https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41

Architecture de microservice et ses 10 modèles de conception les plus importants

Architecture de microservice, base de données par microservice, sourcing d'événements, CQRS, Saga, BFF, API Gateway, étrangleur, disjoncteur, configuration d'externalisation, test de contrat axé sur le consommateur

S'attaquer à la complexité des grands systèmes logiciels a toujours été une tâche ardue depuis les débuts du développement logiciel (années 1960). Au fil des ans, les ingénieurs et architectes logiciels ont fait de nombreuses tentatives pour s'attaquer à la complexité des systèmes logiciels: **modularité et masquage de l'information** de ***David Parnas*** (1972), **Separation of Concern** par ***Edsger W.Dijkstra*** (1974) **, Service Oriented Architecture** (1998).

Tous ont utilisé la technique séculaire et éprouvée pour s'attaquer à la complexité d'un grand système: **diviser et conquérir** . Depuis les années 2010, ces techniques se sont avérées insuffisantes pour s'attaquer aux complexités des applications Web-Scale ou des applications d'entreprise modernes à grande échelle. En conséquence, les architectes et les ingénieurs ont développé une nouvelle approche pour s'attaquer à la complexité des systèmes logiciels des temps modernes: l' **architecture de microservices** . Il utilise également la même vieille technique «Divide and Conquer», quoique d'une manière nouvelle.

Les modèles de conception de logiciels sont des solutions générales et réutilisables au problème courant de la conception de logiciels. Les Design Patterns nous aident à partager un vocabulaire commun et à utiliser une solution testée au combat au lieu de réinventer la roue. Dans un article précédent: **Microservices efficaces: 10 meilleures pratiques** , j'ai décrit un ensemble de meilleures pratiques pour développer des microservices efficaces. Ici, je vais décrire un ensemble de modèles de conception pour vous aider à mettre en œuvre ces meilleures pratiques. Si vous êtes nouveau dans l'architecture de microservice, alors ne vous inquiétez pas, je vais vous présenter l'architecture de microservice.

En lisant cet article, vous apprendrez:

- Architecture de microservices
- Avantages de l'architecture microservice
- Inconvénients de l'architecture de microservices
- Quand utiliser l'architecture de microservice
- Les modèles de conception d'architecture de microservice les plus importants, y compris leurs avantages, inconvénients, cas d'utilisation, contexte, exemple de pile technique et ressources utiles.

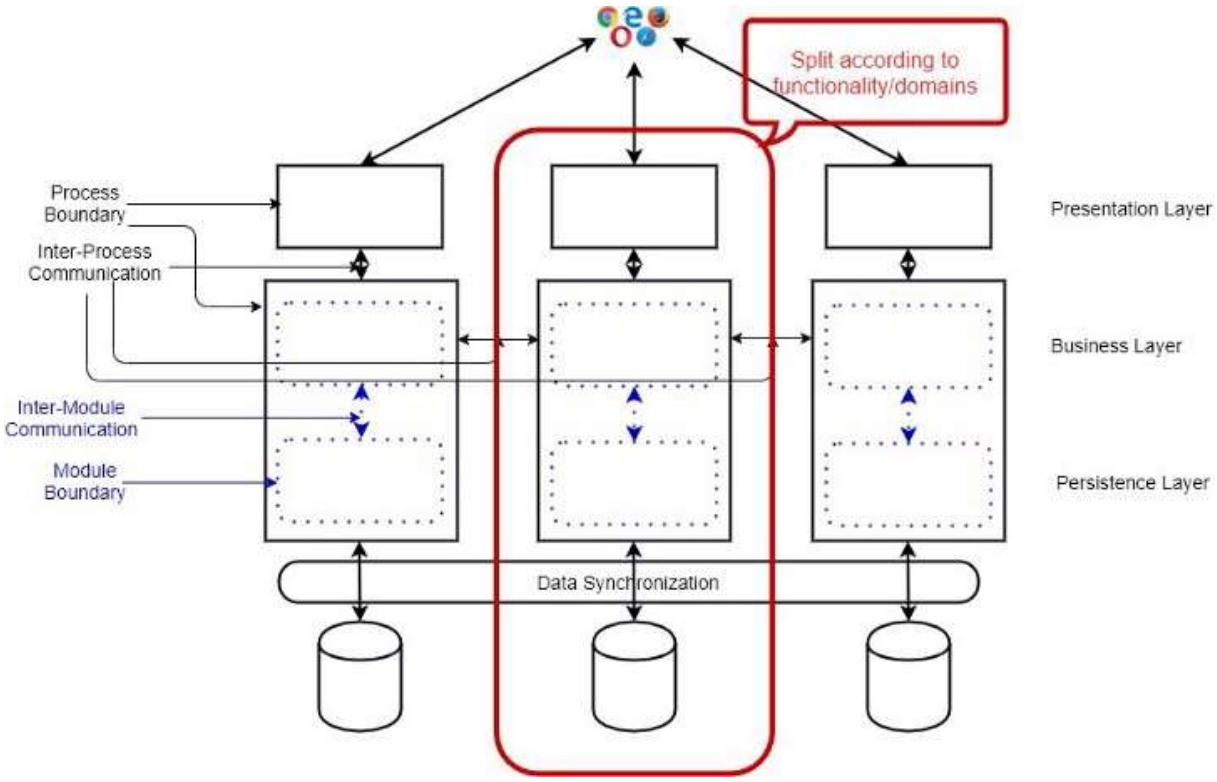
Architecture de microservices

J'ai couvert l'architecture de microservices en détail dans mes précédents articles de blog: **Architecture de microservices: un bref aperçu et pourquoi vous devriez l'utiliser dans votre prochain projet** et l' **architecture logicielle monolithique modulaire est-elle vraiment morte? .** Si vous êtes intéressé, vous pouvez les lire pour les approfondir.

Qu'est-ce qu'une architecture de microservices. Il existe de nombreuses définitions de l'architecture de microservice. Voici ma définition :

L'architecture de microservices consiste à diviser verticalement (par exigences fonctionnelles ou métier) un grand système complexe en sous-systèmes plus petits qui sont des processus (donc déployables indépendamment) et ces sous-systèmes communiquent entre eux via des appels réseau légers et indépendants de la langue, soit synchrones. (par exemple REST, gRPC) ou asynchrone (via Messagerie).

Voici la vue des composants d'une application Web d'entreprise avec une architecture de microservice:



Architecture des microservices par Md Kamaruzzaman

Caractéristiques importantes de l'architecture de microservices:

- L'ensemble de l'application est divisé en processus séparés où chaque processus peut contenir plusieurs modules internes.
- Contrairement aux monolithes modulaires ou SOA, une application Microservice est divisée verticalement (en fonction de la capacité ou des domaines de l'entreprise)
- La limite du microservice est externe. En conséquence, Microservices communique entre eux via des appels réseau (RPC ou message).
- Les microservices étant des processus indépendants, ils peuvent être déployés indépendamment.
- Ils communiquent de manière légère et n'ont besoin d'aucun canal de communication intelligent.

- Meilleure mise à l'échelle du développement.
- Vitesse de développement plus élevée.
- Prend en charge la modernisation itérative ou incrémentielle.
- Profitez de l'écosystème de développement logiciel moderne (Cloud, conteneurs, DevOps, sans serveur).
- Prend en charge la mise à l'échelle horizontale et la mise à l'échelle granulaire.
- Il met une faible complexité cognitive sur la tête du développeur grâce à sa plus petite taille.

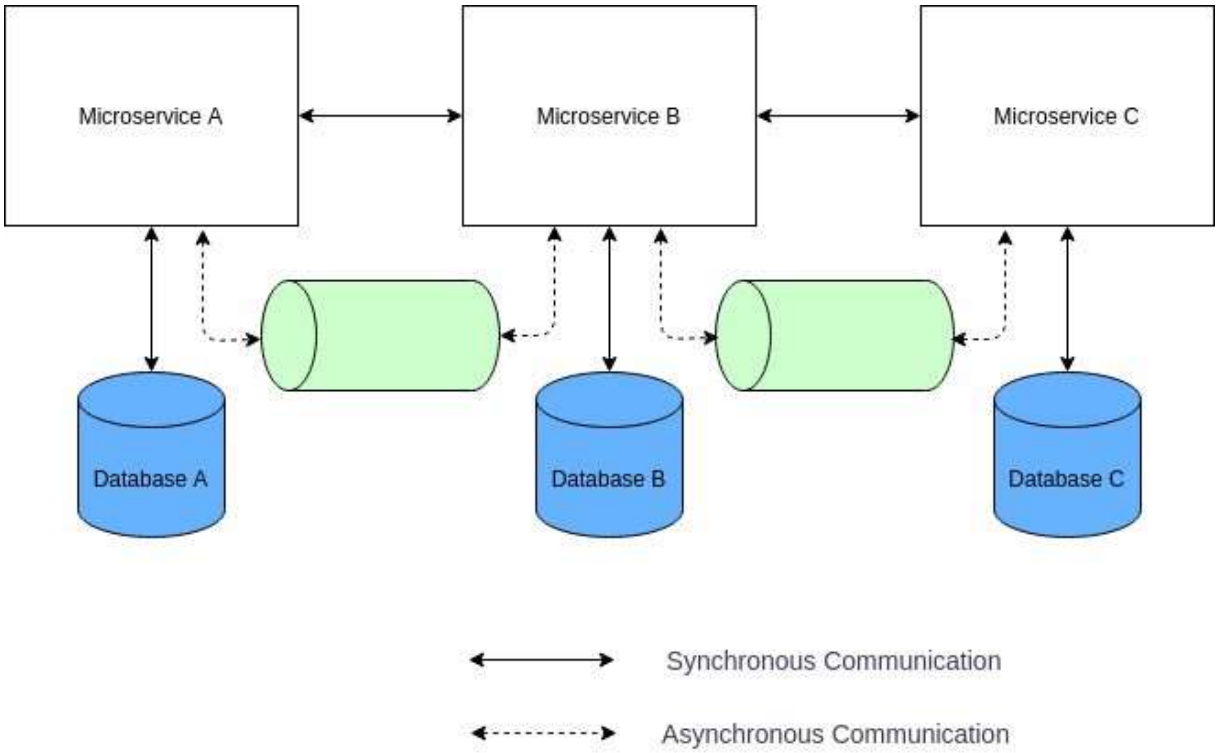
- Un plus grand nombre de pièces mobiles (services, bases de données, processus, conteneurs, cadres).
- La complexité passe du code à l'infrastructure.
- La prolifération des appels RPC et du trafic réseau.
- La gestion de la sécurité du système complet est un défi.
- La conception de l'ensemble du système est plus difficile.
- Présentez les complexités des systèmes distribués.

- Développement d'applications à l'échelle du Web.
- Développement d'applications d'entreprise lorsque plusieurs équipes travaillent sur l'application.
- Le gain à long terme est préféré au gain à court terme.
- L'équipe compte des architectes logiciels ou des ingénieurs seniors capables de concevoir une architecture de microservices.

Base de données par microservice

Une fois qu'une entreprise remplace le grand système monolithique par de nombreux microservices plus petits, la décision la plus importante à laquelle elle doit faire face concerne la base de données. Dans une architecture monolithique, une grande base de données centrale est utilisée. De nombreux architectes préfèrent conserver la base de données telle quelle, même lorsqu'ils passent à l'architecture de microservices. Bien que cela présente des avantages à court terme, il s'agit d'un anti-modèle, en particulier dans un système à grande échelle, car les microservices seront étroitement couplés dans la couche de base de données. L'objet entier du passage au microservice échouera (par exemple, l'autonomisation de l'équipe, le développement indépendant).

Une meilleure approche consiste à fournir à chaque microservice son propre magasin de données, afin qu'il n'y ait pas de couplage fort entre les services dans la couche de base de données. Ici, j'utilise le terme base de données pour montrer une séparation logique des données, c'est-à-dire que les microservices peuvent partager la même base de données physique, mais ils devraient utiliser un schéma / une collection / une table séparés. Il garantira également que les microservices sont correctement séparés conformément à la conception pilotée par le domaine .



Base de données par microservice par Md Kamaruzzaman

Avantages

- Propriété complète des données sur un service.
- Couplage lâche entre les équipes développant les services.

- Le partage de données entre les services devient difficile.
- Offrir une garantie transactionnelle ACID à l'échelle de l'application devient beaucoup plus difficile.
- La décomposition de la base de données Monolith en pièces plus petites nécessite une conception minutieuse et est une tâche difficile.

- Dans les applications d'entreprise à grande échelle.
- Lorsque l'équipe a besoin de s'approprier complètement ses microservices pour la mise à l'échelle du développement et la vitesse de développement.

- Dans des applications à petite échelle.
- Si une équipe développe tous les microservices.

Toutes les bases de données SQL, NoSQL offrent une séparation logique des données (par exemple, des tables séparées, des collections, des schémas, des bases de données).

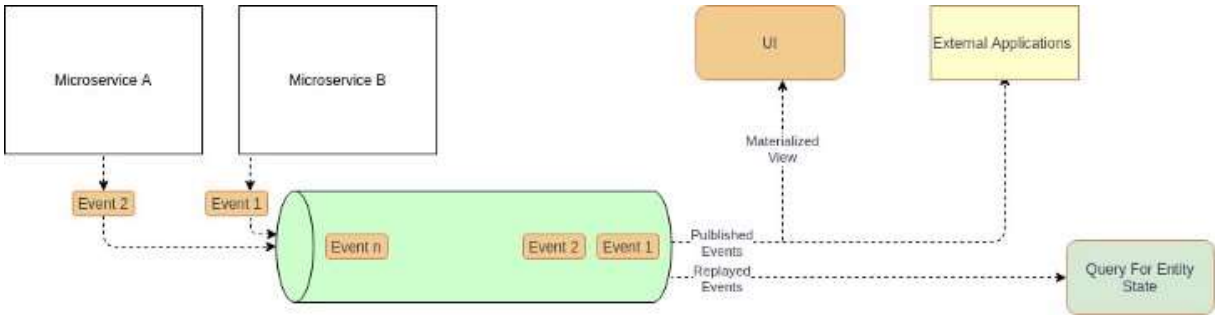
Lectures complémentaires

Sourcing événementiel

Dans une architecture de microservice, en particulier avec une **base** de **données par microservice,** les microservices ont besoin d'échanger des données. Pour les systèmes résilients, hautement évolutifs et tolérants aux pannes, ils doivent communiquer de manière asynchrone en échangeant des événements. Dans un tel cas, vous voudrez peut-être avoir des opérations atomiques, par exemple, mettre à jour la base de données et envoyer le message. Si vous disposez de bases de données SQL et souhaitez disposer de transactions distribuées pour un volume élevé de données, vous ne pouvez pas utiliser le verrouillage en deux phases (2PL)

car il ne se met pas à l'échelle. Si vous utilisez des bases de données NoSQL et souhaitez disposer d'une transaction distribuée, vous ne pouvez pas utiliser 2PL car de nombreuses bases de données NoSQL ne prennent pas en charge le verrouillage en deux phases.

Dans de tels scénarios, utilisez l'architecture basée sur les événements avec l'approvisionnement d'événements. Dans les bases de données traditionnelles, l'entité commerciale avec «l'état» actuel est directement stockée. Dans Event Sourcing, tout événement de changement d'état ou d'autres événements importants sont stockés à la place des entités. Cela signifie que les modifications d'une entité commerciale sont enregistrées sous la forme d'une série d'événements immuables. L'état d'une entité commerciale est déduit en retraitant tous les événements de cette entité commerciale à un moment donné. Étant donné que les données sont stockées sous la forme d'une série d'événements plutôt que via des mises à jour directes des magasins de données, divers services peuvent rejouer des événements à partir du magasin d'événements pour calculer l'état approprié de leurs magasins de données respectifs.



Sourcing événementiel par Md Kamaruzzaman

Avantages

- Fournissez l'atomicité aux systèmes hautement évolutifs.
- Historique automatique des entités, y compris la fonctionnalité de voyage dans le temps.
- Microservices faiblement couplés et pilotés par les événements.

- La lecture des entités à partir du magasin d'événements devient difficile et nécessite généralement un magasin de données supplémentaire (modèle **CQRS**)
- La complexité globale du système augmente et nécessite généralement une conception basée sur le domaine .
- Le système doit gérer les événements en double (idempotents) ou les événements manquants.
- La migration du schéma des événements devient un défi.

- Systèmes transactionnels hautement évolutifs avec bases de données SQL.
- Systèmes transactionnels avec bases de données NoSQL.
- Architecture de microservice hautement évolutive et résiliente.
- Systèmes typiques axés sur les messages ou les événements (systèmes de commerce électronique, de réservation et de réservation).

- Systèmes transactionnels peu évolutifs avec bases de données SQL.
- Dans une architecture microservice simple où les microservices peuvent échanger des données de manière synchrone (par exemple, via API).

Magasin d'événements: EventStoreDB , Apache Kafka , Confluent Cloud , AWS Kinesis , Azure Event Hub , GCP Pub / Sub , Azure Cosmos DB , MongoDB , Cassandra . Amazon DynamoDB ,

Cadres: Lagom , Akka , Spring , akkatecture , Axon , Eventuate

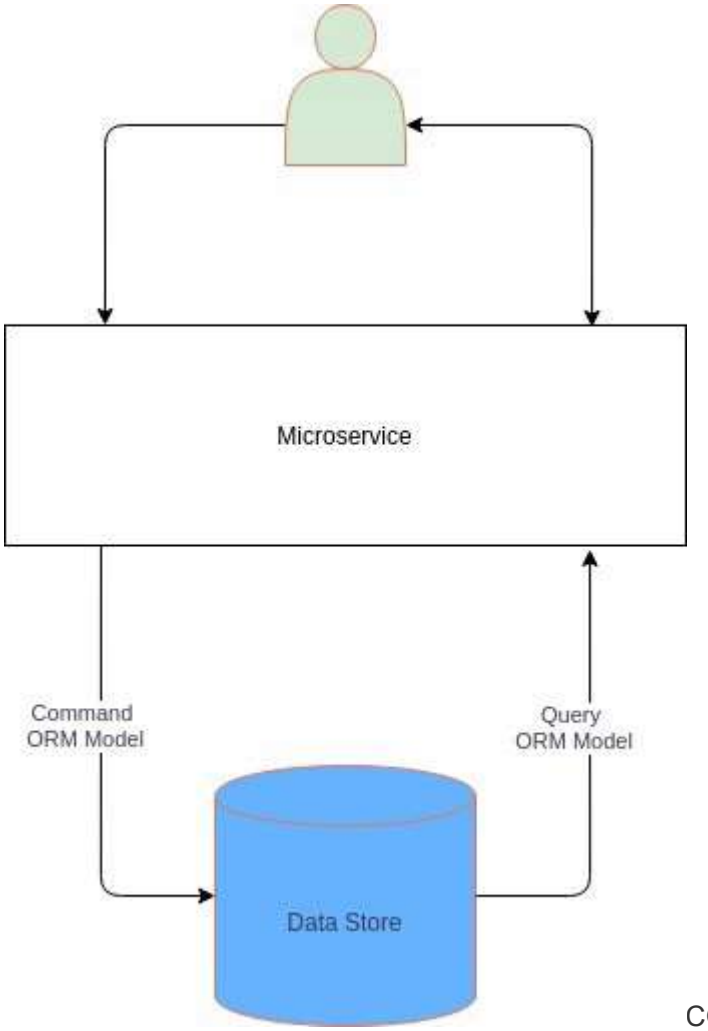
Lectures complémentaires

Ségrégation des responsabilités des requêtes de commande (CQRS)

Si nous utilisons Event Sourcing, alors la lecture des données de l'Event Store devient difficile. Pour récupérer une entité dans le magasin de données, nous devons traiter tous les événements d'entité. De plus, nous avons parfois des exigences de cohérence et de débit différentes pour les opérations de lecture et d'écriture.

Dans de tels cas d'utilisation, nous pouvons utiliser le modèle CQRS. Dans le modèle CQRS, la partie de modification de données du système (commande) est séparée de la partie de lecture de données (requête). Le modèle CQRS a deux formes: simple et avancée, ce qui conduit à une certaine confusion parmi les ingénieurs en logiciel.

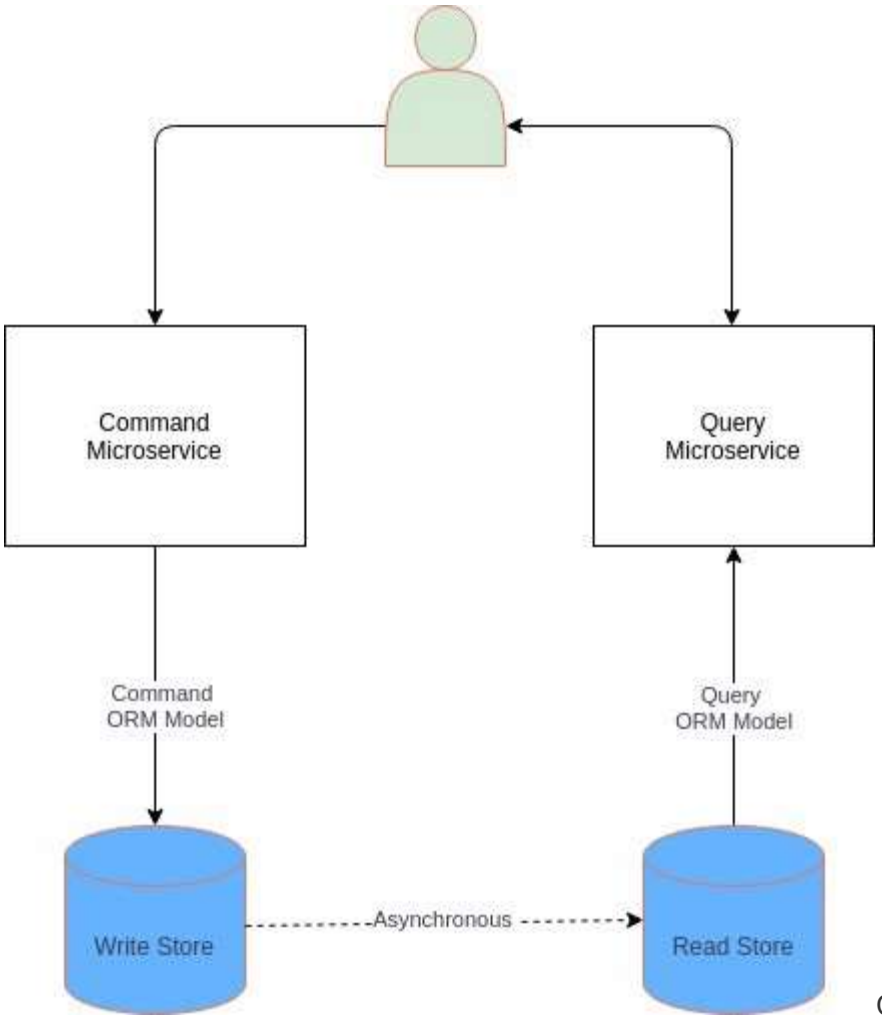
Dans sa forme simple, des entités distinctes ou des modèles ORM sont utilisés pour la lecture et l'écriture, comme indiqué ci-dessous:



CQRS (simple) par Md Kamaruzzaman

Il aide à appliquer le principe de responsabilité unique et la séparation des préoccupations, qui conduisent à une conception plus propre.

Dans sa forme avancée, différents magasins de données sont utilisés pour les opérations de lecture et d'écriture. Le CQRS avancé est utilisé avec Event Sourcing. Selon le cas d'utilisation, différents types de magasin de données d'écriture et de magasin de données de lecture sont utilisés. Le magasin de données d'écriture est le «système d'enregistrements», c'est-à-dire la source d'or du système tout entier.



CQRS (avancé) par Md Kamaruzzaman

Pour les applications à lecture intensive ou l'architecture de microservice, la base de données OLTP (toute base de données SQL ou NoSQL offrant une garantie de transaction ACID) ou la plate-forme de messagerie distribuée est utilisée comme magasin d'écriture. Pour les applications gourmandes en écriture (évolutivité et débit d'écriture élevés), une base de données évolutive en écriture horizontale est utilisée (bases de données globales de cloud public). Les données normalisées sont enregistrées dans le magasin de données d'écriture.

La base de données NoSQL optimisée pour la recherche (par exemple, Apache Solr, Elasticsearch) ou la lecture (magasin de données clé-valeur, magasin de données de document) est utilisée comme magasin de lecture. Dans de nombreux cas, des bases de données SQL évolutives en lecture sont utilisées lorsque la requête SQL est souhaitée. Les données dénormalisées et optimisées sont enregistrées dans le Read Store.

Les données sont copiées du magasin d'écriture vers le magasin de lecture de manière asynchrone. Par conséquent, le magasin de lecture est en retard sur le magasin d'écriture et est **éventuellement cohérent** .

Avantages

- Lecture plus rapide des données dans les microservices événementiels.
- Haute disponibilité des données.
- Les systèmes de lecture et d'écriture peuvent évoluer indépendamment.

- La lecture du magasin de données est faiblement cohérente (cohérence éventuelle)

- La complexité globale du système augmente. La culture de la cargaison CQRS peut compromettre considérablement l'ensemble du projet.

- Dans une architecture de microservices hautement évolutive où l'approvisionnement d'événements est utilisé.
- Dans un modèle de domaine complexe où la lecture des données nécessite une requête dans plusieurs magasins de données.
- Dans les systèmes où les opérations de lecture et d'écriture ont une charge différente.

- Dans l'architecture de microservice, où le volume d'événements est insignifiant, prendre l'instantané du magasin d'événements pour calculer l'état de l'entité est un meilleur choix.
- Dans les systèmes où les opérations de lecture et d'écriture ont une charge similaire.

Magasin d'écriture: EventStoreDB , Apache Kafka , Confluent Cloud , AWS Kinesis , Azure Event Hub , GCP Pub / Sub , Azure Cosmos DB , MongoDB , Cassandra . Amazon DynamoDB

Lire la boutique: Elastic Search , Solr , Cloud Spanner , Amazon Aurora , Azure Cosmos DB , Neo4j

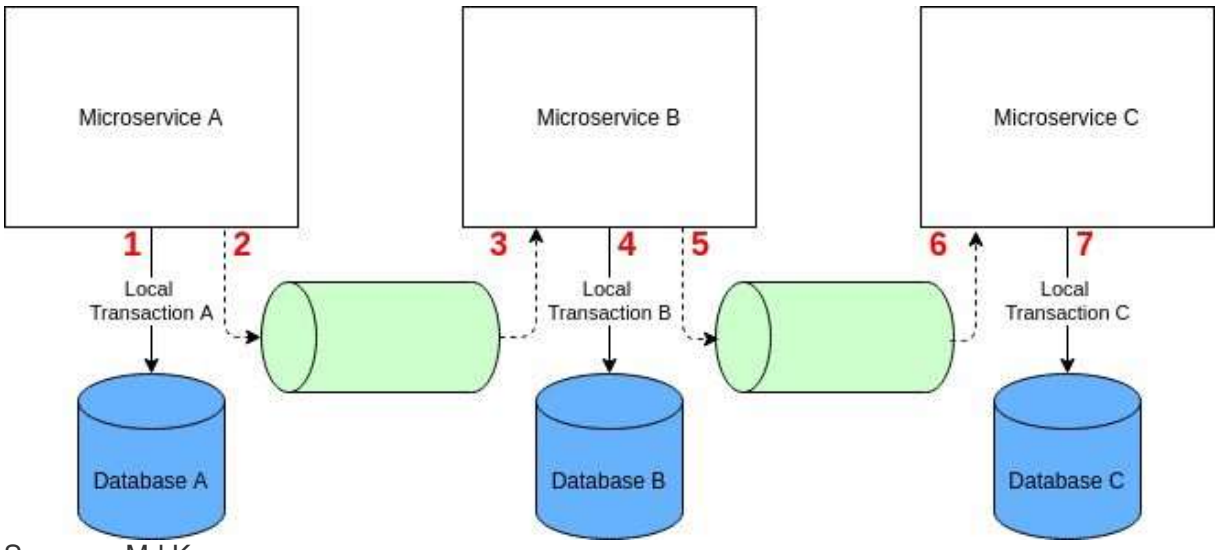
Cadres: Lagom , Akka , Spring , akkatecture , Axon , Eventuate

Lectures complémentaires

Saga

Si vous utilisez l'architecture microservice avec **base** de **données par microservice** , la gestion de la cohérence via des transactions distribuées est un défi. Vous ne pouvez pas utiliser le protocole de validation en deux phases traditionnel car il ne s'adapte pas (bases de données SQL) ou n'est pas pris en charge (de nombreuses bases de données NoSQL).

Vous pouvez utiliser le modèle Saga pour les transactions distribuées dans Microservice Architecture. Saga est un ancien modèle développé en 1987 comme alternative conceptuelle pour les transactions de base de données de longue durée dans les bases de données SQL. Mais une variante moderne de ce modèle fonctionne également à merveille pour la transaction distribuée. Le modèle Saga est une séquence de transactions locale dans laquelle chaque transaction met à jour les données dans le magasin de données au sein d'un seul microservice et publie un événement ou un message. La première transaction d'une saga est initiée par une requête externe (événement ou action). Une fois la transaction locale terminée (les données sont stockées dans le magasin de données et le message ou l'événement est publié), le message / événement publié déclenche la prochaine transaction locale dans la saga.



Saga par Md Kamaruzzaman

Si la transaction locale échoue, Saga exécute une série de transactions de compensation qui annulent les modifications des transactions locales précédentes.

Il existe principalement deux variantes de coordination des transactions Saga:

- *Chorégraphie* : coordinations décentralisées où chaque microservice produit et écoute les événements / messages des autres microservices et décide si une action doit être entreprise ou non.
- *Orchestration* : coordinations centralisées où un orchestrateur indique aux microservices participants quelle transaction locale doit être exécutée.

- Assurer la cohérence via des transactions dans une architecture de microservice hautement évolutive ou faiblement couplée, axée sur les événements.
- Assurer la cohérence via des transactions dans l'architecture Microservice où des bases de données NoSQL sans prise en charge 2PC sont utilisées.

- Besoin de gérer les échecs transitoires et devrait fournir l'idempotence.
- Difficile à déboguer, et la complexité augmente à mesure que le nombre de microservices augmente.

- Dans une architecture de microservices hautement évolutive et faiblement couplée, dans laquelle le sourcing d'événements est utilisé.
- Dans les systèmes où des bases de données NoSQL distribuées sont utilisées.

- Systèmes transactionnels peu évolutifs avec bases de données SQL.
- Dans les systèmes où il existe une dépendance cyclique entre les services.

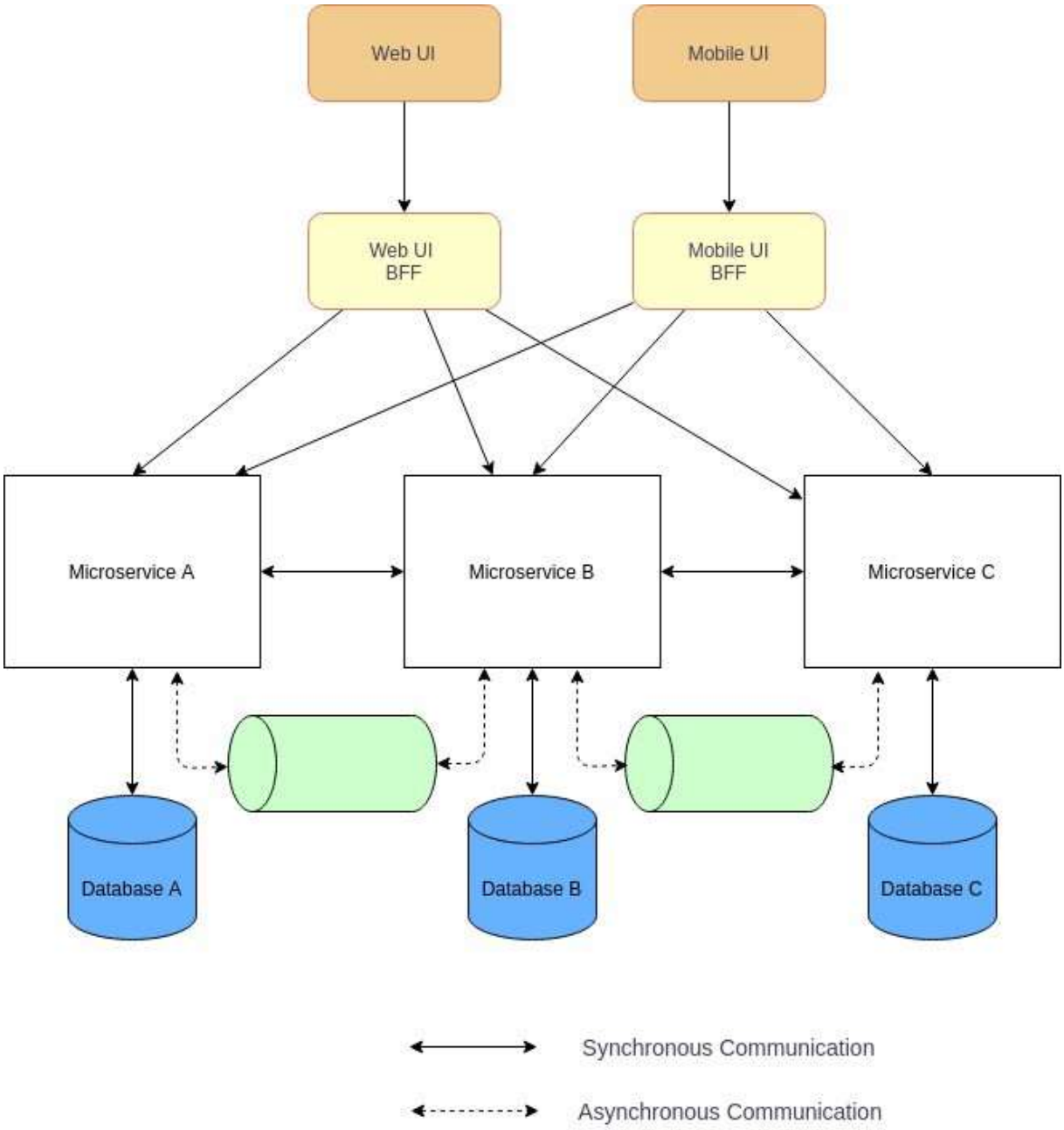
Axon , Eventuate , Narayana

Lectures complémentaires

Backends pour frontends (BFF)

Dans les développements d'applications d'entreprise modernes et en particulier dans l'architecture de microservices, les applications frontales et backend sont des services découplés et séparés. Ils sont connectés via API ou GraphQL. Si l'application dispose également d'un client d'application mobile, l'utilisation du même microservice principal pour le Web et le client mobile devient problématique. Les exigences de l'API du client mobile sont généralement différentes de celles du client Web car elles ont une taille d'écran, un affichage, des performances, une source d'énergie et une bande passante réseau différents.

Le modèle Backends for Frontends peut être utilisé dans des scénarios où chaque interface utilisateur obtient un backend distinct personnalisé pour l'interface utilisateur spécifique. Il offre également d'autres avantages, comme agir en tant que façade pour les microservices en aval, réduisant ainsi la communication bavarde entre l'interface utilisateur et les microservices en aval. En outre, dans un scénario hautement sécurisé où les microservices en aval sont déployés dans un réseau DMZ, les BFF sont utilisés pour fournir une sécurité plus élevée.



Backends pour Frontends par Md Kamaruzzaman

Avantages

- Séparation des préoccupations entre les BFF. Nous pouvons les optimiser pour une interface utilisateur spécifique.
- Fournissez une sécurité plus élevée.
- Fournissez une communication moins bavarde entre l'interface utilisateur et les microservices en aval.

- Duplication de code entre BFF.
- La prolifération des BFF au cas où de nombreuses autres interfaces utilisateur sont utilisées (par exemple, Smart TV, Web, mobile, bureau).
- Besoin d'une conception et d'une mise en œuvre soignées, car les BFF ne doivent contenir aucune logique métier et doivent uniquement contenir une logique et un comportement spécifiques au client.

- Si l'application dispose de plusieurs interfaces utilisateur avec des exigences d'API différentes.
- Si une couche supplémentaire est nécessaire entre l'interface utilisateur et les microservices en aval pour des raisons de sécurité.
- Si des micro-frontends sont utilisés dans le développement de l'interface utilisateur.

- Si l'application dispose de plusieurs interfaces utilisateur, mais qu'elles utilisent la même API.
- Si Core Microservices ne sont pas déployés dans DMZ.

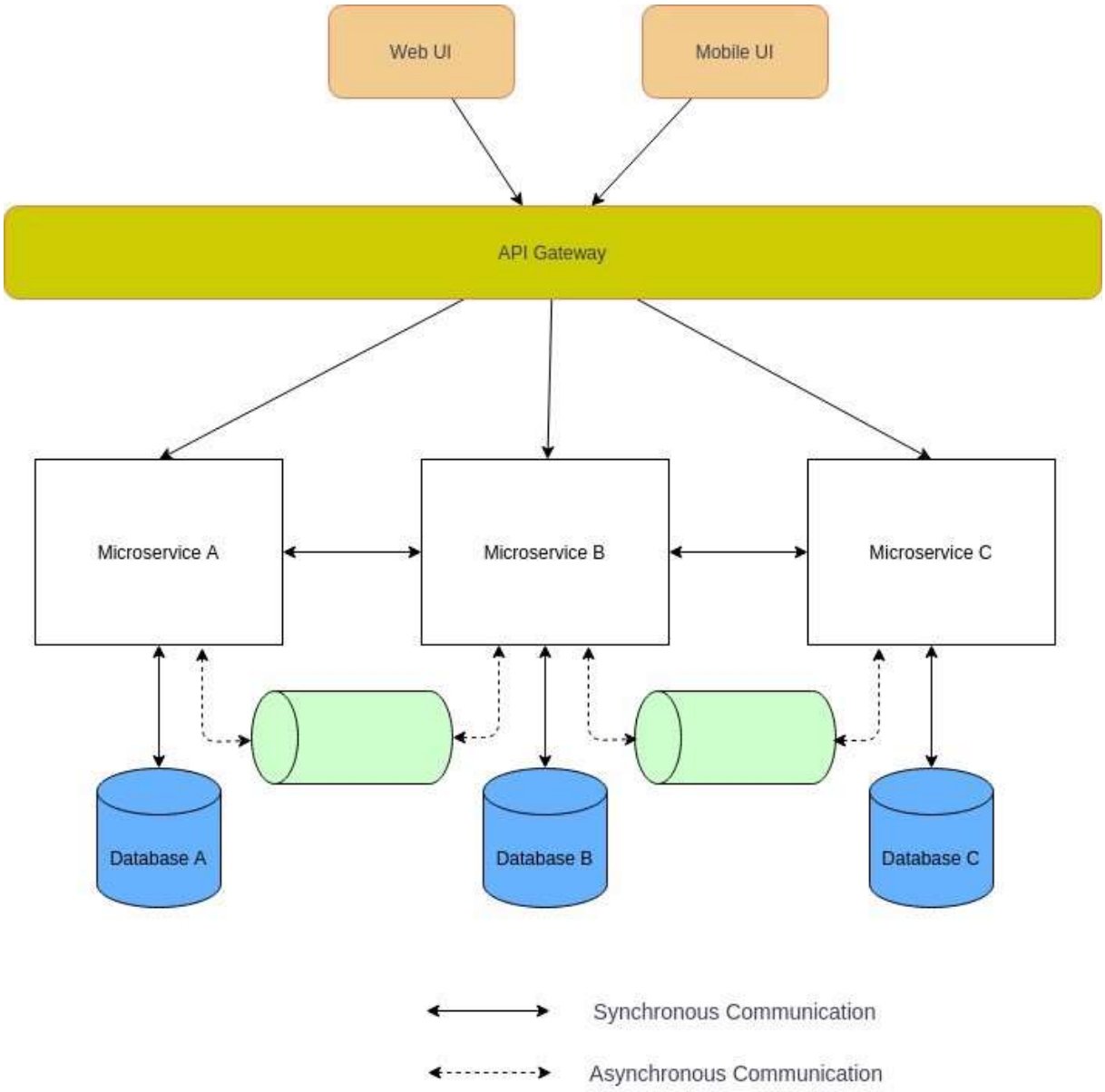
Tous les frameworks Backend (Node.js, Spring, Django, Laravel, Flask, Play,… ..) le supportent.

Lectures complémentaires

Passerelle API

Dans l'architecture de microservices, l'interface utilisateur se connecte généralement à plusieurs microservices. Si les microservices sont finement granulés (FaaS), le client peut avoir besoin de se connecter à de nombreux microservices, ce qui devient bavard et stimulant. De plus, les services, y compris leurs API, peuvent évoluer. Les grandes entreprises aimeront avoir d'autres préoccupations transversales (terminaison SSL, authentification, autorisation, limitation, journalisation, etc.).

Une façon possible de résoudre ces problèmes consiste à utiliser API Gateway. API Gateway se situe entre l'APP client et les microservices backend et fait office de façade. Il peut fonctionner comme un proxy inverse, acheminant la demande du client vers le microservice backend approprié. Il peut également prendre en charge le déploiement de la demande client vers plusieurs microservices, puis renvoyer les réponses agrégées au client. Il soutient en outre des préoccupations transversales essentielles.



Passerelle API par Md Kamaruzzaman

Avantages

- Offrez un couplage lâche entre les microservices frontend et backend.
- Réduisez le nombre d'appels aller-retour entre le client et les microservices.
- Haute sécurité via la terminaison, l'authentification et l'autorisation SSL.
- Problèmes transversaux gérés de manière centralisée, par exemple, journalisation et surveillance, limitation, équilibrage de charge.

- Peut conduire à un point de défaillance unique dans l'architecture de microservice.
- Latence accrue en raison de l'appel réseau supplémentaire.
- S'il n'est pas mis à l'échelle, ils peuvent facilement devenir le goulot d'étranglement pour toute l'entreprise.
- Frais de maintenance et de développement supplémentaires.

- Dans l'architecture microservice complexe, c'est presque obligatoire.
- Dans les grandes entreprises, API Gateway est obligatoire pour centraliser les préoccupations de sécurité et transversales.

- Dans les projets privés ou les petites entreprises où la sécurité et la gestion centrale ne sont pas la priorité absolue.
- Si le nombre de microservices est assez petit.

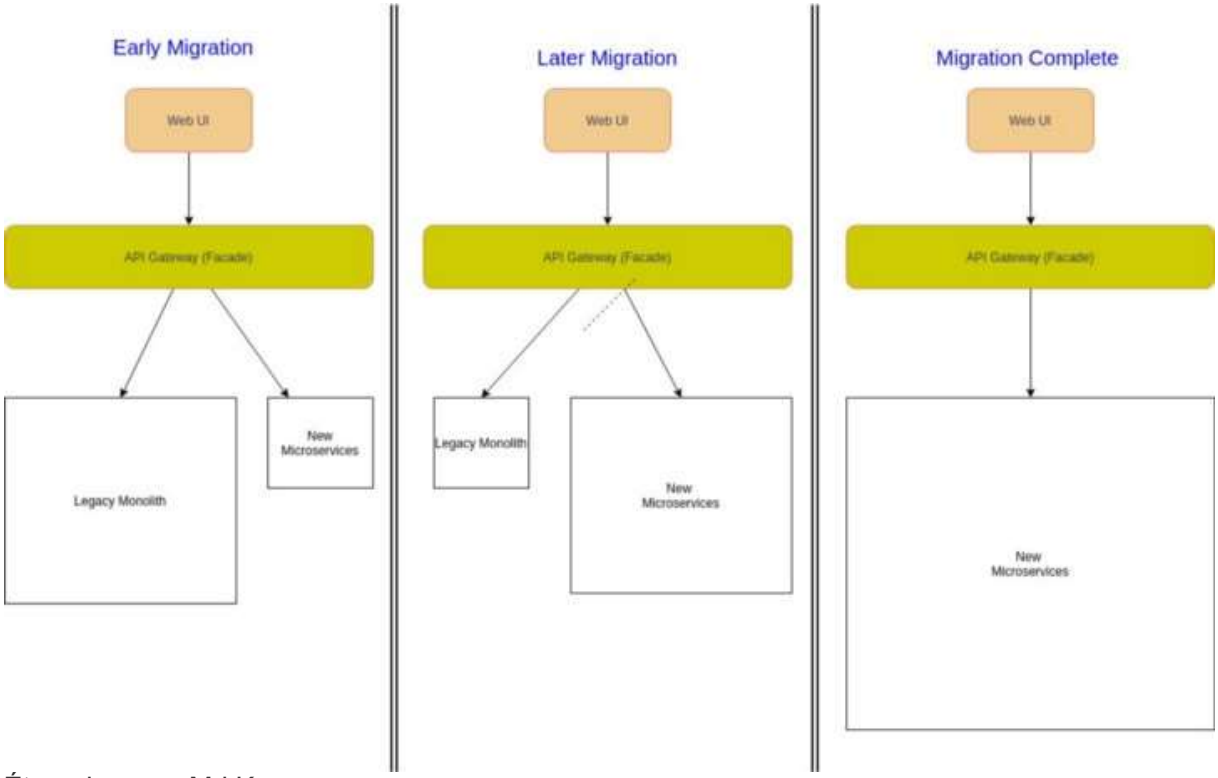
Amazon API Gateway , Azure API Management , Apigee , Kong , WSO2 API Manager

Lectures complémentaires

Étrangleur

Si nous voulons utiliser l'architecture de microservice dans un projet de friche industrielle, nous devons migrer les applications monolithiques héritées ou existantes vers des microservices. Le déplacement d'une application monolithique existante, volumineuse et en production vers des microservices est assez difficile car cela peut perturber la disponibilité de l'application.

Une solution consiste à utiliser le modèle Strangler. Un modèle plus étrange signifie la migration incrémentielle d'une application monolithique vers une architecture de microservices en remplaçant progressivement des fonctionnalités spécifiques par de nouveaux microservices. De plus, de nouvelles fonctionnalités ne sont ajoutées que dans Microservices, en contournant l'ancienne application Monolithic. Une façade (API Gateway) est ensuite configurée pour acheminer les demandes entre l'ancien Monolith et les Microservices. Une fois la fonctionnalité migrée du Monolith vers les microservices, la façade intercepte alors la demande du client et achemine vers les nouveaux microservices. Une fois que toutes les fonctionnalités du monolithe hérité ont été migrées, l'application monolithique héritée est «étranglée», c'est-à-dire mise hors service.



Étrangleur par Md Kamaruzzaman

Avantages

- Migration sécurisée de l'application monolithique vers les microservices.
- La migration et le développement de nouvelles fonctionnalités peuvent se faire en parallèle.

- Le processus de migration peut avoir son propre rythme.

- Le partage du magasin de données entre le Monolith existant et les nouveaux microservices devient difficile.
- L'ajout d'une façade (API Gateway) augmentera la latence du système.
- Les tests de bout en bout deviennent difficiles.

- Migration incrémentielle d'une grande application Backend Monolithic vers Microservices.

- Si le backend monolith est petit, le remplacement en gros est une meilleure option.
- Si la demande du client à l'ancienne application Monolithic ne peut pas être interceptée.

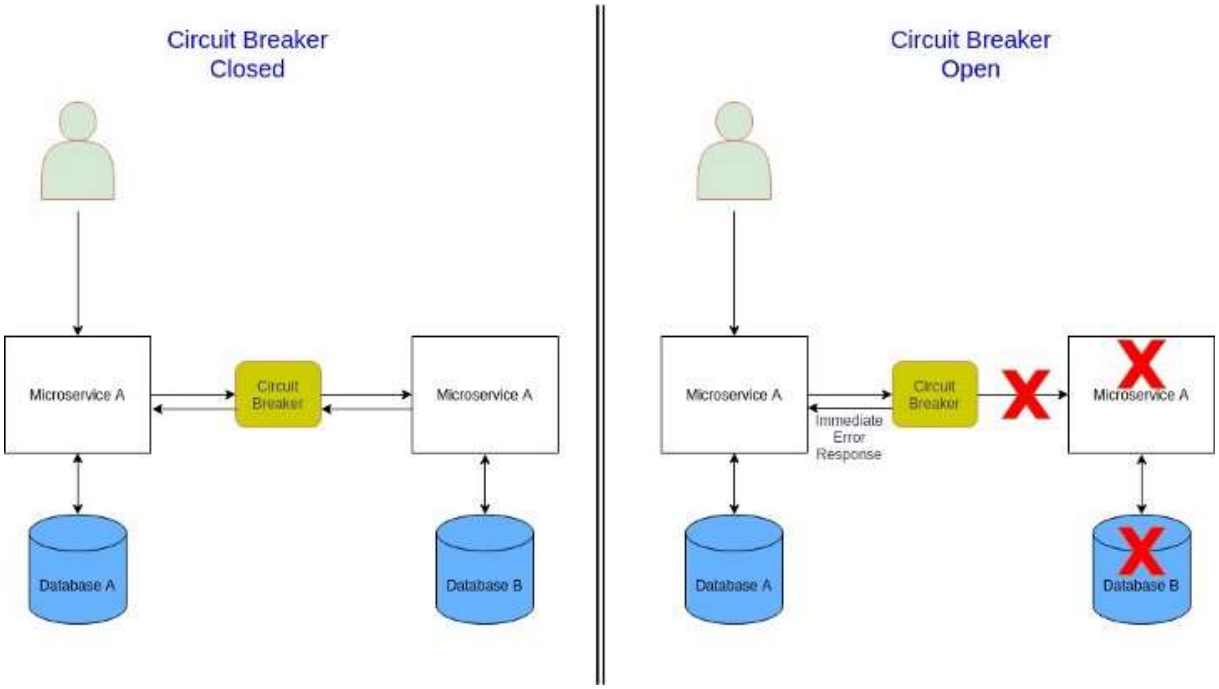
Framework d'applications backend avec API Gateway.

Lectures complémentaires

Disjoncteur

Dans l'architecture de microservices, où les microservices communiquent de manière synchrone, un microservice appelle généralement d'autres services pour répondre aux besoins de l'entreprise. L'appel à un autre service peut échouer en raison de pannes transitoires (connexion réseau lente, délais d'expiration ou indisponibilité temporelle). Dans de tels cas, une nouvelle tentative d'appels peut résoudre le problème. Cependant, en cas de problème grave (échec complet du microservice), le microservice est indisponible pendant plus longtemps. Une nouvelle tentative est inutile et gaspille des ressources précieuses (le thread est bloqué, gaspillage de cycles CPU) dans de tels scénarios. En outre, l'échec d'un service peut entraîner des échecs en cascade dans toute l'application. Dans de tels scénarios, l'échec immédiat est une meilleure approche.

Le modèle Circuit Breaker peut venir à la rescousse pour de tels cas d'utilisation. Un microservice doit demander un autre microservice via un proxy qui fonctionne de la même manière qu'un **disjoncteur électrique.** Le proxy doit compter le nombre d'échecs récents qui se sont produits et l'utiliser pour décider d'autoriser la poursuite de l'opération ou simplement de renvoyer une exception immédiatement.



Disjoncteur par Md Kamaruzzaman

Le disjoncteur peut avoir les trois états suivants:

- *Fermé:* le disjoncteur achemine les demandes vers le microservice et compte le nombre de pannes dans une période de temps donnée. Si le nombre de pannes dans une certaine période de temps dépasse un seuil, il se déclenche et passe à l'état ouvert.
- *Ouvrir* : la demande du microservice échoue immédiatement et une exception est renvoyée. Après un délai d'attente, le disjoncteur passe à un état semi-ouvert.
- *Semi-ouvert* : Seul un nombre limité de demandes du microservice sont autorisés à passer et à appeler l'opération. Si ces demandes aboutissent, le disjoncteur passe à l'état fermé. Si une demande échoue, le disjoncteur passe à l'état ouvert.

- Améliorez la tolérance aux pannes et la résilience de l'architecture des microservices.
- Arrête la cascade de l'échec vers d'autres microservices.

- Besoin d'une gestion sophistiquée des exceptions.
- Journalisation et surveillance.
- Devrait prendre en charge la réinitialisation manuelle.

- Dans une architecture de microservices étroitement couplée où les microservices communiquent de manière synchrone.
- Si un microservice a une dépendance sur plusieurs autres microservices.

- Architecture de microservice faiblement couplée et orientée événement.
- Si un microservice n'a aucune dépendance vis-à-vis d'autres microservices.

API Gateway, Service Mesh, diverses bibliothèques
de disjoncteurs (Hystrix , Reselience4J , Polly .

Lectures complémentaires

Configuration externalisée

Chaque application d'entreprise a de nombreux paramètres de configuration pour diverses infrastructures (par exemple, base de données, réseau, adresses de service connectées, informations d'identification, chemin du certificat). En outre, dans un environnement d'entreprise, l'application est généralement déployée dans **différents** environnements d' **exécution (Local, Dev, Prod)** . Une façon d'y parvenir consiste à utiliser la configuration interne, qui est une mauvaise pratique fatale. Cela peut entraîner un risque de sécurité grave car les informations d'identification de production peuvent facilement être compromises. En outre, toute modification du paramètre de configuration doit reconstruire l'application. Ceci est encore plus critique dans l'architecture de microservice car nous avons potentiellement des centaines de services.

La meilleure approche consiste à externaliser toutes les configurations. Par conséquent, le processus de génération est séparé de l'environnement d'exécution. En outre, cela minimise le risque de sécurité car le fichier de configuration Production n'est utilisé que pendant l'exécution ou via des variables d'environnement.

Avantages

- Les configurations de production ne font pas partie de la base de code et minimisent ainsi les vulnérabilités de sécurité.
- Les paramètres de configuration peuvent être modifiés sans nouvelle version.

- Nous devons choisir un cadre qui prend en charge la configuration externalisée.

- Toute application de production sérieuse doit utiliser une configuration externalisée.

- En preuve de développement de concept.

Presque tous les frameworks modernes de niveau entreprise prennent en charge la configuration externalisée.

Lectures complémentaires

Test de contrat axé sur le consommateur

Dans l'architecture de microservices, il existe de nombreux microservices souvent développés par des équipes distinctes. Ces microservices fonctionnent ensemble pour répondre à une exigence commerciale (par exemple, la demande du client) et communiquent entre eux de manière synchrone ou asynchrone. Les tests d'intégration d'un microservice **grand public** sont difficiles. Habituellement, **TestDouble** est utilisé dans de tels scénarios pour une exécution de test plus rapide et moins chère. Mais TestDouble ne représente souvent pas le véritable microservice du **fournisseur** . De plus, si le microservice du fournisseur modifie son API ou son message, TestDouble ne parvient pas à l' accepter . L'autre option consiste à effectuer des tests de bout en bout. Bien que les tests de bout en bout soient obligatoires avant la production, ils sont fragiles, lents, coûteux et ne remplacent pas les tests d'intégration (Test Pyramid).

Les tests contractuels axés sur les consommateurs peuvent nous aider à cet égard. Ici, l'équipe propriétaire de Consumer Microservice écrit une suite de tests contenant sa demande et sa réponse attendue (pour la communication synchrone) ou les messages attendus (pour la communication asynchrone) pour un microservice fournisseur particulier. Ces suites de tests sont appelées **contrats** explicites . Pour un microservice de fournisseur, toutes les suites de tests de contrat de ses consommateurs sont ajoutées dans son test automatisé. Lorsque le test automatisé pour un microservice fournisseur particulier est effectué, il exécute ses propres tests et les contrats et vérifie le contrat. De cette manière, le test du contrat peut aider à maintenir l'intégrité de la communication de microservice de manière automatisée.

Avantages

- Si le fournisseur modifie l'API ou le message de manière inattendue, il est détecté de manière autonome en peu de temps.
- Moins de surprise et plus de robustesse, surtout une application d'entreprise contenant beaucoup de microservices.
- Amélioration de l'autonomie de l'équipe.

- Besoin de travail supplémentaire pour développer et intégrer des tests de contrat dans Provider Microservice car ils peuvent utiliser des outils de test complètement différents.
- Si le test du contrat ne correspond pas à la consommation réelle du service, cela peut entraîner un échec de la production.

- Dans les applications d'entreprise à grande échelle, où généralement, différentes équipes développent différents services.

- Applications relativement plus simples et plus petites où une équipe développe tous les microservices.
- Si les microservices du fournisseur sont relativement stables et ne sont pas en cours de développement actif.

Pacte , facteur , contrat Spring Cloud

Lectures complémentaires

Conclusion

Dans le développement de logiciels d'entreprise moderne à grande échelle, Microservice Architecture peut aider le développement à évoluer avec de nombreux avantages à long terme. Mais Microservice Architecture n'est pas une Silver Bullet qui peut être utilisée dans tous les cas d'utilisation. S'il est utilisé dans le mauvais type d'application, Microservice Architecture peut donner plus de peine en tant que gains. L'équipe de développement qui souhaite adopter l'architecture de microservice doit suivre un ensemble de bonnes pratiques et utiliser un ensemble de modèles de conception réutilisables et endurcis.

Le modèle de conception le plus essentiel dans l'architecture de microservice est la **base de données par microservice** . La mise en œuvre de ce modèle de conception est difficile et nécessite plusieurs autres modèles de conception étroitement liés (**Event Sourcing, CQRS, Saga**). Dans les applications d'entreprise typiques avec plusieurs clients (Web, mobile, bureau, appareils intelligents), les communications entre le client et les microservices peuvent être bavardes et peuvent nécessiter un contrôle central avec une sécurité accrue. Les modèles de conception **Backends for Frontends** et **API Gateway** sont très utiles dans de tels scénarios. En outre, le modèle de **disjoncteur** peut grandement aider à gérer les scénarios d'erreur dans de telles applications. La migration d'une application monolithique héritée vers des microservices est assez difficile, et le modèle **Strangler** peut faciliter la migration. Le **test de contrat** axé sur le **consommateur** est un modèle instrumental pour le test d'intégration de microservices. Dans le même temps, **externaliser la configuration** est un modèle obligatoire dans tout développement d'application moderne.