MicroServices

Copyright 👍

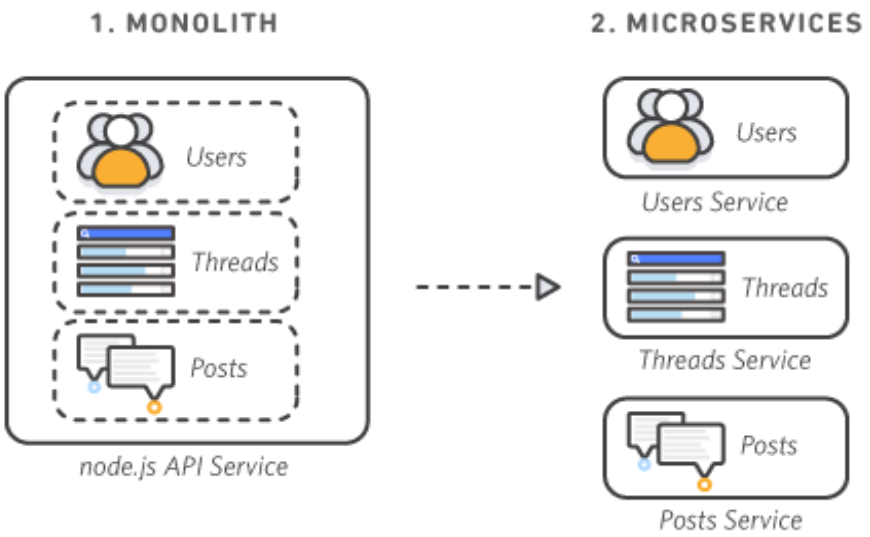
KAKE Abdoulaye
Architecte Logiciel
Formateur Conception, développement d'applications et DevOps
Technologies : C# - .Net , SQL Server, React JS, Azure, Azure DevOps, Firebase

Architecture MicroServices

Introduction aux MicroServices

Les **microServices** sont un modèle architectural de **conception logicielle** qui a gagné en popularité ces dernières années. Contrairement aux approches **monolithiques**, où une application est développée comme une entité unique et massive, l'architecture microservices divise une application en un ensemble de services plus petits et autonomes. Chaque **microService** est responsable d'une fonction spécifique et communique avec d'autres **microservices** via des **API** bien définies. Cette approche favorise la modularité, la flexibilité et la scalabilité, permettant aux équipes de développement de travailler indépendamment sur des parties distinctes de l'application.

L'architecture monolithique vs. microservices



Lors de la conception et du développement d'applications logicielles, deux approches architecturales principales sont souvent considérées : l'**architecture monolithique** et l'**architecture microservices**. Chacune de ces approches a ses propres avantages, inconvénients et cas d'utilisation appropriés.

- **Architecture Monolithique** :

  L'**architecture monolithique** est un modèle traditionnel où l'ensemble de l'application est conçu et développé comme une seule entité, généralement dans un seul code source et exécuté dans un seul processus. Toutes les fonctionnalités, la logique métier et l'interface utilisateur sont intégrées au sein de cette seule application.

- **Avantages de l'Architecture Monolithique** :
    1. **Simplicité de développement** : Le développement, le déploiement et la maintenance sont plus simples car toute l'application est contenue dans un seul projet : toutes les parties de l'application sont regroupées dans un seul code source.
    2. **Moins de Complexité** : La gestion des données et de la communication entre les différentes parties de l'application est plus simple car elles sont toutes au sein d'un même environnement et plus rapide car généralement les appels de fonction se font au sein de la même mémoire.
    3. **Facilité de débogage** : Le débogage est plus facile car toutes les fonctionnalités sont intégrées au même ensemble de code.

- **Inconvénients de l'Architecture Monolithique** :

1. **Évolutivité limitée** : L'évolutivité peut être limitée car l'ensemble de l'application doit être mis à l'échelle, même si certaines parties ne nécessitent pas de ressources supplémentaires ou n'en ont pas besoin.
2. **Dépendances fortes ou Couplage Élevé** : Les changements dans une partie de l'application peuvent avoir un impact sur d'autres parties, rendant les modifications plus risquées, car une modification peut avoir des répercussions inattendues ailleurs.
3. **Limitations Technologiques** : L'utilisation de différentes technologies pour différentes parties de l'application peut être limitée en raison de la nature monolithique.
4. **Longs Cycles de Développement** : Les mises à jour et les déploiements peuvent nécessiter des cycles de développement plus longs, ce qui peut ralentir l'innovation.

● Architecture Microservices :

L'**architecture microservices** divise l'application en un ensemble de services autonomes et indépendants, chacun responsable d'une fonction spécifique. Chaque microservice peut être développé, déployé et mis à l'échelle indépendamment.

- **Avantages de l'Architecture Microservices** :

1. **Évolutivité ciblée** : Les microservices peuvent être mis à l'échelle individuellement en fonction des besoins, ce qui améliore l'évolutivité et permet d'allouer efficacement les ressources là où elles sont nécessaires.
2. **Technologies variées** : Différents microservices peuvent utiliser différentes technologies, ce qui permet aux équipes de choisir les outils les mieux adaptés à chaque tâche et d'exploiter les dernières innovations.
3. **Agilité de Développement** : Les équipes peuvent travailler indépendamment sur des microservices, accélérant ainsi le développement et la mise sur le marché de nouvelles fonctionnalités.

4. **Déploiement rapide** : Les microservices peuvent être déployés séparément, ce qui permet des mises à jour plus rapides et des cycles de développement plus courts.
5. **Meilleure Isolation** : Les microservices sont isolés les uns des autres, ce qui limite l'impact des erreurs et des pannes à un seul microservice.
6. **Réutilisation** : Les microservices peuvent être réutilisés dans d'autres projets, ce qui favorise la modularité et la réduction du travail répétitif.

- **Inconvénients de l'Architecture Microservices** :

1. **Complexité de Gestion** : La gestion de plusieurs microservices peut être complexe et nécessite des outils de surveillance, de gestion et de déploiement sophistiqués.
2. **Coordination** : La coordination entre les microservices peut être nécessaire pour garantir une exécution cohérente des processus métier, ce qui peut être complexe à gérer.
3. **Communication** : La communication entre les microservices peut entraîner des problèmes de latence, de cohérence des données et nécessite une gestion minutieuse.
4. **Tests Plus Complexes** : Les tests d'intégration entre les microservices sont plus complexes que dans une application monolithique, nécessitant des stratégies et des outils spécifiques.
5. **Sécurité** : La gestion de la sécurité doit être mise en place pour chaque microservice et pour les communications entre eux, ce qui peut être un défi.
6. **Taille des équipes** : Les projets basés sur des microservices peuvent nécessiter des équipes plus importantes pour gérer et développer chaque microservice.

Principes clés des microservices : découplage, indépendance, remplacement

Les microservices sont basés sur plusieurs principes fondamentaux qui contribuent à leur agilité, à leur évolutivité et à leur facilité de maintenance. Trois de ces principes clés sont le **découplage**, **l'indépendance** et la capacité de **remplacement**.

1. Découplage :

Le principe du **découplage** dans les microservices consiste à minimiser les dépendances entre les différentes parties d'une application. Chaque microservice est conçu pour fonctionner de manière autonome et indépendante des autres. Cela permet de modifier, mettre à jour ou remplacer un microservice sans affecter l'ensemble du système.

- **Exemple** : Imaginons une application e-commerce avec deux microservices distincts : **"CatalogService"** pour gérer le catalogue de produits et **"OrderService"** pour

gérer les commandes. Si le **"CatalogService"** est mis à jour pour ajouter de nouvelles fonctionnalités, le **"OrderService"** ne sera pas affecté, car ils sont découplés et communiquent via des API bien définies.

2. Indépendance :

L'indépendance est le principe selon lequel chaque microservice peut être développé, déployé et géré de manière autonome. Chaque service est responsable d'une fonction spécifique et peut être géré par une équipe distincte. Cela favorise l'agilité et permet aux équipes de se concentrer sur leur propre domaine sans affecter les autres parties de l'application.

- **Exemple** : Reprenons l'exemple de l'application e-commerce. L'équipe en charge du **"CatalogService"** peut travailler de manière indépendante pour ajouter de nouveaux produits, modifier les descriptions, etc. L'équipe en charge du **"OrderService"** peut simultanément développer de nouvelles fonctionnalités de traitement de commandes sans interférer avec le **"CatalogService"**.

3. Capacité de Remplacement :

Les microservices doivent être conçus de manière à pouvoir être remplacés par de nouveaux services sans perturber l'ensemble de l'application. Cela permet d'adopter de nouvelles technologies, de mettre en œuvre des améliorations ou de répondre aux évolutions du marché sans subir de lourdes modifications.

- **Exemple** : Supposons que dans notre application e-commerce, nous voulions remplacer le **"CatalogService"** par un nouveau service qui intègre une intelligence artificielle pour recommander des produits aux utilisateurs. Grâce à la capacité de remplacement, nous pouvons développer ce nouveau service sans affecter les autres parties de l'application, et le déployer progressivement tout en maintenant la stabilité de l'ensemble.

Ces principes clés du découplage, de l'indépendance et de la capacité de remplacement sont essentiels pour la conception et la mise en œuvre réussies des microservices. Ils favorisent l'agilité, la modularité et la maintenance facilitée des applications complexes.

Conception de Microservices

Décomposition de l'application en services

La décomposition d'une application en services est une étape clé dans la conception d'une architecture microservices. Voici comment procéder pour décomposer votre application en services cohérents et indépendants :

1. Analyse des Fonctionnalités : Identifiez toutes les fonctionnalités de votre application. Ces fonctionnalités peuvent inclure des actions que les utilisateurs peuvent effectuer, telles que la gestion des utilisateurs, la recherche de produits, le traitement des paiements, etc.

2. Découpage en Domaines Métier : Regroupez les fonctionnalités en domaines métier. Chaque domaine métier deviendra la base d'un microservice. Par exemple, si vous avez une application de commerce électronique, les domaines métier pourraient être la gestion des produits, la gestion des commandes, la gestion des utilisateurs, etc.

3. Identification des Responsabilités : Pour chaque domaine métier, identifiez les responsabilités spécifiques qu'il doit gérer. Assurez-vous que chaque microservice ait une responsabilité claire et distincte, et évitez de mélanger les préoccupations.

4. Définition des Interfaces : Pour chaque microservice, définissez les interfaces (API) qu'il exposera pour communiquer avec d'autres microservices et avec les clients externes. Déterminez les types de requêtes et de réponses, les formats de données, etc.

5. Conception des Bases de Données : Chaque microservice doit avoir sa propre base de données (il est possible d'avoir une base de données communes). Concevez les schémas de données spécifiques à chaque microservice pour minimiser les dépendances entre eux. Utilisez des mécanismes comme les vues, les projections ou les événements pour partager des données si nécessaire.

6. Communication Entre Services : Déterminez comment les microservices vont communiquer entre eux. Utiliserez-vous des appels d'API synchrones, des messages asynchrones, ou une combinaison des deux ? Choisissez les mécanismes qui conviennent le mieux à chaque interaction.

7. Gestion des États : Réfléchissez à la gestion des états et à la cohérence des données. Les modèles Event Sourcing ou CQRS peuvent être utilisés pour maintenir la cohérence entre les microservices.

8. Planification de la Sécurité : Prévoyez les mécanismes d'authentification, d'autorisation et de sécurité pour chaque microservice et pour les communications entre eux.

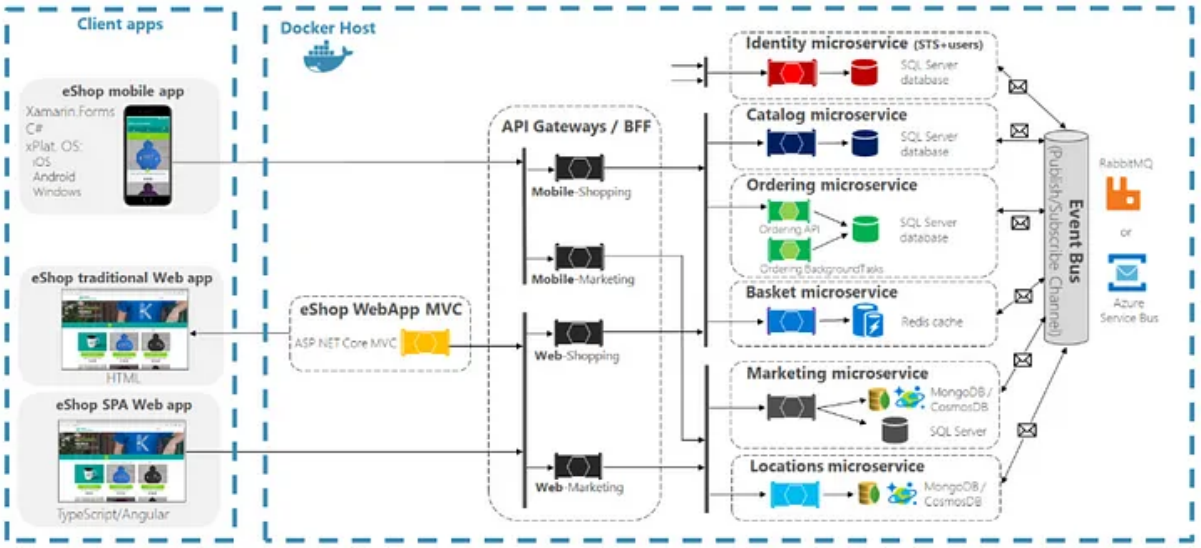
9. Tests et Développement : Développez chaque microservice individuellement en suivant les meilleures pratiques de développement. Écrivez des tests unitaires, des tests d'intégration et des tests de bout en bout pour vous assurer que chaque microservice fonctionne correctement.

10. Déploiement et Gestion : Utilisez des outils de conteneurisation comme Docker et des orchestrateurs comme Kubernetes pour déployer, gérer et surveiller vos microservices.

En décomposant votre application en services basés sur des domaines métier clairement définis, vous obtiendrez une architecture microservices qui favorise l'indépendance, la modularité et l'évolutivité. Chaque microservice peut être développé, déployé et mis à jour de manière indépendante, ce qui facilite la maintenance et l'ajout de nouvelles fonctionnalités au fil du temps.

Exemple : Application de Commerce Électronique

L'application de commerce électronique permet aux utilisateurs de parcourir des produits, de les ajouter au panier, de passer des commandes et de gérer leur compte.



Décomposition en Microservices :

1. Service de Gestion des Produits : Gère la création, la mise à jour et la suppression des produits.

- **Nom** : `ProductService`
- **Fonctionnalités** : Ajouter un produit, mettre à jour les détails d'un produit, supprimer un produit.

2. Service de Gestion des Commandes : Gère la création, la modification et la gestion des commandes.

- **Nom** : `OrderService`
- **Fonctionnalités** : Passer une commande, annuler une commande, obtenir le statut d'une commande.

3. Service de Gestion des Utilisateurs : Gère les informations et les actions liées aux utilisateurs.

- **Nom** : `UserService`
- **Fonctionnalités** : Créer un compte utilisateur, se connecter, mettre à jour les informations du profil.

4. Service de Gestion des Paniers : Gère les paniers d'achat temporaires des utilisateurs.

- **Nom** : `CartService`
- **Fonctionnalités** : Ajouter un produit au panier, supprimer un produit du panier, obtenir le contenu du panier.

5. Service de Gestion des Paiements : Gère les paiements et les transactions financières.

- **Nom** : `PaymentService`
- **Fonctionnalités** : Traiter les paiements, vérifier la disponibilité des fonds, confirmer le paiement.

Illustrations :

- Un utilisateur visite le site, recherche un produit et ajoute un article à son panier. Le service `CartService` gère l'ajout du produit au panier.
- L'utilisateur souhaite passer une commande. Le service `OrderService` crée une nouvelle commande, vérifie la disponibilité des produits dans le panier via le service `ProductService`, puis met à jour les stocks.
- L'utilisateur souhaite mettre à jour son adresse de livraison. Le service `UserService` gère la mise à jour des informations du profil de l'utilisateur.
- L'utilisateur effectue un paiement pour sa commande. Le service `PaymentService` traite le paiement et confirme la transaction.
- Un administrateur souhaite ajouter un nouveau produit à la boutique en ligne. Le service `ProductService` gère la création du produit et met à jour la liste des produits disponibles.

Granularité des services : comment déterminer leur taille

La granularité des services, c'est-à-dire leur taille et leur étendue, est un élément crucial dans la conception des microservices. Une granularité inappropriée peut entraîner des problèmes d'évolutivité, de maintenance et de complexité. Voici quelques conseils pour déterminer la granularité des services :

1. Cohérence Métier : Les services doivent correspondre à des concepts métier cohérents et clairement définis. Évitez de créer des services trop petits ou trop grands qui ne reflètent pas efficacement les fonctionnalités et les responsabilités.

2. Principe de Responsabilité Unique : Chaque service devrait avoir une seule responsabilité claire et bien définie. Si un service fait trop de choses, il peut devenir difficile à maintenir et à évoluer.

3. Éviter les Microservices "Nano" : Créer des microservices extrêmement petits peut entraîner une surcharge de gestion et de communication. Si un microservice n'offre pas de valeur significative, il pourrait être combiné avec d'autres services.

4. Scénarios d'Utilisation : Analysez comment les utilisateurs interagissent avec votre application. Identifiez les flux d'utilisateurs et considérez si un service unique pourrait répondre à ces flux de manière cohérente.

5. Partage de Données : Si plusieurs services nécessitent des données similaires, cela peut indiquer une granularité incorrecte. Les microservices devraient éviter de partager trop de données, car cela peut entraîner des problèmes de cohérence.

6. Testabilité : Chaque microservice doit être testable de manière isolée. Une granularité inappropriée peut rendre les tests plus difficiles à réaliser.

7. Flexibilité : Les microservices doivent être suffisamment petits pour être déployés, mis à jour et évolués indépendamment. Une granularité appropriée favorise cette flexibilité.

8. Communication Entre Services : Si la communication entre microservices est trop fréquente et complexe, cela peut indiquer une granularité incorrecte. Les interactions entre services doivent être minimisées et bien définies.

9. Taille de l'équipe : La taille et la complexité d'un microservice peuvent influencer la taille de l'équipe qui le gère. Des microservices trop grands peuvent nécessiter des équipes plus importantes pour les développer et les maintenir.

10. Évolutivité : Les microservices doivent être suffisamment petits pour permettre une mise à l'échelle ciblée. Si un microservice devient un goulot d'étranglement en raison de sa taille, il peut être nécessaire de le diviser.

En fin de compte, il n'y a pas de taille de microservice universelle qui convienne à toutes les applications. C'est un équilibre subtil qui nécessite une compréhension approfondie de votre domaine métier, de vos besoins et des avantages attendus des microservices. La flexibilité, l'évolutivité et la facilité de maintenance devraient être au cœur de votre décision concernant la granularité des services.

Définir les frontières de contexte (bounded context)

Les frontières de contexte, également connues sous le nom de bounded contexts en anglais, sont un concept clé dans la conception de l'architecture microservices, notamment dans le contexte du modèle Domain-Driven Design (DDD). Les bounded contexts délimitent les limites et la signification d'un domaine métier spécifique au sein de votre application. Voici comment définir les bounded contexts :

1. Identifier les Domaines Métier : Identifiez les différents domaines métier au sein de votre application. Chaque domaine métier deviendra un bounded context. Par exemple, la

gestion des produits, la gestion des commandes et la gestion des utilisateurs peuvent être des domaines distincts.

2. Définir les Limites : Pour chaque domaine métier, définissez clairement les limites ou les frontières qui le séparent des autres domaines. Ces limites déterminent où chaque domaine commence et où il se termine.

3. Clarifier la sémantique : À l'intérieur de chaque bounded context, clarifiez la signification des termes, des concepts et des règles spécifiques à ce domaine. Ces termes peuvent avoir des significations différentes dans d'autres bounded context.

4. Délimitation Technique : Les bounded contexts ne se limitent pas seulement aux concepts métier, mais peuvent également englober des aspects techniques tels que les bases de données, les interfaces utilisateur, etc.

5. Communication entre Bounded Contexts : Lorsqu'il y a une interaction entre deux bounded context, définissez clairement les interfaces et les contrats pour la communication. Utilisez des événements, des API ou d'autres mécanismes selon les besoins.

6. Taille des Bounded Contexts : Évitez de créer des bounded context trop vastes ou trop petits. Un bounded context doit représenter un ensemble cohérent de fonctionnalités métier. Si un contexte devient trop complexe, envisagez de le diviser en sous-contextes.

7. Cohérence à l'intérieur du Bounded Context : À l'intérieur d'un bounded context, assurez-vous que les règles, les termes et les concepts sont cohérents et clairement compréhensibles.

8. Documenter les Bounded Contexts : Fournissez une documentation détaillée pour chaque bounded context, en expliquant ses responsabilités, ses limites, sa sémantique et ses interactions.

9. Éviter les Anti-Patterns : Évitez les anti-patterns tels que le "big ball of mud" (tout-en-un) où tous les domaines sont mélangés, ou la "distributed monolith" où chaque bounded context est trop petit et interconnecté.

10. Revue Continue : Les bounded contexts ne sont pas figés et peuvent évoluer avec le temps en fonction des besoins de l'application. Assurez-vous de les revisiter régulièrement pour garantir leur pertinence.

En définissant les bounded contexts, vous établissez des limites claires entre les différents domaines métier de votre application. Cela favorise la compréhension, la collaboration et la gestion des interactions complexes dans une architecture microservices.

Exemple 1 : Application de Commerce Électronique

Bounded Context 1 : Gestion des Produits

- **Responsabilités** : Gérer les informations sur les produits, les stocks et les prix.
- **Limites** : Inclut la création, la mise à jour et la suppression des produits.
- **Exemples de Termes** : Produit, Stock, Prix.
- **Interfaces** : API pour ajouter/éditer/supprimer des produits.

Bounded Context 2 : Gestion des Commandes
- **Responsabilités** : Gérer les commandes passées par les utilisateurs.
- **Limites** : Inclut la création de commandes, la gestion de l'état des commandes.
- **Exemples de Termes** : Commande, Ligne de Commande, Statut de Commande.
- **Interfaces** : API pour créer/annuler/obtenir des détails de commande.

Bounded Context 3 : Gestion des Utilisateurs
- **Responsabilités** : Gérer les informations et les actions liées aux utilisateurs.
- **Limites** : Inclut la création de comptes utilisateurs, la gestion des profils.
- **Exemples de Termes** : Utilisateur, Profil, Authentification.
- **Interfaces** : API pour créer/modifier des utilisateurs, gérer l'authentification.

Exemple 2 : Plateforme de Médias Sociaux

Bounded Context 1 : Gestion des Profils Utilisateurs
- **Responsabilités** : Gérer les profils des utilisateurs, leurs informations et leurs interactions.
- **Limites** : Inclut la création de profils, les informations personnelles.
- **Exemples de Termes** : Profil, Utilisateur, Amis.
- **Interfaces** : API pour créer/modifier des profils, gérer les interactions sociales.

Bounded Context 2 : Flux d'Actualités
- **Responsabilités** : Gérer le flux d'actualités personnalisé pour chaque utilisateur.
- **Limites** : Inclut la création de publications, la gestion du contenu.
- **Exemples de Termes** : Publication, Commentaire, J'aime.
- **Interfaces** : API pour créer/lire/modifier des publications et des commentaires.

Bounded Context 3 : Messagerie Instantanée
- **Responsabilités** : Gérer les conversations et les messages entre utilisateurs.
- **Limites** : Inclut la création de conversations, l'envoi/réception de messages.
- **Exemples de Termes** : Conversation, Message, Destinataire.
- **Interfaces** : API pour créer/afficher des conversations, envoyer/recevoir des messages.

Ces exemples illustrent comment définir les bounded contexts en fonction des responsabilités métier distinctes. Chaque contexte a des limites claires et des termes spécifiques, et les interactions entre les contextes peuvent être gérées à l'aide d'API bien définies. Cette approche favorise la modularité, la compréhension et la gestion de chaque aspect du système de manière indépendante.

Patterns de communication entre services : API REST, gRPC, événements

Les patterns de communication entre les services sont essentiels pour déterminer comment les microservices interagissent dans une architecture distribuée. Trois patterns couramment utilisés sont les API REST, gRPC et les événements.

1. API REST (Representational State Transfer) :

L'API REST est un style architectural pour la création de services web basés sur des ressources. Chaque ressource est identifiée par une URL et peut être manipulée à l'aide de méthodes HTTP (GET, POST, PUT, DELETE, etc.). Les services REST sont stateless, ce qui signifie qu'ils ne conservent pas d'état entre les requêtes.

- **Avantages** :
    - Simplicité et facilité d'utilisation.
    - Utilisation des standards HTTP.
    - Grande compatibilité avec différents clients (navigateurs, applications mobiles, etc.).
    - Supporte l'évolutivité grâce à la séparation client-serveur.

- **Inconvénients** :
    - Peut nécessiter plusieurs appels pour obtenir toutes les données nécessaires.
    - Ne convient pas aux besoins de communication asynchrone.

2. gRPC :

gRPC est un framework de communication à distance qui utilise le protocole HTTP/2 pour la communication. Il prend en charge plusieurs langages de programmation et permet la définition de contrats via le langage IDL (Interface Definition Language). gRPC prend en charge des appels de procédure à distance (RPC) pour permettre aux microservices de s'appeler mutuellement.

- **Avantages** :
    - Haute performance grâce à HTTP/2 et à la sérialisation binaire.
    - Génération automatique de code à partir des contrats IDL.
    - Prise en charge de différents langages de programmation.
    - Supporte la communication synchrone et asynchrone.

- **Inconvénients** :
    - Peut nécessiter plus de configuration et de complexité par rapport à REST pour les projets plus simples.
    - Nécessite la définition préalable des contrats.

3. Événements :

La communication basée sur des événements implique que les microservices s'envoient des messages asynchrones pour signaler les changements ou les événements qui se produisent. Cela peut être réalisé à l'aide de files d'attente (comme RabbitMQ ou Apache Kafka) ou d'autres mécanismes de messagerie.

- **Avantages** :
    - Désaccouple les services, ce qui permet une évolution indépendante.
    - Prise en charge de la communication asynchrone.
    - Facilite la mise en œuvre de flux de travail et de traitement en temps réel.

- **Inconvénients** :
    - Gestion complexe des messages, de la répétition et de l'ordonnancement.
    - Peut nécessiter une architecture de messagerie dédiée.

Le choix du pattern de communication dépendra des besoins spécifiques de votre application. En général, les API REST sont simples et bien adaptées aux interactions client-serveur, tandis que gRPC offre des performances élevées et une génération de code automatique. Les événements sont utiles pour la communication asynchrone et la gestion des événements dans des systèmes distribués. Souvent, une combinaison de ces patterns peut être utilisée pour répondre à différents scénarios de communication.

Communication et Coordination

Communication synchrone vs. asynchrone

La communication synchrone et asynchrone sont deux approches distinctes pour échanger des informations entre microservices dans une architecture distribuée. Chaque approche a ses avantages et ses inconvénients, et le choix dépendra des besoins spécifiques de votre application.

Communication Synchrone :

Dans la communication synchrone, le service appelant attend une réponse immédiate du service appelé. Cela signifie que le service appelant bloque son exécution jusqu'à ce qu'il reçoive une réponse du service appelé.

- **Avantages** :
    - Facilité de mise en œuvre et de compréhension.
    - Idéal pour les opérations simples et rapides.
- **Inconvénients** :

    - Peut causer des retards si le service appelé est lent ou indisponible.
    - Risque de créer des goulets d'étranglement si plusieurs services attendent une réponse simultanément.

Exemple :
Supposons qu'un service de gestion de commandes synchronise avec un service de gestion de stocks pour vérifier la disponibilité d'un produit avant de passer une commande. Le service de gestion de commandes envoie une requête au service de gestion de stocks et attend la réponse pour savoir si le produit est disponible.

Communication Asynchrone :

Dans la communication asynchrone, le service appelant envoie une requête au service appelé mais ne bloque pas son exécution pour attendre une réponse immédiate. Le service appelant continue son exécution et peut gérer d'autres tâches pendant ce temps.

- **Avantages** :
    - Ne bloque pas le service appelant, ce qui peut améliorer la performance.
    - Idéal pour les opérations qui prennent du temps ou pour éviter les goulets d'étranglement.
- **Inconvénients** :
    - Plus complexe à mettre en œuvre et à suivre.
    - Peut nécessiter des mécanismes pour gérer les réponses et les erreurs.

Exemple :
Dans un système de traitement de commandes, lorsqu'un utilisateur passe une commande, le service de gestion de commandes peut envoyer un message asynchrone à un service de facturation pour générer la facture. Le service de gestion de commandes n'attend pas la facture immédiatement, ce qui lui permet de continuer à fonctionner sans délai.

En résumé, la communication synchrone convient aux interactions simples et rapides, tandis que la communication asynchrone est préférable pour les opérations plus complexes ou lorsqu'il est important d'éviter le blocage. Dans de nombreux systèmes microservices, une combinaison de ces deux approches est utilisée pour répondre aux différents scénarios de communication.

Considérations sur les API : endpoints, formats de données, versioning

Les API (Interfaces de Programmation Applicative) jouent un rôle crucial dans l'architecture microservices en permettant aux services de communiquer entre eux de manière structurée et normalisée. Voici quelques considérations importantes concernant les API dans le contexte des microservices :

Endpoints (Points d'Extrémité) :

Les endpoints sont les points d'accès à vos services via l'API. Ils doivent être soigneusement conçus pour correspondre aux fonctionnalités offertes par chaque service.

- **Simplicité** : Les endpoints doivent être intuitifs et faciles à comprendre pour les développeurs qui les utilisent.
- **Noms Sémantiques** : Utilisez des noms de endpoints qui reflètent clairement la fonction ou la ressource qu'ils exposent.
- **Pluralisation** : Pour les ressources, préférez la pluralisation (ex : /products) pour une meilleure cohérence.

Exemple :

- Endpoint de gestion des produits : GET /api/products, POST /api/products
- Endpoint de gestion des utilisateurs : GET /api/users/{id}, PUT /api/users/{id}

Formats de Données :

Les formats de données déterminent comment les informations sont échangées entre les services.

- **JSON** : JSON est le format de données le plus couramment utilisé en raison de sa lisibilité et de sa légèreté.
- **XML** : Moins courant que JSON, mais peut encore être pertinent selon les besoins.

Versioning (Versionnage) :

Le versionnage d'une API est crucial pour assurer la compatibilité lors de l'évolution des services.

- **URL Versioning** : Incluez la version de l'API dans l'URL (ex : /v1/products).
- **Header Versioning** : Utilisez un en-tête personnalisé pour spécifier la version (ex : X-API-Version).

Exemple :

- **Versionnement dans l'URL** : GET /api/v1/products, GET /api/v2/products

Gestion des Erreurs :

Les erreurs doivent être gérées de manière appropriée pour fournir des informations utiles aux développeurs consommant l'API.

- **Codes d'Erreur** : Utilisez des codes d'erreur standard pour indiquer les problèmes spécifiques.
- **Messages d'Erreur** : Fournissez des messages d'erreur clairs et informatifs pour aider les développeurs à diagnostiquer les problèmes.

Sécurité :

Assurez-vous que votre API est sécurisée et protégée contre les attaques potentielles.

- **Authentification et Autorisation** : Mettez en œuvre des mécanismes d'authentification (JWT, OAuth) et d'autorisation pour contrôler l'accès.
- **Validation des Données** : Validez les données entrantes pour éviter les failles de sécurité.

Documentation :

Une documentation complète et précise de l'API est essentielle pour aider les développeurs à comprendre comment l'utiliser correctement.

- **Swagger/OpenAPI** : Utilisez des outils comme Swagger ou OpenAPI pour générer automatiquement une documentation interactive.

En suivant ces considérations, vous pouvez créer des API cohérentes, évolutives et bien documentées pour faciliter la communication entre vos microservices.

Utilisation de file d'attente et de systèmes de messages pour la communication

L'utilisation de files d'attente et de systèmes de messagerie est une approche puissante pour faciliter la communication asynchrone entre les microservices dans une architecture distribuée. Cette approche permet de découpler les services, d'améliorer la scalabilité et la résilience, et de gérer des tâches qui nécessitent un traitement différé. Voici comment cela fonctionne et quelques considérations importantes :

Fonctionnement :

1. Producteur (Émetteur) : Un microservice (producteur) envoie un message à une file d'attente en spécifiant le contenu et les détails de la tâche à accomplir.

2. File d'Attente : La file d'attente (ou système de messagerie) agit comme un intermédiaire qui stocke les messages en attente de traitement.

3. Consommateur (Récepteur) : Un autre microservice (consommateur) surveille la file d'attente et récupère les messages pour les traiter.

4. Traitement Asynchrone : Le consommateur traite le message en fonction de son contenu. Ce traitement peut inclure des opérations de calcul, des mises à jour de base de données, des notifications, etc.

Avantages :

- **Découplage** : Les microservices peuvent fonctionner indépendamment, sans avoir à attendre la réponse immédiate d'un autre service.

- **Résilience** : Si un service est temporairement indisponible, les messages restent dans la file d'attente et sont traités dès que le service est opérationnel.

- **Évolutivité** : Vous pouvez ajouter des instances de consommateurs pour traiter les messages plus rapidement en cas de charge élevée.

- **Gestion de Tâches Différées** : Les messages peuvent contenir des tâches qui nécessitent un traitement différé, comme l'envoi d'e-mails ou la génération de rapports.

Considérations :

- **Durabilité des Messages** : Les systèmes de messagerie offrent souvent des options de durabilité pour garantir que les messages ne sont pas perdus en cas de défaillance du système.

- **Ordre des Messages** : La file d'attente ne garantit pas nécessairement l'ordre d'exécution des messages. Si l'ordre est important, vous devez le gérer dans vos services.

- **Gestion des Erreurs** : Prévoyez des mécanismes pour gérer les messages qui échouent lors du traitement, tels que le retrait des messages défectueux ou la gestion des erreurs.

- **Réplication** : Pour garantir la disponibilité, envisagez la réplication de votre système de messagerie pour éviter les points de défaillance uniques.

Exemples d'Utilisation :

- **Notifications** : Un microservice peut envoyer un message de notification à un autre service chargé de la diffusion de notifications aux utilisateurs.

- **Traitement de Commandes en Ligne** : Un service de commande peut envoyer un message à un service de facturation pour générer des factures et à un service de livraison pour organiser la livraison.

- **File d'Attente de Tâches** : Un microservice peut mettre des tâches dans une file d'attente pour un traitement ultérieur, par exemple pour générer des rapports ou envoyer des e-mails.

Gestion des Commandes

Lorsqu'un utilisateur passe une commande dans un système de commerce électronique, le service de gestion des commandes peut publier un message dans une file d'attente, indiquant qu'une nouvelle commande a été passée. Un service de traitement des

commandes surveille ensuite cette file d'attente, récupère le message, et traite la commande (génération de facture, mise à jour des stocks, etc.).

Système de Médias Sociaux

Lorsqu'un utilisateur publie un message sur une plateforme de médias sociaux, un événement est généré. Ce message est ensuite diffusé via un système de messagerie à tous les abonnés de cet utilisateur. Chaque abonné reçoit l'événement, ce qui permet d'afficher les nouveaux messages dans leur flux d'actualités.

Traitement de Batch

Un service de traitement de batch peut utiliser une file d'attente pour gérer l'exécution de tâches lourdes en arrière-plan. Par exemple, l'envoi de notifications par e-mail à tous les utilisateurs peut être géré via une file d'attente. Le service de batch récupère les tâches de la file d'attente et les exécute à un rythme approprié.

Gestion des Stocks

Lorsqu'un service de gestion des stocks reçoit une mise à jour sur les niveaux de stock, il peut publier un événement de stock bas à un système de messagerie. Les services intéressés par cette information, comme le service de gestion des commandes, peuvent souscrire à cet événement pour prendre des mesures en conséquence.

Traitement de Paiements

Lorsqu'un utilisateur effectue un paiement en ligne, le service de traitement des paiements peut publier un message dans une file d'attente. Ce message déclenche ensuite la sérialisation de la transaction de paiement, y compris la vérification de la validité, l'envoi de confirmations, etc.

L'utilisation de files d'attente et de systèmes de messagerie permet de décomposer les interactions entre les microservices, de gérer efficacement les tâches asynchrones et de garantir la cohérence et la résilience du système, même en cas de pannes temporaires.

Gestion des Données dans les Microservices

La gestion des données dans les microservices est un aspect essentiel pour assurer la cohérence, la disponibilité et la performance de votre système. Étant donné que chaque microservice peut avoir sa propre base de données et sa propre responsabilité, il est important de considérer plusieurs facteurs pour garantir une gestion efficace des données.

1. Base de Données Par Service :

Chaque microservice devrait avoir sa propre base de données qui correspond à ses besoins métier spécifiques. Cela permet de découpler les données et de garantir qu'un microservice ne modifie que les données qui lui sont propres.

2. Communication Entre Microservices :

Lorsqu'un microservice doit accéder aux données d'un autre microservice, évitez de manipuler directement la base de données de ce dernier. Utilisez plutôt des API pour communiquer et récupérer les données nécessaires.

3. Découplage des Bases de Données :

Chaque microservice doit avoir le contrôle total sur sa propre base de données. Évitez de partager des schémas ou des tables entre les microservices, car cela peut entraîner des dépendances indésirables.

4. Considérations sur les Modèles de Données :

Les modèles de données doivent être conçus en fonction des besoins spécifiques de chaque microservice. Évitez de normaliser les données à l'extrême, car cela pourrait entraîner des performances médiocres lors de la récupération de données entre différents microservices.

5. API pour l'Accès aux Données :

Chaque microservice doit fournir des API bien définies pour accéder à ses données. Cela permet de contrôler l'accès aux données et d'éviter les accès directs à la base de données.

6. Gestion des Relations :

Si des relations existent entre les données de différents microservices, envisagez d'utiliser des identifiants uniques (UUID) pour éviter les dépendances rigides.

7. Systèmes de Cache :

Utilisez des systèmes de cache pour stocker temporairement les données fréquemment utilisées et réduire les requêtes à la base de données.

8. Stratégies de Mise à Jour :

Lorsqu'une mise à jour de données est nécessaire, envisagez d'utiliser des modèles comme le **"Saga Pattern"** pour gérer les étapes de mise à jour de manière cohérente.

Le **Saga Pattern** permet de gérer une série d'étapes ou d'opérations qui doivent être exécutées de manière cohérente, même en présence de défaillances ou d'erreurs. Chaque

étape du saga représente une action qui doit être effectuée. Si une étape échoue, le saga doit être capable d'annuler les étapes précédentes pour maintenir la cohérence du système.

Exemple : Prenons l'exemple d'une commande en ligne. Le processus de traitement d'une commande implique plusieurs étapes telles que la vérification des stocks, le traitement de paiement, l'emballage et l'expédition. Si l'une de ces étapes échoue, il est nécessaire d'annuler toutes les étapes précédentes pour éviter des incohérences dans le système.

La gestion des données dans les microservices peut être complexe, mais en suivant ces considérations et en adaptant vos choix en fonction des besoins spécifiques de votre application, vous pouvez créer un système efficace et évolutif tout en maintenant l'intégrité des données.

Modèles de base de données : Database per Service, Database per Context

Les modèles de base de données **"Database per Service"** (Base de données par Service) et **"Database per Context"** (Base de données par Contexte) sont deux approches pour gérer les données dans une architecture microservices. Chacun de ces modèles a ses avantages et ses inconvénients, et le choix dépend des besoins et des objectifs de votre application.

1. Database per Service (Base de données par Service) :

Le modèle **"Database per Service"** est une approche architecturale qui préconise l'utilisation d'une base de données dédiée pour chaque service au sein d'une application distribuée. Cela permet à chaque service d'être autonome et de gérer ses propres données de manière indépendante.

Imaginons que nous développons une application de commerce électronique comprenant les services suivants :

1. **Service de Gestion des Produits** :
    a. Ce service est responsable de la gestion des produits, y compris la création, la modification et la suppression de produits.
    b. Il dispose de sa propre base de données qui stocke les informations sur les produits, telles que les noms, les descriptions, les prix, etc.
2. **Service de Gestion des Commandes:**
    a. Ce service gère les commandes passées par les clients, y compris la création de nouvelles commandes, leur suivi et leur traitement.
    b. Il possède sa propre base de données pour stocker les détails des commandes, tels que les articles commandés, les adresses de livraison, etc.
3. **Service de Gestion des Utilisateurs :**

    a. Ce service est en charge de la gestion des utilisateurs, y compris l'authentification, la création de comptes, la gestion des profils, etc.
    b. Il a sa propre base de données qui stocke les informations sur les utilisateurs, comme les noms, les adresses e-mail, les mots de passe (cryptés), etc.

Avantages :
- **Isolation des données et des responsabilités** : Les données d'un service ne sont pas directement accessibles par d'autres services, ce qui réduit les dépendances, favorise une conception modulaire et décentralisée de l'application.
- **Indépendance technologique** : Chaque service peut choisir la base de données qui convient le mieux à ses besoins. Par exemple, le service de gestion des produits pourrait utiliser une base de données NoSQL pour stocker des données non structurées, tandis que le service de gestion des commandes pourrait opter pour une base de données relationnelle.
- **Évolutivité** : Chaque service peut être mis à l'échelle indépendamment en fonction de ses besoins. Par exemple, si le service de gestion des commandes connaît une forte demande, il peut être mis à l'échelle sans affecter les autres services.

Inconvénients :
- **Consistance des Données** : En cas de besoin de données partagées entre les services, des mécanismes de synchronisation ou de communication entre services sont nécessaires.
- **Complexité de Configuration** : La gestion de plusieurs bases de données peut être complexe et nécessite une planification minutieuse.
- **Gestion des Transactions** : Les transactions impliquant plusieurs services peuvent être plus complexes à gérer et peuvent nécessiter l'utilisation de modèles tels que le "Saga Pattern".

2. Database per Context (Base de données par Contexte) :

Le modèle **"Database per Context"** est une approche architecturale qui suggère d'utiliser une seule base de données partagée par plusieurs services ou composants d'une application. Cela signifie qu'une base de données est partagée entre plusieurs fonctionnalités ou contextes de l'application.

Imaginons que nous développons une application de gestion d'événements avec les fonctionnalités suivantes :

1. **Gestion des Événements :**
    a. Ce service gère la création, la modification et la suppression d'événements. Il traite également les inscriptions d'utilisateurs à des événements.
2. **Gestion des Utilisateurs :**
    a. Ce service gère les utilisateurs, leur authentification, leur inscription à des événements, etc.
3. **Gestion des Commentaires :**
    a. Ce service permet aux utilisateurs de laisser des commentaires sur les événements.

Dans ce cas, les trois services partagent une même base de données, qui contient plusieurs tables :

- Une table **"Événements"** pour stocker les informations sur les événements tels que les titres, les dates, les lieux, etc.
- Une table **"Utilisateurs"** pour gérer les informations sur les utilisateurs, comme les noms, les adresses e-mail, les mots de passe, etc.
- Une table **"Commentaires"** pour stocker les commentaires associés à chaque événement.

Avantages :
- **Meilleure cohérence** : Les données qui appartiennent à un contexte spécifique sont regroupées dans une base de données, ce qui facilite la gestion de la cohérence.
- **Requêtes simplifiées** : Les requêtes spécifiques à un contexte sont plus simples à réaliser car les données nécessaires sont dans la même base de données.
- **Communication asynchrone** : Les contextes peuvent communiquer via des événements sans se soucier des implications sur les données.
- **Facilité de Gestion** : Une seule base de données à gérer peut être plus simple à administrer et à surveiller.

Inconvénients :
- **Dépendances entre Services** : Les services peuvent devenir dépendants les uns des autres, ce qui peut compliquer le déploiement et la mise à l'échelle indépendante.
- **Impact des changements** : Les modifications dans un service peuvent potentiellement affecter d'autres services qui partagent la même base de données.
- **Isolation des Données** : Les données sont partagées, ce qui signifie qu'il faut être attentif à l'isolation des données pour éviter les conflits.

Sécurité dans les Microservices

La sécurité est un aspect critique dans les microservices, et la mise en œuvre appropriée de l'authentification, de l'autorisation et de la gestion des tokens est essentielle pour protéger vos services et vos données.

1. Authentification :

L'authentification est le processus de vérification de l'identité d'un utilisateur ou d'un service qui souhaite accéder à un microservice. Les méthodes courantes d'authentification incluent :

- **Authentification basée sur les Tokens** : L'authentification basée sur les tokens, comme JWT (JSON Web Token), est populaire pour les environnements distribués. Les utilisateurs ou les services obtiennent un token d'authentification après s'être connectés à un service d'authentification central. Ce token est inclus dans chaque requête pour prouver l'identité de l'utilisateur ou du service.

- **Authentification à Deux Facteurs (2FA)** : Ajoutez une couche de sécurité en demandant une deuxième forme d'authentification, comme un code SMS, en plus du mot de passe.

2. Autorisation :

L'autorisation détermine les ressources et les actions auxquelles un utilisateur ou un service est autorisé à accéder. C'est le contrôle des droits d'accès aux ressources.

- **RBAC (Role-Based Access Control)** : Attribuez des rôles à chaque utilisateur ou service et accordez des permissions en fonction de ces rôles.

- **ABAC (Attribute-Based Access Control)** : Autorisez l'accès en fonction des attributs spécifiques de l'utilisateur, comme son rôle, son département, etc.

3. Gestion des Tokens :

La gestion des tokens, comme les JWT, est cruciale pour sécuriser vos microservices et vos communications. Voici quelques bonnes pratiques :

- **Durée de Validité** : Définissez une durée de validité appropriée pour les tokens. Trop long, c'est risqué, trop court, c'est contraignant pour les utilisateurs.

- **Signature** : Signez les tokens avec des clés secrètes pour empêcher la falsification.

- **Révocation** : Mettez en place un mécanisme de révocation pour invalider les tokens avant leur expiration, en cas de compromission.

- **Rafraîchissement** : Utilisez des tokens de rafraîchissement pour éviter de demander fréquemment des informations d'identification aux utilisateurs.

- **Stockage** : Stockez les tokens de manière sécurisée, de préférence dans des cookies HTTPOnly pour les cookies de session et dans des variables sécurisées pour les tokens côté client.

- **Centralisation** : Utilisez un service d'authentification centralisé pour générer et valider les tokens, ce qui facilite la gestion et la mise en œuvre cohérente de la sécurité.

La mise en œuvre de ces mesures de sécurité aidera à prévenir les attaques et à maintenir l'intégrité et la confidentialité de vos microservices. Il est recommandé de suivre les meilleures pratiques de sécurité et d'utiliser des bibliothèques et des frameworks de sécurité éprouvés pour faciliter la mise en œuvre.

Meilleures Pratiques

Meilleures pratiques pour la conception de microservices réussis

Concevoir des microservices réussis implique de suivre des meilleures pratiques qui garantissent la modularité, la scalabilité, la résilience et la facilité de gestion de votre architecture. Voici quelques meilleures pratiques avec des exemples concrets pour vous aider dans la conception de microservices réussis :

1. Découpage Basé sur les Domaines Métier :

Concevez vos microservices en fonction des domaines métier, en délimitant les responsabilités et les fonctionnalités spécifiques à chaque service.

Exemple : Dans une application de commerce électronique, séparez les microservices de gestion des produits, de gestion des commandes et de gestion des paiements pour encapsuler les fonctionnalités spécifiques à chaque domaine.

2. Cohérence des Interfaces :

Définissez des interfaces claires et cohérentes pour les interactions entre les microservices, en utilisant des contrats bien définis.

Exemple : Pour les microservices de communication, utilisez une API REST avec des méthodes standardisées (GET, POST, PUT, DELETE) pour faciliter la compréhension et l'utilisation.

3. Isolation des Données :

Chaque microservice devrait avoir sa propre base de données dédiée pour éviter les dépendances et garantir l'isolation des données.

Exemple : Dans une application de médias sociaux, séparez les microservices de gestion des profils, des messages et des amis, chacun avec sa propre base de données.

4. Communication Légère :

Utilisez des mécanismes de communication légers tels que les appels d'API HTTP, les messages asynchrones ou les événements pour éviter les dépendances lourdes.

Exemple : Utilisez Kafka pour envoyer des événements de mise à jour de statut dans une application de streaming en temps réel.

5. Automatisation du Déploiement :

Automatisez le déploiement, la mise à l'échelle et la gestion des microservices à l'aide d'outils de conteneurisation et d'orchestration comme Docker et Kubernetes.

Exemple : Déployez vos microservices dans des conteneurs Docker et gérez-les avec Kubernetes pour une scalabilité automatique.

6. Surveillance et Gestion des Erreurs :

Mettez en place des mécanismes de surveillance pour détecter les erreurs et les problèmes de performance en temps réel.

Exemple : Utilisez des solutions comme Prometheus et Grafana pour surveiller les métriques et les performances de vos microservices.

7. Conception pour la Résilience :

Anticipez les pannes en concevant vos microservices pour qu'ils gèrent les erreurs de manière gracieuse et puissent récupérer rapidement.

Exemple : Utilisez des stratégies de retry avec des mécanismes de circuit breaker pour gérer les appels aux services tiers.

8. Tests Complets et Continus :

Assurez-vous de tester chaque microservice individuellement avec des tests unitaires, d'intégration et de bout en bout pour garantir la qualité.

Exemple : Automatisez les tests avec des frameworks comme Jest pour les tests unitaires JavaScript ou JUnit pour Java.

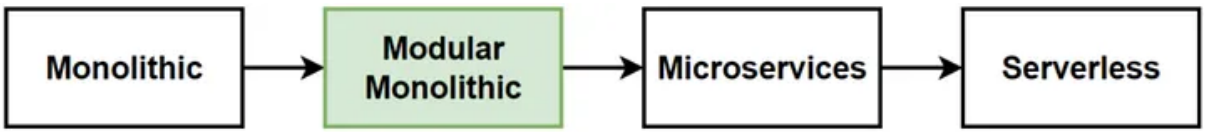
9. Documentation Clair :

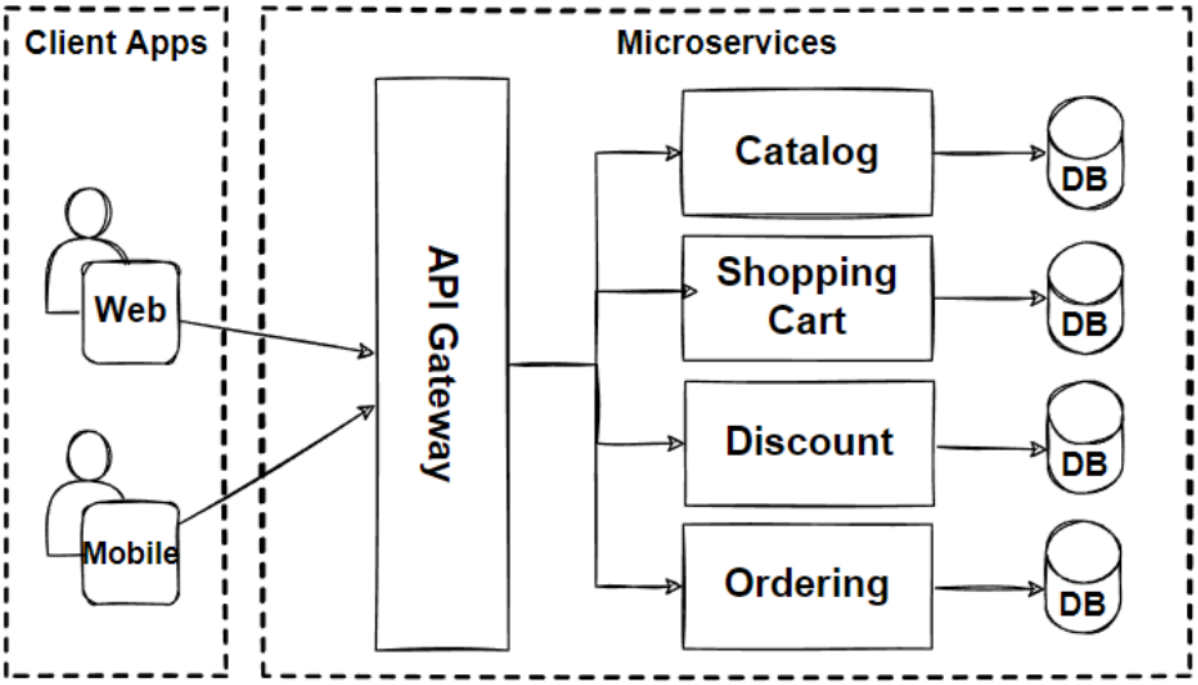
Fournissez une documentation claire et concise pour chaque microservice, y compris les dépendances, les API et les processus d'intégration.

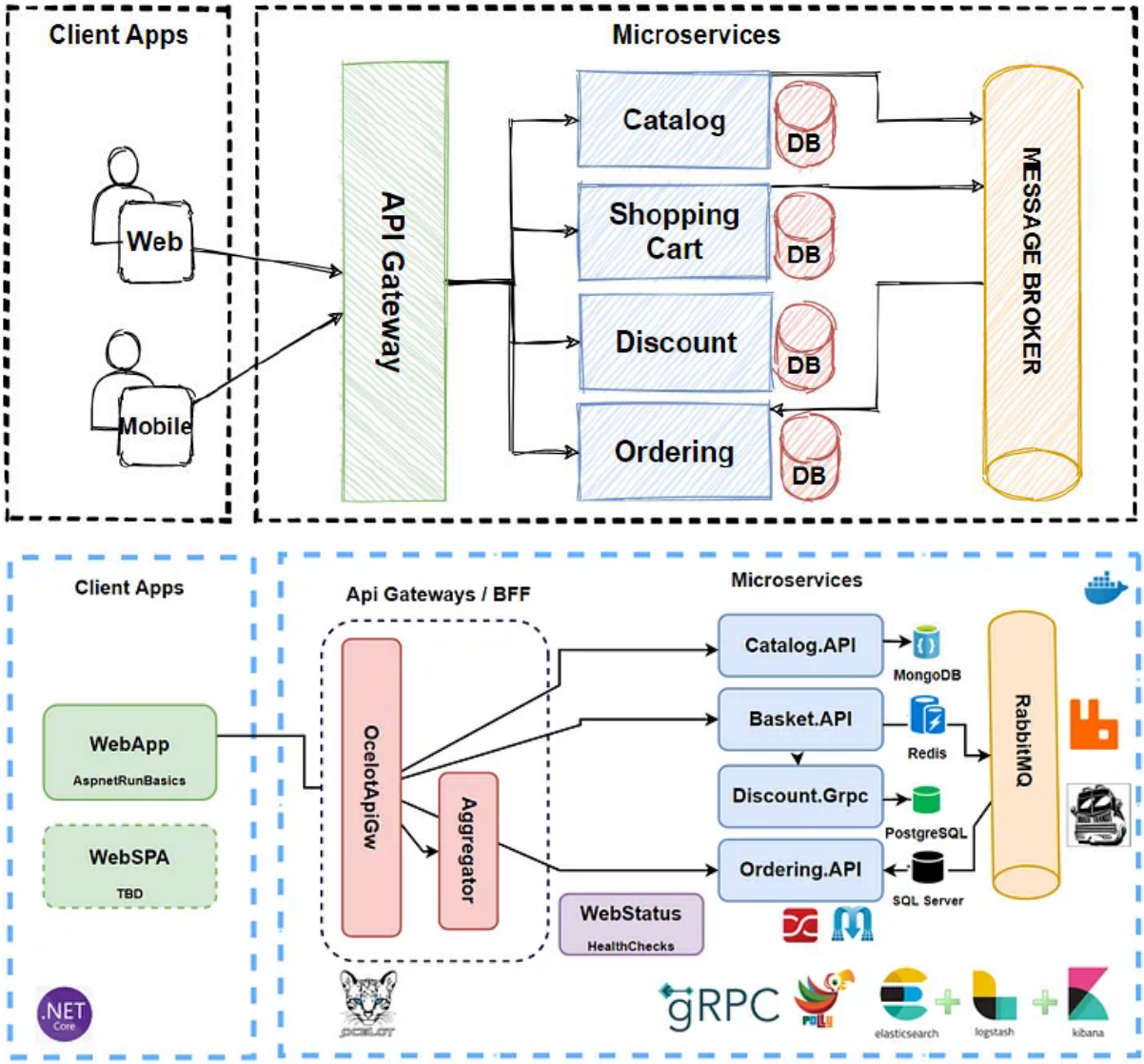
Exemple : Utilisez des outils comme Swagger pour générer automatiquement une documentation d'API pour vos microservices.

En suivant ces meilleures pratiques, vous créerez des microservices bien conçus qui sont modulaires, évolutifs, résilients et faciles à gérer. Cela permettra à votre architecture de microservices de fonctionner de manière fluide tout en répondant aux exigences métier et en évoluant avec le temps.

Images d’illustrations







Ressources

- https://www.redhat.com/en/topics/microservices/what-are-microservices
- https://www.leanix.net/en/wiki/vsm/microservices-architecture
- https://openclassrooms.com/fr/courses/4668056-construisez-des-microservices/7651431-apprehendez-larchitecture-microservices
- https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-2bec9da7d42a
- https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a