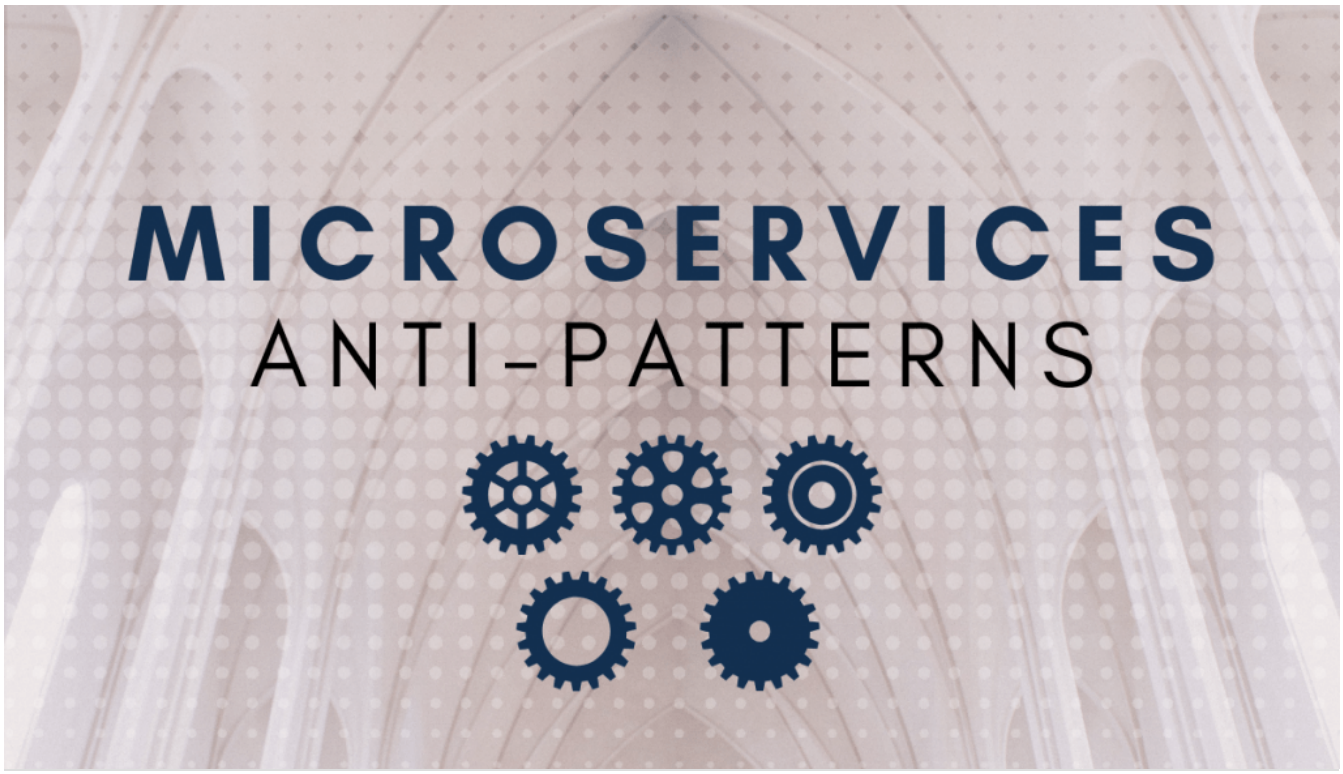
(https://keyholesoftware.com)



Anti-modèles de microservices

DALLAS MONSON (HTTPS://KEYHOLESOFTWARE.COM/AUTHOR/DMONSON/) /
18 MARS 2019 /
AGILE (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/DEV-METHODOLOGY/AGILE/) ,
CONSEIL (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/CONSULTING-2/) ,
MICROSERVICES (HTTPS://KEYHOLESOFTWARE.COM/CATEGORY/ARCHITECTURE/MICR
1 COMMENTAIRE (HTTPS://KEYHOLESOFTWARE.COM/2019/03/18/MICROSERVICES-AN

Attention : L'article suivant a été publié il y a plus de 3 ans, et les informations fournies peuvent être anciennes ou obsolètes.
Veuillez garder cela à l'esprit pendant que vous lisez le message.

Microservices ? Ouais, tu le fais mal.

Les microservices sont une solution miracle, une pilule magique, une solution instantanée et une solution infaillible à tous les problèmes des logiciels. En fait, dès que vous implémentez même les bases des

microservices, tous vos rêves deviennent réalité ; vous triplerez votre productivité, atteindrez votre poids idéal, décrocherez l'emploi de vos rêves, gagnerez 10 fois à la loterie et pourrez voler clairement.

Bien que cela ressemble à beaucoup d'hyperboles enveloppées dans certains BS, si vous avez récemment écouté quoi que ce soit autour des microservices, vous aurez probablement entendu quelque chose de pas trop loin de ce sentiment exagéré - surtout s'il vient des vendeurs.

En conséquence, vous ou quelqu'un que vous connaissez aurez probablement été facturé par la direction pour implémenter une solution dans les microservices ou refactoriser une application existante pour tirer parti des microservices afin de vous assurer que vous obtenez toute la magie. Avec autant de surgonflage de la vérité, il est probable que vous ayez également mis en œuvre un anti-modèle de microservices. Ces anti-modèles sont en fait plus courants dans la nature que les architectures de microservices entièrement fonctionnelles.

Aperçu

Dans cet article, nous couvrirons les anti-modèles les plus courants dont j'ai été témoin dans la nature :

- Casser la tirelire
- Tout Micro (sauf pour les données)
- Nous sommes Agiles ! alias le Frankenstein

Chacun d'eux résulte d'une idée fausse commune. Nous ferons de notre mieux pour définir ces schémas et leurs symptômes. Après chacun, nous montrerons également un moyen de sortir du pétrin afin que vous puissiez récupérer et commencer à avancer vers une meilleure mise en œuvre. Commençons!

Casser la tirelire

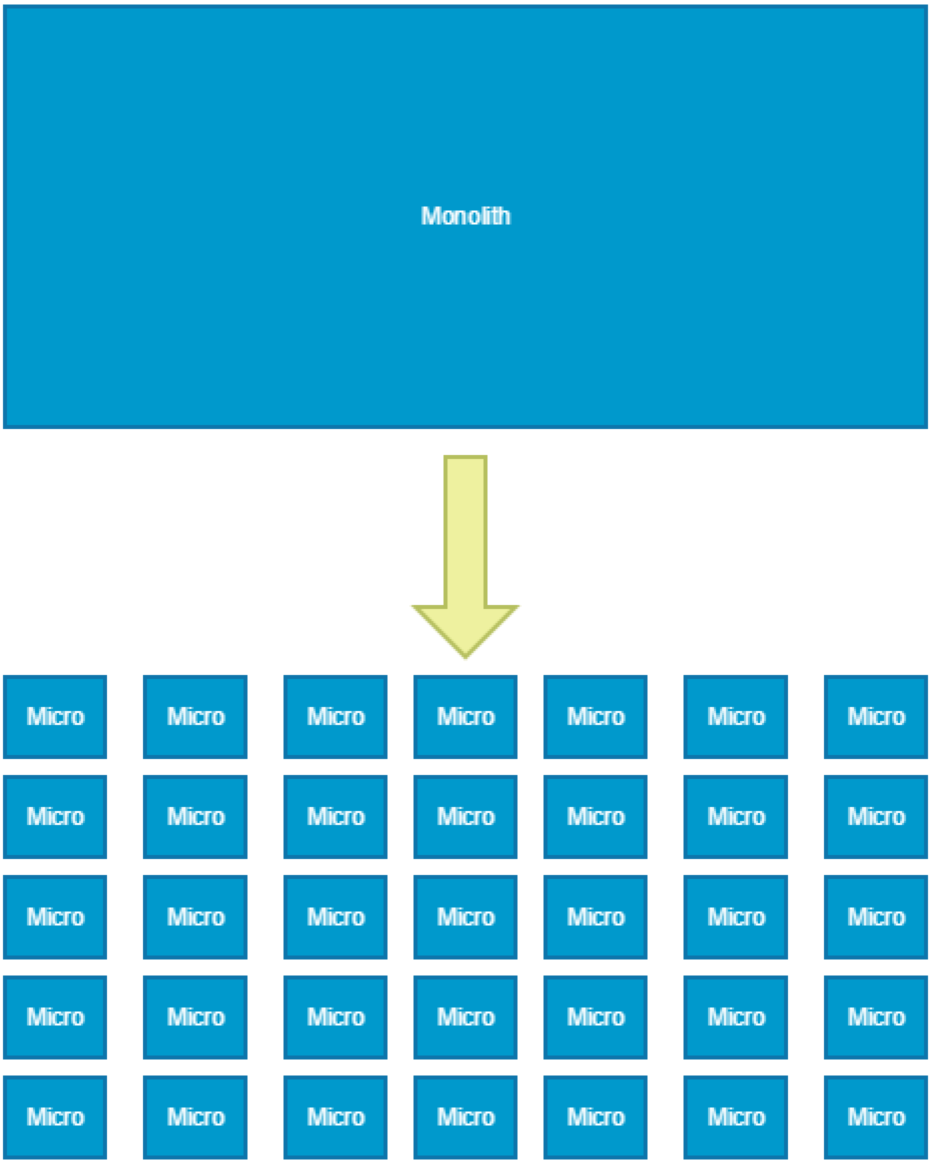
Cet anti-modèle est l'un des plus courants lors de la refactorisation d'une application existante vers une architecture de microservices. Lorsque les applications commencent comme des applications monolithiques et se développent au fil du temps, elles finissent par devenir si volumineuses que même les parties de base du SDLC deviennent des tâches atroces :

- Les déploiements peuvent prendre plusieurs heures, voire plusieurs jours et sont souvent très risqués
- La maintenance devient à la fois rigueur technique et archéologie
- Les performances commencent à être mesurées en « combien de jours depuis une panne sev 1 » ?
- Les tests de régression nécessitent des équipes, de l'automatisation, des centres de données dédiés et une toute nouvelle organisation logicielle

Lorsqu'une application est dans cet état, il est facile d'y penser comme un cochon. Le monolithe est devenu intenable et est désormais un candidat de choix pour les microservices.

Les microservices traitent du déploiement, de la maintenance, des performances et des tests en décomposant la grande base de code en services plus petits et découplés qui communiquent via des « canaux muets » (le plus souvent le protocole HTTP).

Lorsqu'une équipe est initiée à cela, en particulier lorsqu'elle a affaire à un cochon, la première tendance est de briser le tout en petits morceaux, comme une tirelire avec un marteau. Parce que, bien sûr, 1000 services vaudront bien mieux qu'un service pléthorique, n'est-ce pas ? Pas si vite. La fragmentation d'un monolithe en ses parties n'est pas aussi simple qu'un simple *écrasement* .



Le problème avec cette méthode est que sans division intentionnelle des services par domaine, unité de travail ou potentiel de changement, vous finissez par créer plusieurs mini-monolithes. Souvent, les services sont décomposés de manière trop granulaire et ne sont pas vraiment séparés, ils finissent donc par devoir être déployés, mis à l'échelle et maintenus ensemble.

L'autre problème que cela exacerbe est la complexité de l'application globale. Avec le code, la complexité n'est ni créée ni détruite lors du passage d'un monolithe aux microservices, elle passe simplement des appels de service in-proc aux appels HTTP(s) qui ajoutent une toute autre couche. L'orchestration, les transactions distribuées, la

découverte de services et la récupération ne sont que quelques nouvelles préoccupations que les développeurs d'applications doivent prendre en compte, oui.

Une voie à suivre

Si vous vous retrouvez dans cette situation, tout n'est pas perdu. Il existe des solutions mais elles demanderont du travail. La première chose à faire pour résoudre tous les problèmes causés par un "smash de tirelire" est de faire une analyse de votre application et d'essayer de recomposer les limites de service en fonction des domaines.

Cela se traduira généralement par la combinaison de plusieurs services qui ne sont pas vraiment séparés et qui définissent plus clairement vos entités et objets de valeur. Nous ne plongerons pas dans DDD dans cet article, mais il est très utile pour refactoriser des applications qui ont été divisées en trop de morceaux. Pour plus d'informations, consultez le lien suivant :

https://en.wikipedia.org/wiki/Domain-driven_design (https://en.wikipedia.org/wiki/Domain-driven_design)

Une fois que vous disposez de services de taille raisonnable et composés, la mise en œuvre de l'automatisation vous permettra de résoudre les problèmes de déploiement et de création à long terme. Si vous n'avez pas encore implémenté de pipeline CI/CD, c'est le moment. Avec plusieurs options, de Jenkins à TFS, vous pouvez trouver une plate-forme CI/CD qui vous aidera à activer vos efforts d'automatisation. À partir de là, des éléments comme Ansible, SaltStack, Chef/Puppet et d'autres peuvent également vous aider à automatiser l'infrastructure. L'idée avec tout cela est de supprimer l'élément humain des tâches répétitives dans autant de zones de la pile que possible. De cette façon, si quelque chose ne va pas avec vos services nouvellement composés, vous pouvez rapidement

récupérer ou, dans certains cas, reconstruire complètement votre application et vos dépendances avec un processus automatisé. Cela devient une tâche gérable si vous avez une poignée de services,

Même si vous n'automatisez pas entièrement l'ensemble de votre solution et ne suivez pas les principes DDD à la lettre, la recomposition de votre application se traduira par une solution beaucoup plus efficace qui sera maintenable et facilitera généralement les problèmes de déploiement. Dans cet anti-modèle, la clé pour en sortir est de s'assurer que vous ne créez pas de services sur des limites arbitraires, mais que vous les divisez plutôt par unités de travail/domaines afin qu'ils puissent vraiment être déployés, mis à l'échelle et maintenus séparément du reste de L'application.

Tout Micro (sauf pour les données)

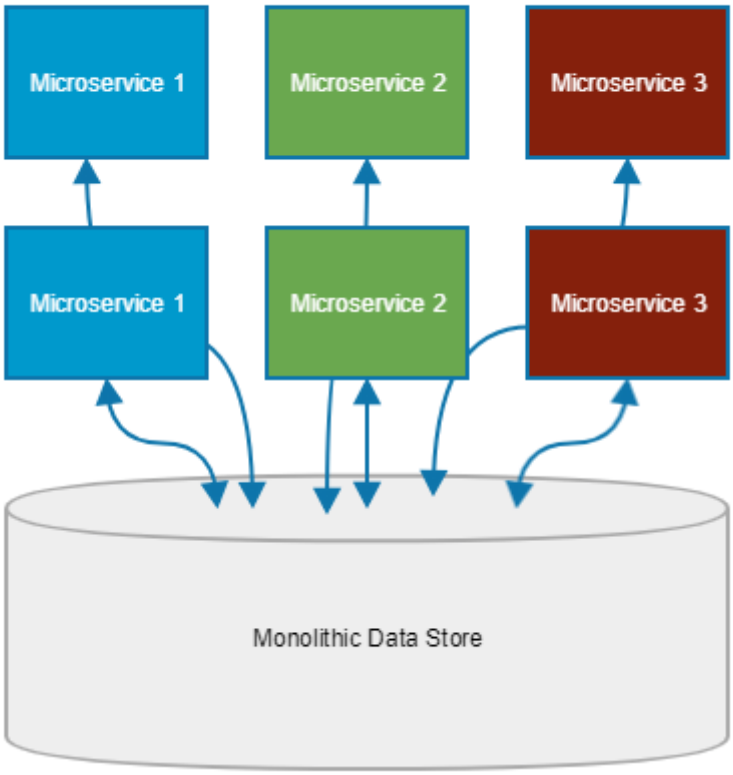
Il s'agit d'un autre anti-modèle extrêmement courant, en particulier dans les organisations d'entreprise. Le plus souvent, cette conception découle d'une certaine forme de dépenses d'investissement dans un centre de données, d'un achat de SGBDR ou d'une équipe de données existante qui n'est pas à l'aise avec les données dans le cloud et/ou les microdonnées en général.

Le signe clé de cet anti-modèle est qu'à peu près tout dans l'espace d'application est décomposé de manière raisonnable, qu'un processus CI/CD mature est en place, peut-être même que certains modèles de conception distribués sont en place. Cependant, il existe un magasin de données géant derrière tous les microservices. Il peut s'agir d'une haute disponibilité et d'une réplication complète, mais il s'agit toujours d'une structure de données monolithique.

Ceux-ci sont les plus courants avec les magasins de données SQL Server, Oracle et DB2 de Microsoft, principalement parce que leurs modèles de licence ne se prêtent pas facilement à l'implémentation d'une base de données par service dans la nature.

Les défis de cette approche sont plus subtils que d'autres anti-modèles, car souvent, cela peut n'avoir un impact négatif notable sur l'application que plus tard dans son cycle de vie. Le suivi des modifications des données/schémas est un défi courant avec cette configuration, car toute modification potentielle d'une base de données de production peut nécessiter une panne complète de la base de données, ou peut provoquer des verrous et des blocages si le système est toujours actif lorsque les modifications sont exécutées.

Il s'agit généralement de problèmes pouvant être résolus qui nécessitent des processus de gouvernance et d'approbation détaillés pour atténuer les problèmes. Les SLA sont également les plus risqués avec cet anti-modèle en raison du potentiel susmentionné de pannes lors de l'exécution des modifications. Avec les grands magasins de données, le contrôle d'accès peut devenir extrêmement complexe car les applications sont généralement limitées dans les schémas/fonctions qu'elles peuvent affecter ainsi que les activités qu'elles peuvent effectuer, afin de ne pas affecter d'autres applications sur le même magasin de données.



Avec plusieurs applications accédant au même magasin de données, les conflits de ressources deviennent finalement le principal problème. Au fur et à mesure que les applications grandissent et se développent, l'empreinte des données augmente également. L'archivage, le nettoyage et le réglage des données deviennent des tâches cruciales pour que tout reste en équilibre. De plus, la taille d'un magasin de données est souvent associée au coût, de sorte que les très grandes instances peuvent éventuellement devenir d'un coût prohibitif à exploiter.

Avec un magasin de données monolithique, l'échelle verticale et une sorte de clustering deviennent le seul moyen possible de résoudre les problèmes de performances. Rien n'est plus amusant que d'essayer de déterminer combien de cartes d'E/S flash vous pouvez bloquer dans un serveur de base de données avant que votre application ne plante et ne bloque complètement la base de données !

Une voie à suivre

Malheureusement, le moyen de sortir de cette situation difficile implique généralement de changer les cœurs et les esprits de votre organisation. Afin de s'éloigner d'un magasin de données monolithique, vous devez commencer petit. Si votre application est déjà composée d'un contexte limité au domaine, il sera plus facile d'en prendre un morceau et de le placer dans un autre magasin de données à usage unique. Si vous n'avez pas implémenté de contexte limité, cela peut être un excellent moyen d'aider à définir ces limites dans votre application.

Pour plus d'informations, veuillez consulter le billet de blog Keyhole sur la mise en œuvre d'un contexte limité :

Implémentation d'un contexte délimité (https://keyholesoftware.com/2016/03/21/im
a-bounded-context/)

Selon la plate-forme de votre choix, vous avez souvent le choix entre plusieurs magasins de données : des stockages basés sur SQL, basés sur des documents ou même un simple stockage blob peuvent faire l'affaire. La clé est d'utiliser un magasin de données qui correspond le mieux à la fonction que votre application essaie d'exécuter.

Si vous avez un domaine spécifique dans votre application qui se concentre sur les données de profil utilisateur, par exemple, une structure SQL normalisée peut ne pas être la meilleure solution. Un magasin de données No-SQL plus plat et basé sur des objets peut être le mieux adapté, car les formes peuvent être flexibles et ne pas nécessiter de mise à jour du modèle de données à mesure que leur contenu change et évolue. L'extraction de ces données spécifiques et

leur migration de votre base de données monolithique vers la base de données No-SQL peuvent être effectuées de plusieurs manières. Avant que cela puisse être accompli, l'organisation doit être convaincue que le nouveau magasin de données sera efficace et pris en charge.

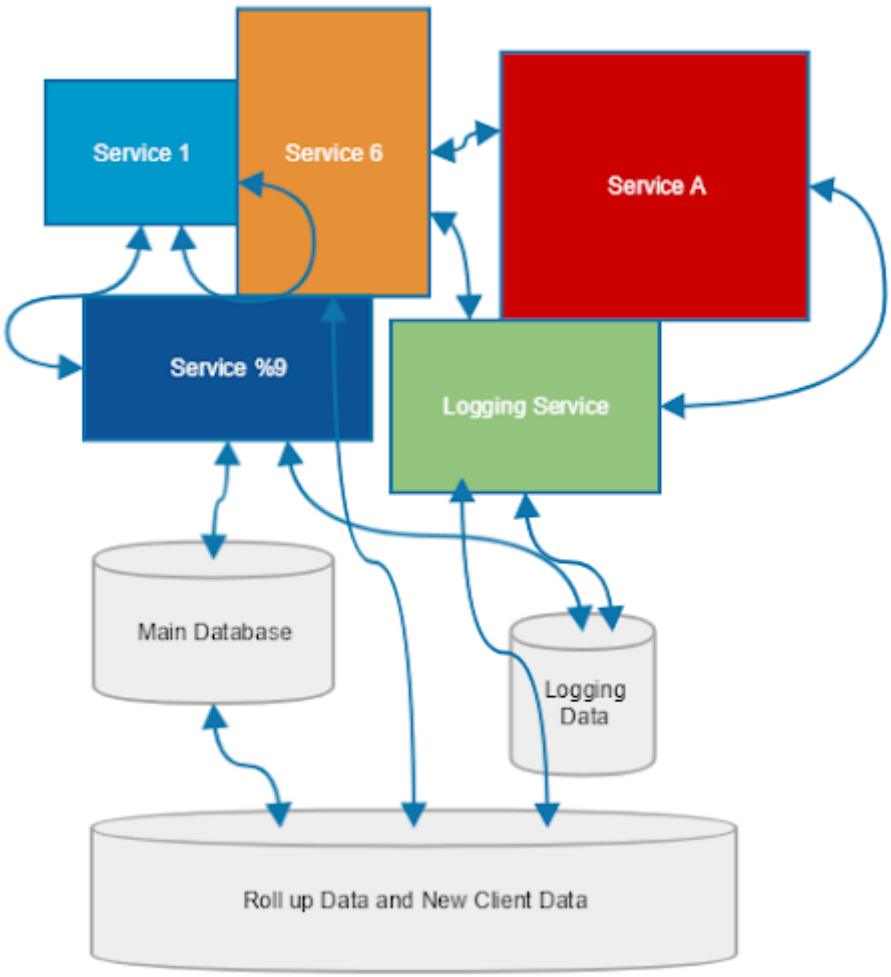
Quelques façons de mettre en œuvre peuvent répondre à ces préoccupations, par exemple, si l'application est assez jeune, un simple lift-and-shift devrait suffire. Pour les ensembles de données plus volumineux, une passerelle quelconque devant les appels de données peut être utilisée pour tirer parti d'un modèle lambda et utiliser le trafic normal dans le système pour remplir le nouveau magasin de données au fil du temps. Dans tous les cas, commencer par un domaine ou une section de données défini vous permettra de sortir progressivement vos données d'application du monolithe.

Nous sommes Agiles ! alias le Frankenstein

Ce dernier anti-modèle que nous examinerons se produit lorsque les équipes commencent à passer du développement logiciel en cascade au développement logiciel agile. Au début de ces changements de processus, les équipes finissent généralement par implémenter une version d'agile-fall. Souvent, cela est marqué par le dicton « Nous sommes agiles, nous n'avons donc plus à planifier les choses ! » Ceci est en réaction aux documents de conception lourds, aux plans de projet et aux calendriers de Gantt qui étaient standard dans la méthodologie en cascade.

L'idée fausse selon laquelle agile signifie que l'équipe n'a plus à planifier les choses et peut être agile et s'adapter aux besoins des clients (lire: caprices et choses brillantes) entraîne une décomposition des fonctionnalités dans le vide. Que veut le client/client/product owner ? – eh bien, c'est ce que nous allons construire ce sprint !

Avec suffisamment de cycles tout au long de ce processus, vous vous retrouvez avec plusieurs fonctionnalités disparates, souvent implémentées de la même manière ou de manière similaire à quelque chose de précédemment construit, qui doivent toutes être boulonnées ensemble pour partager des données et créer un semblant d'application cohérente. Cela crée finalement un monstre logiciel Frankenstein qui n'est qu'un tas de pièces cousues ensemble et qui s'aggrave avec le temps.



Cet anti-modèle devient auto-entretenu car au fil du temps, il devient plus complexe à déployer et à boulonner sur de nouvelles choses, surtout si elles doivent interagir avec des pièces existantes. Cela peut entraîner une dette technique qui semble gonfler et se manifeste souvent sous la forme de comportements indésirables mais cachés comme des transactions perdues, des instances orphelines et des ralentissements inexpliqués.

Finalement, Frankenstein commencera à s'effondrer et plus d'efforts seront déployés simplement pour essayer de garder les choses ensemble, que pour développer réellement de nouvelles fonctionnalités.

Une voie à suivre

Pour battre Frankenstein dans ce cas, vous devez prendre quelques cycles pour définir ce que fait votre application et le mapper aux implémentations concrètes de votre code. Vous trouverez très probablement des doublons ou des chemins de code qui ne sont jamais touchés. L'élimination de cette crasse est une bonne première étape.

Ensuite, pour aller de l'avant, concentrez-vous sur les contrats des interfaces de votre système. Chaque fois qu'un service appelle un autre service ou une ressource de données, définissez ce contrat dans quelque chose comme Swagger pour aider à faire la lumière sur ce que fait réellement le système.

Une fois ces deux activités terminées, l'étape suivante consiste à travailler avec votre chef de projet ou votre coach Agile pour vous aider à redéfinir la manière dont les fonctionnalités sont implémentées dans l'application. Tirer à partir de la hanche ne doit plus être envisagé. Initialement, un petit pic de conception au début de la fonctionnalité aidera à garantir que la fonctionnalité n'est pas dupliquée, utilise de bonnes pratiques de codage/des modèles de conception et exploite le code existant le cas échéant. Cette activité est généralement réalisée par un développeur senior ou un architecte technique, même si la mise en œuvre est réalisée par un autre membre de l'équipe. Cela permettra de garantir que tout le code est volontairement ajouté au système d'une manière qui étend les fonctionnalités sans encourir de dette technique supplémentaire.

La dernière partie pour s'éloigner du monstre est de commencer à ajouter du temps ou des sprints pour régler la dette technique qui a été accumulée. Cela peut être fait en pourcentage de chaque sprint, ou en sprints forfaitaires s'il y a des accalmies d'exigences. Quoi qu'il en soit, l'incorporation de la collecte des ordures et du nettoyage commencera à remettre le dentifrice dans le tube et aidera l'équipe et l'application à aller de l'avant.

Sommaire

Dans cet article, nous avons parlé des anti-modèles de microservices dont j'ai été témoin en travaillant avec des clients de toutes tailles. Ceux dont nous avons parlé ici étaient :

- Casser la tirelire
- Tout Micro (sauf pour les données)
- Nous sommes Agiles ! alias le Frankenstein

Après chacun, nous avons également essayé de donner un peu d'espoir et de montrer une voie à suivre pour aider à corriger les erreurs de chacun.

J'espère que vous avez apprécié cela au moins autant que le déploiement bleu/vert sans aucun problème de niveau 1 !

Merci,

Dallas Monson

Articles Similaires

(https://keyholesoftware.com/2016/03/07/developerweek-2016-retrospective/)

DEVELOPERWEEK 2016 RETROSPECTIVE (HTTPS://KEYHOLESOFTWARE.COM/2016/03/07/DEVELOPERWEEK-2016-RETROSPECTIVE/)

(https://keyholesoftware.com/2016/02/15/microservices-a-la-service-fabric/)

MICROSERVICES À LA SERVICE FABRIC (HTTPS://KEYHOLESOFTWARE.COM/2016/02/15/MICROSERVICES-A-LA-SERVICE-FABRIC/)

(https://keyholesoftware.com/2021/05/21/microservices-in-the-wild-three-types-how-to-implement/)

MICROSERVICES À L'ÉTAT SAUVAGE : TROIS TYPES ET COMMENT LES IMPLÉMENTER (HTTPS://KEYHOLESOFTWARE.COM/2021/05/21/MICROSERVICES-IN-THE-WILD-THREE-TYPES-HOW-TO-IMPLEMENT/)

ANTI-MODÈLES (HTTPS://KEYHOLESOFTWARE.COM/TAG/ANTI-PATTERNS/)

S MEILLEURES PRATIQUES (HTTPS://KEYHOLESOFTWARE.COM/TAG/BEST-PRACTICES/)

CONCEPTION AXÉE SUR LE DOMAINE (HTTPS://KEYHOLESOFTWARE.COM/TAG/DOMAIN-DRIVEN-DESIGN/)

DÉVELOPPEMENT D'ENTREPRISE (HTTPS://KEYHOLESOFTWARE.COM/TAG/ENTERPRISE-DEVELOPMENT/)

MICRO-DONNÉES (HTTPS://KEYHOLESOFTWARE.COM/TAG/MICRO-DATA/)

MICROSERVICES (HTTPS://KEYHOLESOFTWARE.COM/TAG/MICROSERVICES/)

APPLICATIONS MONOLITHIQUES (HTTPS://KEYHOLESOFTWARE.COM/TAG/MONOLITHIC-APPLICATIONS/)

0

Évaluation des articles

S'abonner

Connectez-vous avec Connexion (https://keyholesoftware.com/11205keysoft/?redirect_to=https%3A%2F%2Fkeyholesoftware.com%2F2019%2F03%2F18%2Fmicroservices-anti-patterns%2F)

Join the discussion

1 COMMENTAIRE Le plus ancien

Zach Gardner (https://keyholesoftware.com) il y a 2 ans

Excellent post, Dallas. This dives into definitely some of the pain points we ran into when transitioning from a monolith to a microservice paradigm. The other thing I'd recommend to think about when going down the microservice path is that it's a full stack shift. Rather than having all of your individual UI modules or screens in one big deployable unit, break them apart where each screen can be developed and deployed independent of the others. It's the same reason why we have a separate .sln file in the Microsoft world, or a separate .jar in the Java world, for… Read more »

(https://keyholesoftware.com/author/zgardner/)

0 Reply

Strategic Partners

We have various partnerships to best benefit our clients including:

(https://keyholesoftware.com/company/about/microsoft-competency-partner/)

(https://keyholesoftware.com/company/about/aws-consulting-partner/)

Pick a Topic

Select Category

Ressources

- Papiers blanc (https://keyholesoftware.com/company/creations/white-papers/)
- Vidéos et présentations (https://keyholesoftware.com/company/creations/presentations/)
- Créations de trou de serrure (https://keyholesoftware.com/company/creations/)
- Événements éducatifs (https://keyholesoftware.com/company/creations/presentations/events/)
- Blogue technique (https://keyholesoftware.com/blog/)
- Tutoriels (https://keyholesoftware.com/company/creations/tutorials/)

Messages récents

Code hérité : utilisez les meilleures pratiques comme SOLID lors de la conversion (https://keyholesoftware.com/2021/10/26/legacy-code-tribulations/)

Utilisation de la grille CSS pour empêcher la page Jank (https://keyholesoftware.com/2021/10/18/using-css-grid-to-prevent-layout-shift/)

Utiliser Node.js pour créer un bot Discord (https://keyholesoftware.com/2021/10/13/using-node-js-to-create-a-discord-bot/)

Utilisation de JAXB et StaxEventItemReader pour lire des données XML (https://keyholesoftware.com/2021/10/05/using-jaxb-and-staxeventitemreader-to-read-xml-data/)

[Vidéo] IaC basé sur Terraform : principes de base avec le code (https://keyholesoftware.com/company/creations/presentations/terraform-based-infrastructure-as-code/)

Abonnez-vous au blog

Nom

Adresse e-mail

Adhérer maintenant!

(https://www.expertise.com/ks/overland-park/software-development)


(https://facebook.com/keyholesoftware)

(https://twitter.com/keyholesoftware)

(https://www.linkedin.com/company/keyhole-software)

(https://www.youtube.com/channel/UCAIUNkXmnAPgLWnqUDpUGAQ)

(https://keyholesoftware.com/feed)

Impossible d'établir une connexion avec le service reCAPTCHA. Veuillez vérifier votre connexion Internet, puis actualiser la page pour afficher une image reCAPTCHA.