Engineer

Microservices

Architecture de petits services déployables séparément

Ce que c’est

Les microservices sont un style d’architecture divisant une application en petits services. Chacun est un programme à part entière, exécuté dans un processus distinct et déployable sans les autres. James Lewis et Martin Fowler, dont l’article de 2014 en a donné la description la plus connue, le résument comme la construction d’applications en « ensembles de services déployables indépendamment ». Ils listent les traits fréquents : échanges par mécanismes légers, souvent une API de ressources HTTP ; organisation autour des capacités métier ; déploiement entièrement automatisé ; gestion centrale minimale ; liberté de langage et de stockage pour chaque service. Ils précisent qu’il n’existe pas de définition formelle, seulement des caractéristiques communes à la plupart des systèmes portant cette étiquette.

Le mot est plus jeune que l’idée. Une note du même article rapporte qu’un atelier d’architectes logiciels près de Venise a discuté du terme en mai 2011, puis que le même groupe a retenu « microservices » en mai 2012. Adrian Cockcroft, chez Netflix, parlait d’une forme de SOA à granularité fine. Lewis et Fowler ne prétendent pas que le style est nouveau : ses racines remontent au moins aux principes fondateurs d’Unix.

L’alternative habituelle est le monolithe : une application serveur construite et déployée comme une unité. Toute la logique d’une requête tourne dans un processus. Avec du soin, l’application entière peut fonctionner et être testée sur le portable d’un développeur ; elle monte en charge en multipliant ses copies derrière un équilibreur. Les frictions arrivent ensuite. Un petit changement exige de tout reconstruire et redéployer ; une bonne modularité est difficile à garder au fil des années ; et la partie ayant besoin de davantage de capacité ne peut monter en charge qu’avec tout le reste.

Lewis et Fowler définissent un composant comme une unité logicielle pouvant être remplacée et mise à niveau indépendamment. Une bibliothèque est liée à un programme et appelée en mémoire. Un service est un composant hors processus, accessible par le réseau via une requête web ou un appel de procédure distant. La différence compte pour le couplage. Dans la plupart des langages, seules documentation et discipline empêchent d'atteindre les détails internes d'une bibliothèque ; un service n'est accessible que par des appels distants explicites, son interface est donc bien plus difficile à contourner. Il peut néanmoins comprendre plusieurs processus, par exemple une application et sa base de données exclusive.

Le « micro » dit moins qu'il n'y paraît. Lewis et Fowler notent que le nom attire l'attention sur la taille et suscite des débats sur ce qui mérite le mot. Les plus grands services dont ils avaient entendu parler suivaient l'équipe « deux pizzas » d'Amazon, soit au plus une douzaine de personnes ; à l'autre bout, ils avaient vu une demi-douzaine de personnes s'occuper d'une demi-douzaine de services.

Utilisez le concept pour prendre une décision

Utilisez « Que sont les microservices ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

Le premier gain est la liberté du calendrier de livraison. Une application divisée permet souvent de ne redéployer qu'un service : une équipe livre sans attendre les autres. Lewis et Fowler précisent que ce n'est pas absolu : modifier l'interface exige toujours de se coordonner avec les services appelants.

Le deuxième est que le logiciel peut suivre l'organisation plutôt que la combattre. En 1968, Melvin Conway soutenait dans Datamation que les organisations concevant des systèmes sont contraintes de reproduire leurs propres structures de communication. Lewis et Fowler l'appliquent au découpage habituel par couche technique : équipe d'interface, équipe de logique serveur et équipe de base de données. Même un changement simple devient un

projet entre équipes, et la logique atterrit là où une équipe a accès. Les microservices renversent cela. Chaque service couvre une capacité métier, comme commandes, stock ou paiements, de l'interface au stockage, et appartient à une équipe pluridisciplinaire possédant toutes les compétences nécessaires. Cette équipe devrait être responsable du service pendant toute sa durée de vie, plutôt que de le transférer en fin de projet ; chez Amazon, qui construit un service l'exploite aussi en production.

Le troisième gain est que la logique reste dans les services. Lewis et Fowler résument cette préférence par des extrémités intelligentes et des tuyaux simples. Les services échangent par requêtes HTTP à des API de ressources ou par un bus de messages léger ne faisant que les acheminer ; les décisions sont prises aux extrémités. C'est un éloignement délibéré du bus de services d'entreprise, qui route, transforme et applique les règles métier au milieu du réseau.

Les données suivent les mêmes frontières. Chaque service gère sa propre base, et les systèmes de gestion de bases de données peuvent différer selon les besoins. La conception pilotée par le domaine d'Eric Evans en donne la raison : un grand système contient plusieurs modèles, chacun demandant un contexte explicitement délimité selon l'organisation de l'équipe, les parties applicatives régies et les éléments physiques comme bases de code et schémas de bases. La vision d'un client par les ventes peut ainsi différer de celle du support sans devoir les réconcilier dans une seule table.

Les coûts font partie de la définition, et Lewis et Fowler les exposent clairement. Un appel distant coûte bien plus qu'un appel en mémoire : les interfaces entre services doivent donc être moins fines. Déplacer du code à travers une frontière de service est difficile : changements d'interface à coordonner, couches de compatibilité avec les versions antérieures à ajouter, tests plus complexes. Les transactions distribuées sont difficiles à réaliser : les services se coordonnent sans elles, acceptent que la cohérence puisse n'être atteinte qu'à terme et réparent les erreurs par compensation. Tout appel peut échouer. Si les services s'assemblent mal, la complexité n'a pas disparu : elle s'est déplacée dans les connexions, moins visibles. Lewis et Fowler jugent raisonnable l'argument suivant : construire d'abord un monolithe proprement modulaire, puis le diviser seulement lorsqu'il devient gênant. Avec une réserve : une bonne interface interne à un processus fait rarement une bonne interface entre services.

À quoi cela ressemble en pratique

La note Vendre le dernier billet exactement une fois n'introduit un service que lorsqu'une base ne suffit plus, puis pose la règle liant la conception : chaque article a exactement un propriétaire de son stock, le service d'inventaire, et rien d'autre ne décide. Ce service est réparti en fragments (shards) par organisation cliente et article ; les plus grandes organisations ont leurs cellules dédiées. Seule la réservation du stock doit être synchrone. Paiement, commande et reçu sont des systèmes distincts sans transaction commune. La note refuse la validation en deux phases, car elle bloque dès qu'un participant ralentit et que les prestataires de paiement n'y participent pas. Elle utilise une saga : suite d'étapes locales, chacune associée à une étape capable de l'annuler.

Chaque étape écrit son message dans une table d'envoi dans la même transaction que la modification annoncée. Dans PostgreSQL, une instruction modifiant les données dans `WITH` peut transmettre son résultat à une insertion : le service réserve donc une unité et consigne le message en une instruction :

```
-- inventory service, its own database
WITH held AS (
    UPDATE inventory
    SET available = available - 1
    WHERE tenant_id = 't-7' AND item_id = 'item-881'
      AND available >= 1
    RETURNING item_id
)
INSERT INTO outbox (order_id, item_id, event, units)
SELECT 'A-17', item_id, 'StockReserved', 1 FROM held;
-- sold out: no row matches, so no message is written
```

La capture des modifications de données publie ensuite les lignes de cette table d'envoi, et chaque consommateur est idempotent, avec la commande comme clé, car les messages arrivent au moins une fois. Dans cette esquisse, aucun autre service ne touche `inventory` : les autres obtiennent les informations de stock par ces messages ou en interrogeant son propriétaire.

La même note situe aussi la séparation dans l'ordre du travail. Dans ses priorités, donner au stock son service et son propriétaire vient en quatrième, après la mesure, l'écriture ramenée à une instruction conditionnelle et la réduction de charge par salle d'attente, dispositif d'admission et réponse « épuisé » en cache. Les corrections moins coûteuses viennent d'abord.

L'étude de cas Une plateforme maritime construite en couches nettes consigne la contribution architecturale de la fondatrice chez NextMariner et montre l'autre choix. Le produit était jeune, ses exigences allaient beaucoup changer ; modifier une règle métier ne devait pas obliger à toucher simultanément interface, API et base. Trois couches ont été retenues, chacune avec un rôle : interface Next.js pour la présentation, API Go volontairement fine qui route, valide, filtre pour la sécurité et orchestre, et toute la logique métier en fonctions PostgreSQL. Les frontières ont été consignées par écrit et leur respect a été imposé lors des revues. Lewis et Fowler notent que maintenir des frontières modulaires au sein d'une même application demande beaucoup de discipline, et que la séparation plus explicite des services clarifie celles des équipes. Cette plateforme a gardé cette discipline sans répartir sa logique métier entre des services distincts.

Où les problèmes surviennent

Les chaînes d'appels synchrones multiplient les indisponibilités. Si répondre exige qu'un premier service en appelle un deuxième, puis un troisième, la requête ne réussit que si les trois fonctionnent, et elle attend chacun à son tour. Lewis et Fowler parlent de l'effet multiplicateur des indisponibilités et citent la règle du Guardian : un appel synchrone par requête utilisateur. Gardez le chemin aussi court que dans la note : seule la réservation attend une réponse, tout le reste est message. Donnez à chaque appel restant un délai limite et un disjoncteur, les patrons de conception qu'ils citent dans Release It!, pour couper un service défaillant plutôt que l'attendre.

Les services changent toujours ensemble. De mauvaises frontières produisent des services impossibles à livrer seuls : chaque fonctionnalité en touche trois, livrés en bloc. Le test de Lewis et Fowler est simple : si deux services changent sans cesse ensemble, fusionnez-les. Un autre échec transforme chaque appel de méthode en appel distant, créant des services bavards sur le réseau et peu performants. Tracez les frontières selon les capacités métier et contextes délimités, donnez aux interfaces une granularité grossière et concevez chaque service pour tolérer les changements de ceux qu'il appelle : un fournisseur doit évoluer sans casser ses consommateurs. Lewis et Fowler réservent le versionnage au dernier recours.

Deux services partagent une base. Si le service de commande lit et écrit directement les tables de stock, le schéma devient une interface sans propriétaire ; aucune équipe ne peut le changer seule et le stock gagne un deuxième composant qui l'écrit, sans jamais passer par les contrôles du service d'inventaire. Donnez un propriétaire à chaque donnée, comme la note pour le

stock, et demandez aux autres services d'interroger ce propriétaire ou de lire ses messages. Si une modification traverse les services, utilisez une saga avec des étapes de compensation plutôt qu'une transaction distribuée.

Personne ne peut suivre une requête. Une requête traversant plusieurs services laisse un fragment de son histoire dans chaque journal. Lewis et Fowler donnent une place centrale à la supervision en temps réel, observant les nombres techniques, comme les requêtes par seconde vers la base, à côté des nombres métier, comme les commandes par minute. Ils avertissent aussi que les services réagissant aux événements des autres produisent des comportements émergents à détecter vite. Tracez chaque requête. Dans OpenTelemetry, une trace est le chemin d'une requête dans le système, composé de segments partageant un identifiant de trace et pointant vers leur parent ; la propagation du contexte réunit ceux des différents services. Transmettez ce contexte dans chaque appel et message, y compris ceux de la table d'envoi, sinon la trace s'arrête au premier saut asynchrone.

Pour aller plus loin

- Microservices — James Lewis et Martin Fowler, 2014 : neuf caractéristiques, histoire du nom, encadrés sur la taille et les appels synchrones, et raisons de rester prudent.
- Comment les comités inventent-ils ? — l'article de Melvin Conway dans Datamation en 1968 : la structure d'un système copie les communications de l'organisation conceptrice.
- Référence de conception pilotée par le domaine — le résumé des patrons de conception par Eric Evans, dont le contexte délimité et les cartes de leurs relations.
- OpenTelemetry : traces — segments, identifiants de trace, parents et propagation du contexte pour rendre une requête visible entre services.
- PostgreSQL : requêtes WITH — instructions de modification des données dans `WITH`, permettant de modifier une ligne et consigner ailleurs le changement en une instruction.



Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Que sont les microservices ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/microservices/