffic to the subnet. You can associate an NSG with an individual NIC or with a subnet. If you associate it with a subnet, the NSG rules apply to all VMs in that subnet.

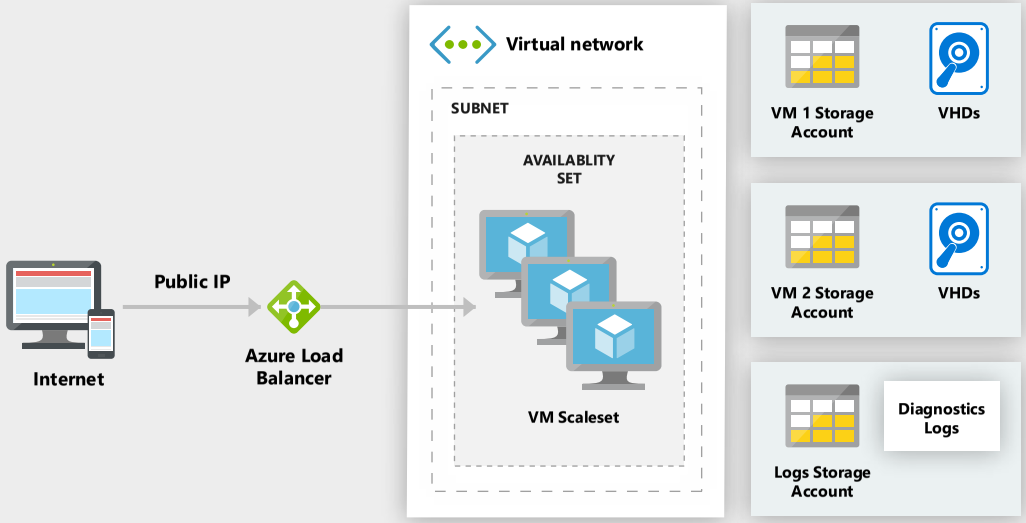
Diagnostics

Diagnostic logging is crucial for managing and troubleshooting the VM.

Microsoft

Run load-balanced VMs for scalability and availability

This architecture shows running several Windows virtual machines (VMs) running behind a load balancer, to improve availability and scalability. This architecture can be used for any stateless workload, such as a web server, and is a building block for deploying n-tier applications.



Recommendations

- Consider using a VM scale set if you need to quickly scale out VMs, or need to autoscale. If you don't use a scale set, place the VMs in an availability set.
- Use Managed disks, which do not require a storage account. You simply specify the size and type of disk and it is deployed in a highly available way.
- Place the VMs within the same subnet. Do not expose the VMs directly to the Internet, but instead give each VM a private IP address. Clients connect using the public IP address of the load balancer.
- For incoming Internet traffic, the load balancer rules define which traffic can reach the back end. However, load balancer rules don't support IP whitelisting, so if you want to add certain public IP addresses to a whitelist, add an NSG to the subnet.
- The load balancer uses health probes to monitor the availability of VM instances. If your VMs run an HTTP server, create an HTTP probe. Otherwise create a TCP probe.
- Virtual networks are a traffic isolation boundary in Azure. VMs in one VNet cannot communicate directly to VMs in a different VNet. VMs within the same VNet can communicate, unless you create network security groups (NSGs) to restrict traffic.

Azure

Architecture Components

Resource group

Resource groups are used to group resources so they can be managed by lifetime, owner, and other criteria.

Virtual Network (VNet) and Subnet

Every VM in Azure is deployed into a VNet that is further divided into subnets.

Azure Load Balancer

The load balancer distributes incoming Internet requests to the VM instances. The load balancer includes some related resources:

Public IP Address

A public IP address is needed for the load balancer to receive Internet traffic.

Front-end Configuration

Associates the public IP address with the load balancer.

Back-end Address Pool

Contains the network interfaces (NICs) for the VMs that will receive the incoming traffic.

Load Balancer Rules

Used to distribute network traffic among all the VMs in the back-end address pool.

VM Scale set

A VM scale set is a set of identical VMs used to host a workload. Scale sets allow the number of VMs to be scaled in or out manually, or based on predefined rules.

Availability Set

The availability set contains the VMs, making the VMs eligible for the availability service level agreement (SLA) for Azure VMs. In order for the SLA to apply, the availability set must include a minimum of two VMs. Availability sets are implicit in scale sets. If you create VMs outside a scale set, you need to create the availability set independently.

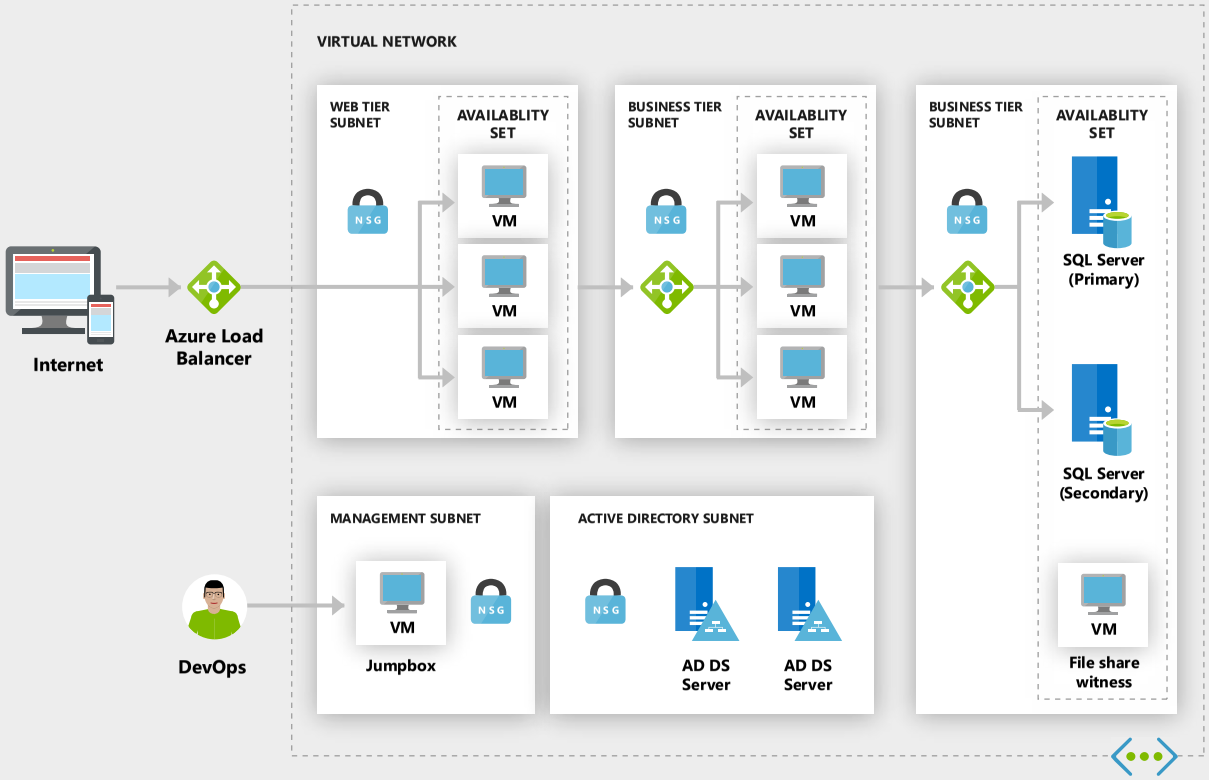
Storage

If you are not using managed disks, storage accounts hold the VM images and other file-related resources, such as VM diagnostic data captured by Azure.

Microsoft

Run Windows VMs for an N-tier application

This architecture shows how to deploy Windows virtual machines (VMs) to run an N-tier application in Azure. For the data tier, this architecture uses SQL Server Always On Availability Groups, which provide replication and failover.



Recommendations

- When you create the VNet, determine how many IP addresses your resources in each subnet require.
- Choose an address range that does not overlap with your on-premises network, in case you need to set up a gateway between the VNet and your on-premises network later.
- Design subnets with functionality and security requirements in mind. All VMs within the same tier or role should go into the same subnet, which can be a security boundary.
- Use NSG rules to restrict traffic between tiers. For example, in the 3-tier architecture shown above, the web tier should not communicate directly with the database tier.
- Do not allow remote desktop (RDP) access from the public Internet to the VMs that run the application workload. Instead, all RDP access to these VMs must come through the jumpbox.
- The load balancers distribute network traffic to the web and business tiers. Scale horizontally by adding new VM instances. Note that you can scale the web and business tiers independently, based on load.
- We recommend Always On Availability Groups for SQL Server high availability. When a SQL client tries to connect, the load balancer routes the connection request to the primary replica. If there is a failover to another replica, the load balancer automatically starts routing requests to the new primary replica.

Azure

Architecture Components

Availability Sets

Create an availability set for each tier, and provision at least two VMs in each tier. This makes the VMs eligible for a higher service level agreement (SLA) for VMs.

Subnets

Create a separate subnet for each tier. Specify the address range and subnet mask using CIDR notation.

Load Balancers

Use an Internet-facing load balancer to distribute incoming Internet traffic to the web tier, and an internal load balancer to distribute network traffic from the web tier to the business tier.

Jumpbox

Also called a bastion host. A secure VM on the network that administrators use to connect to the other VMs. The jumpbox has an NSG that allows remote traffic only from public IP addresses on a safe list. The NSG should permit remote desktop (RDP) traffic.

Monitoring

Monitoring software such as Nagios, Zabbix, or Icinga can give you insight into response time, VM uptime, and the overall health of your system. Install the monitoring software on a VM that's placed in a separate management subnet.

NSGs

Use network security groups (NSGs) to restrict network traffic within the VNet. For example, in the 3-tier architecture shown here, the database tier does not accept traffic from the web front end, only from the business tier and the management subnet.

SQL Server Always On Availability Group

Provides high availability at the data tier, by enabling replication and failover.

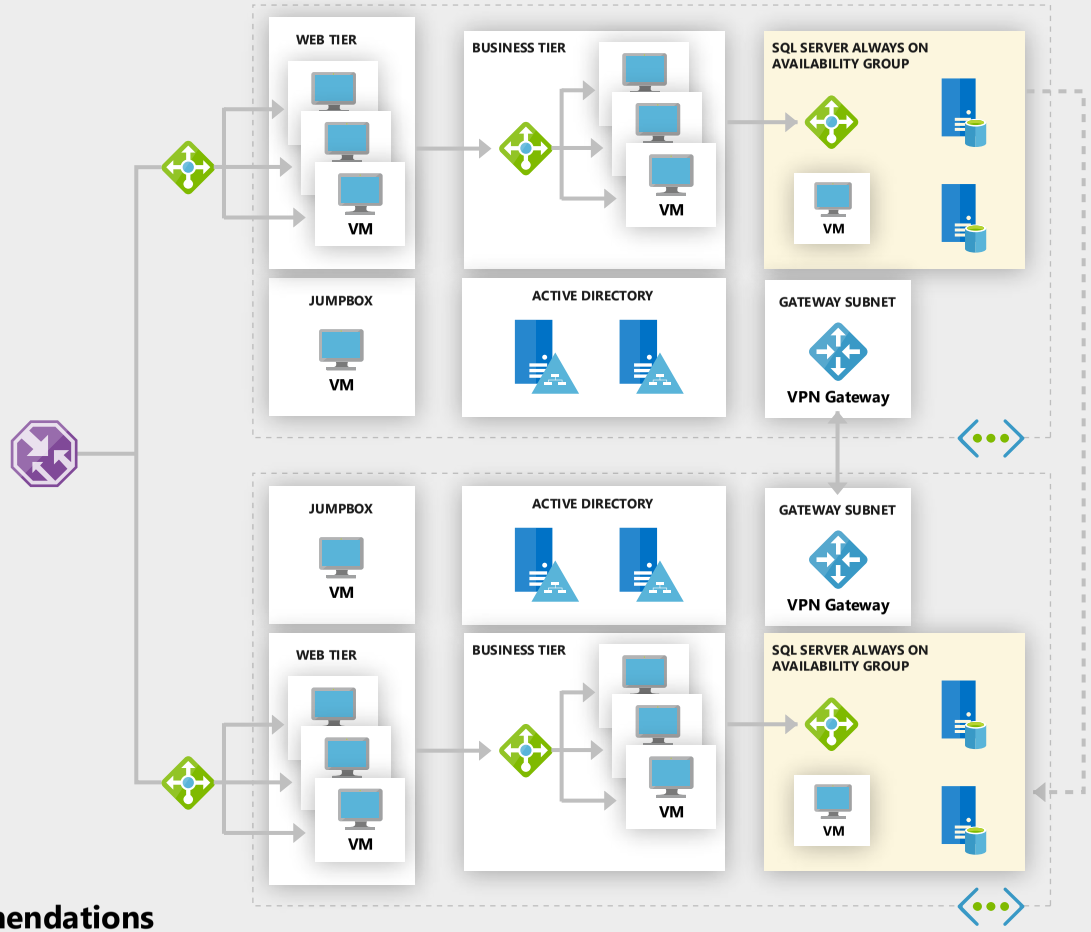
Active Directory Domain Services (AD DS) Servers

Prior to Windows Server 2016, SQL Server Always On Availability Groups must be joined to a domain. This is because Availability Groups depend on Windows Server Failover Cluster (WSFC) technology. Windows Server 2016 introduces the ability to create a Failover Cluster without Active Directory, in which case the AD DS servers are not required for this architecture. For more information, see What's new in Failover Clustering in Windows Server 2016.

Microsoft

Run Windows VMs in multiple regions for high availability

This architecture shows an N-tier application deployed in two Azure regions. This architecture can provide higher availability than a single region. If an outage occurs in the primary region, the application can fail over to the secondary region. However, you must consider issues such as data replication and managing failover.



Recommendations

- Each Azure region is paired with another region within the same geography. In general, choose regions from the same regional pair. If there is a broad outage, recovery of at least one region out of every pair is prioritized.
- Configure Traffic Manager to use priority routing, which routes all requests to the primary region. If the primary region becomes unreachable, Traffic Manager automatically fails over to the secondary region.
- If Traffic Manager fails over, we recommend performing a manual failback rather than implementing an automatic failback. Verify that all application subsystems are healthy before failing back.
- Traffic Manager uses an HTTP (or HTTPS) probe to monitor the availability of each region. Create a health probe endpoint that reports the overall health of the application.
- Traffic Manager is a possible failure point in the system. Review the Traffic Manager SLA, and determine whether using Traffic Manager alone meets your business requirements for high availability. If not, consider adding another traffic management solution as a failback.
- Create a SQL Server Always On Availability Group that includes the SQL Server instances in both the primary and secondary regions. Configure the replicas in the secondary region to use asynchronous commit, for performance reasons.
- If all of the SQL Server database replicas in the primary region fail, you can manually fail over the availability group. With forced failover, there is a risk of data loss. Once the primary region is back online, take a snapshot of the database and use tablediff to find the differences.
- When you update your deployment, update one region at a time to reduce the chance of a global failure from an incorrect configuration or an error in the application.
- Test the resiliency of the system to failures. Measure the recovery times and verify they meet your business requirements.

Azure

Architecture Components

Primary and Secondary Regions

Use two regions to achieve higher availability. One is the primary region. The other region is for failover.

Azure Traffic Manager

Traffic Manager routes incoming requests to one of the regions. During normal operations, it routes requests to the primary region. If that region becomes unavailable, Traffic Manager fails over to the secondary region. For more information, see the section Traffic Manager configuration.

Resource Groups

Create separate resource groups for the primary region, the secondary region, and for Traffic Manager. This gives you the flexibility to manage each region as a single collection of resources. For example, you could redeploy one region, without taking down the other one. Link the resource groups, so that you can run a query to list all the resources for the application.

VNets

Create a separate VNet for each region. Make sure the address spaces do not overlap.

SQL Server Always On Availability Group

If you are using SQL Server, we recommend SQL Always On Availability Groups for high availability. Create a single availability group that includes the SQL Server instances in both regions.

Note

Also consider Azure SQL Database, which provides a relational database as a cloud service. With SQL Database, you don't need to configure an availability group or manage failover.

VPN Gateways

Create a VPN gateway in each VNet, and configure a VNet-to-VNet connection, to enable network traffic between the two VNets. This is required for the SQL Always On Availability Group.