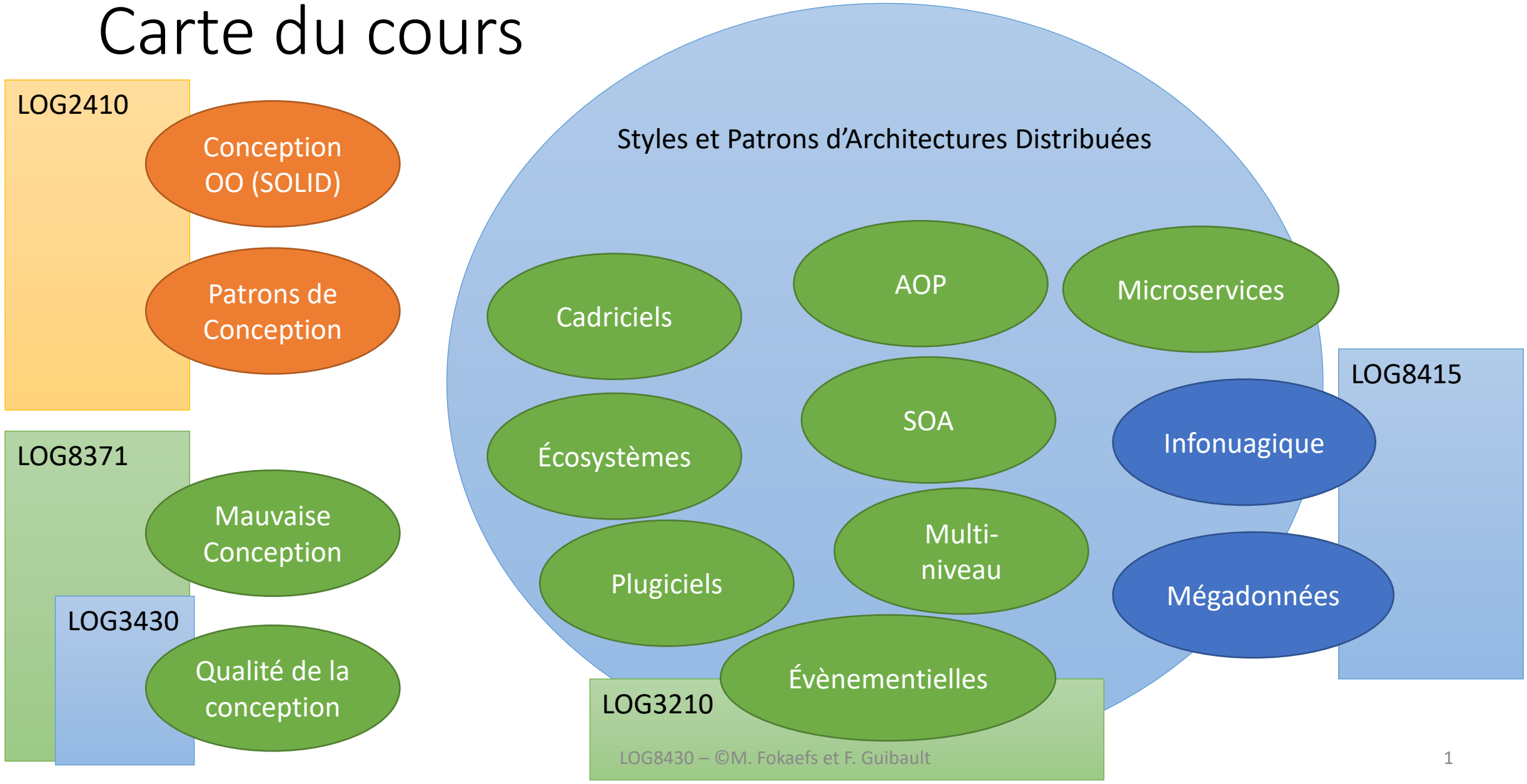
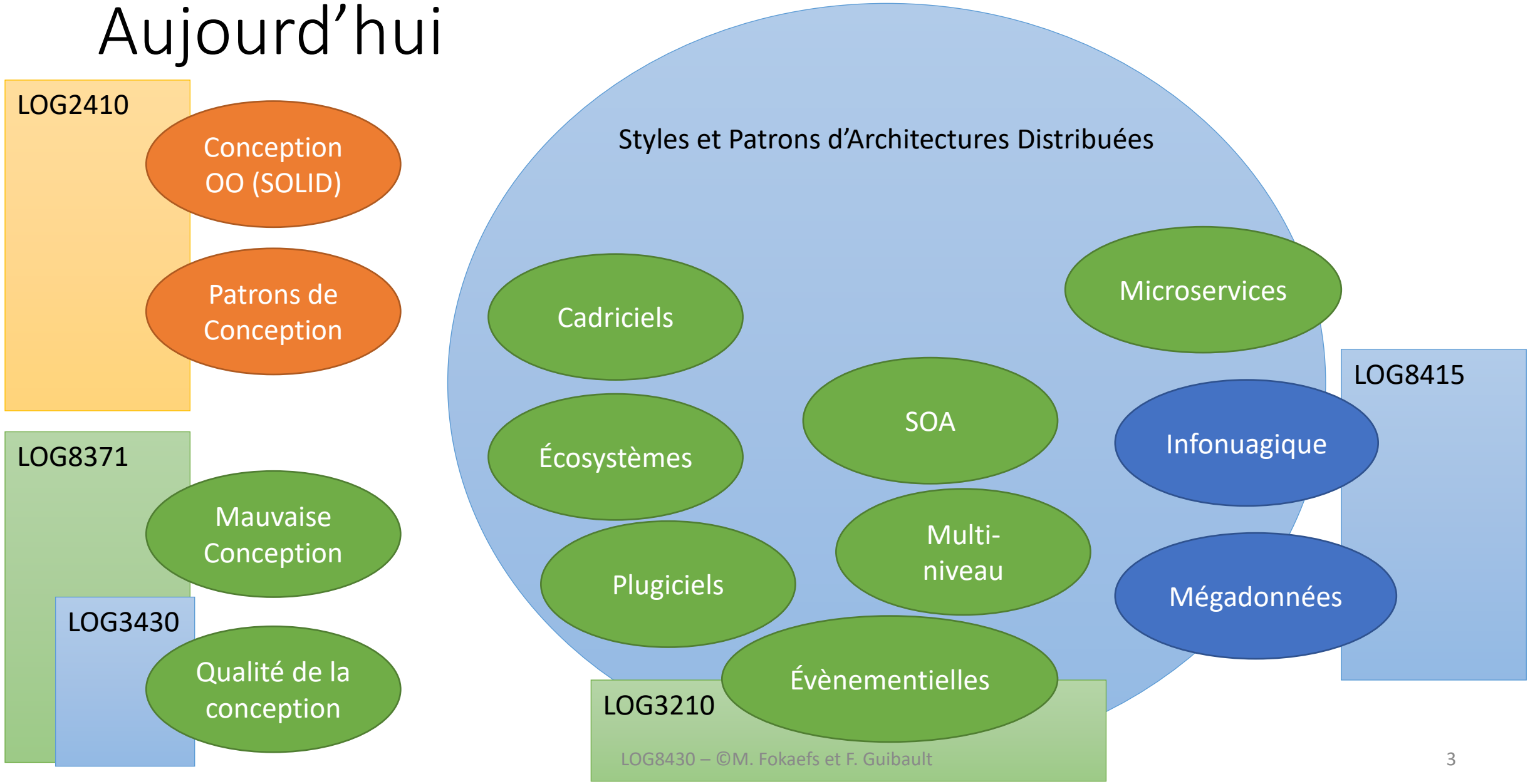
Carte du cours



LOG8430E: Microservice Architectures

Aujourd'hui


LOG2410
Conception OO (SOLID)
Patrons de Conception
LOG8371
Mauvaise Conception
LOG3430
Qualité de la conception
Styles et Patrons d'Architectures Distribuées
Cadriciels
Écosystèmes
Plugiciels
SOA
Multi-niveau
Évènementielles
Microservices
Infonuagique
Mégadonnées
LOG8415
LOG3210

Qu’est-ce que c’est les microservices?

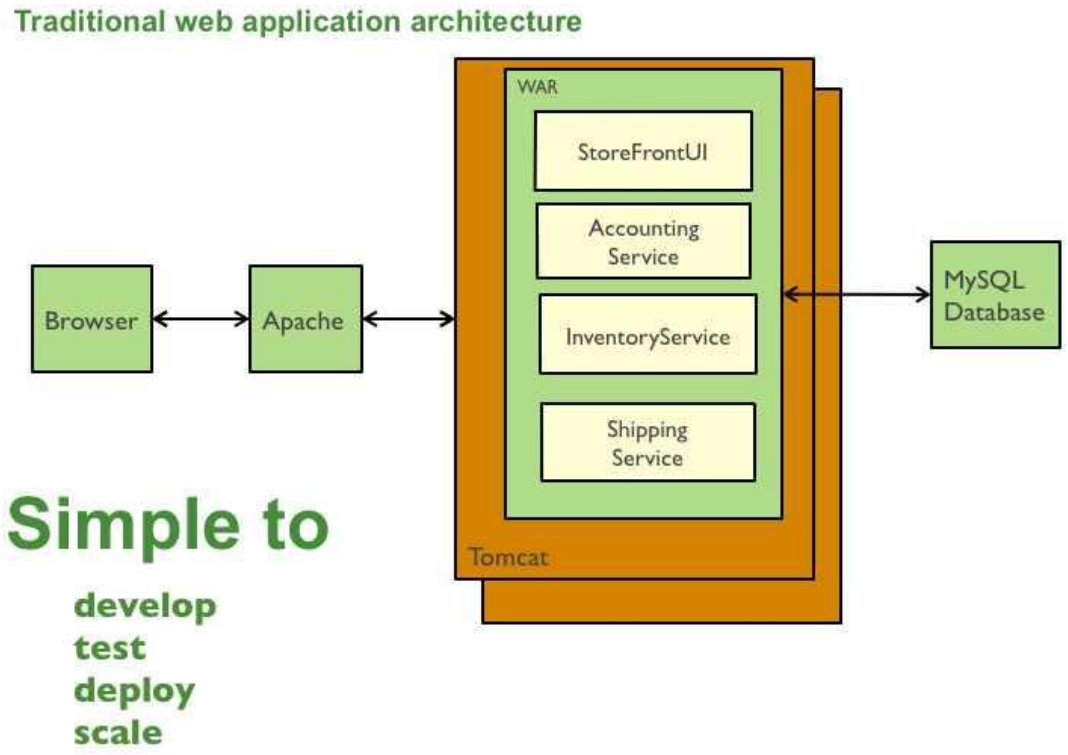
- C’est une version…plus petite que la SOA.
- Chaque service est juste un processus avec une seule responsabilité (selon les principes Unix).
  - Ou selon le principe de Amazon « Two Pizza Team » (c.-à-d. que toute l’équipe peut être nourrie par deux pizza), environ une douzaine des personnes.
  - Donc, un service que ne peut être développé et géré que par une douzaine des personnes.
  - Les microservices maintiennent toutes les propriétés de SOA : la réutilisabilité, la portabilité et l’interopérabilité.
  - En combinaison avec les containeurs, les microservices sont la technologie fondamentale pour les DevOps.
  - Ils promeuvent l’automatization, l’évolution et la livraison continues.

Du monolithe à microservices.

- Imaginez une application web typique en trois couche, avec un client, un serveur et une bases de données.
- L'application serveur est conçue pour :
  - gérer les requêtes HTTP et d'autres messages.
  - des divers clients (navigateurs, mobile, d'autres applications web)
  - et exécuter la logique d'affaires (l'expertise) qui correspond à la requête appropriée
  - interagir avec la base des données et remplir les vues HTML sur le navigateur.
- L'application serveur est parfois conçue comme un *monolithe* :
  - Toute la logique est exécutée comme un seul processus.
  - Elle permet de bénéficier des caractéristiques du langage de l'implémentation (p.ex. classes, méthodes etc.)
  - On peut la tester dans une seule machine, automatiser son déploiement et la mettre en échelle en façon horizontales, afin que plusieurs serveurs peuvent exécuter la même application et servir plusieurs requêtes.

Une application web monolithe

- Navigateur = Application cliente
- Apache = Équilibrer de charge (pour permettre la mise en échelle)
- Tomcat = Serveur d'application
- WAR = Application serveur
- MySQL = Base des données



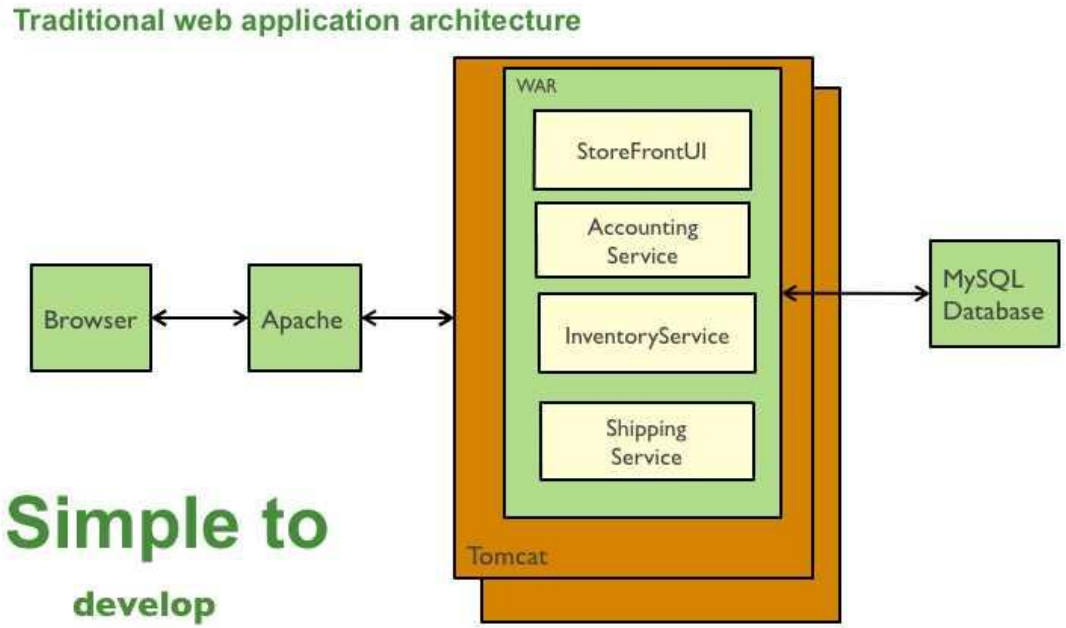
Avantages du monolithe

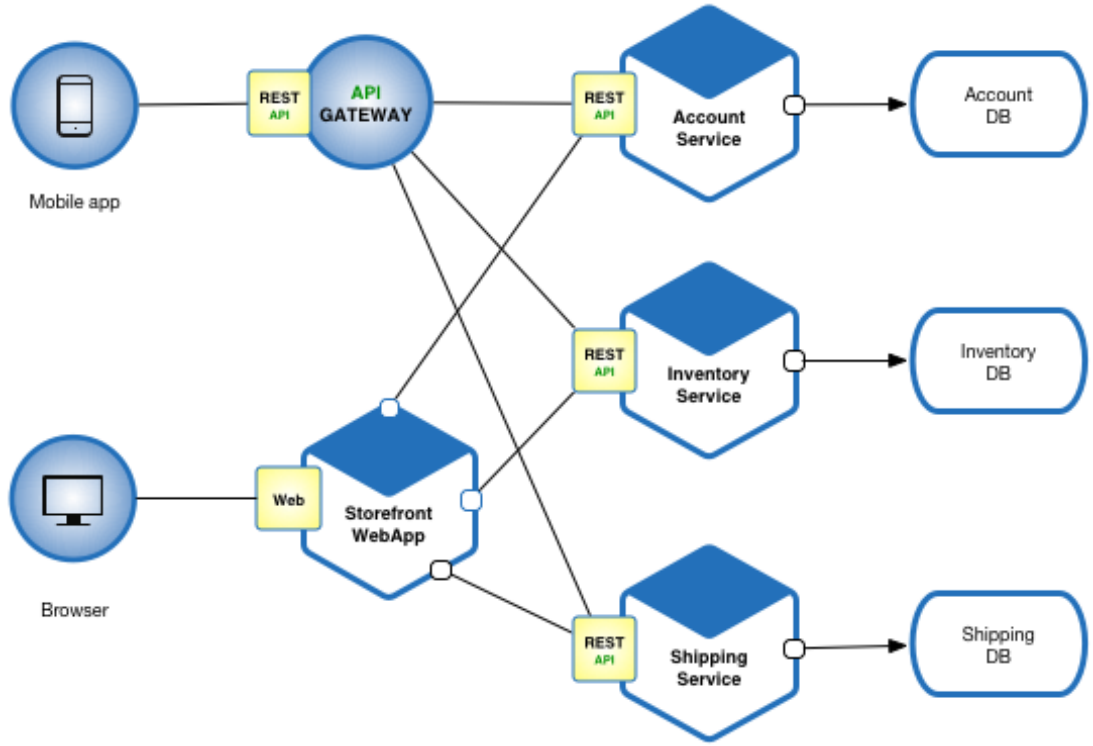
- Une seule équipe des développeurs
- Facile à développer
  - La plupart de IDE et d'outils de développement soutiennent le modèle monolith couramment.
- Compréhensibilité et modifiabilité accrues
  - C'est relativement facile à « onboard » des nouveaux développeurs.
- Facile à tester, permettre le déploiement continu.
  - Paqueté comme un seul module pour faciliter le déploiement (p.ex. des fichiers WAR en Java)
- Possible de mettre en échelle en façon horizontale.
- Se bénéficier des cadriciels et des caractéristiques de langage de programmation existant.

Désavantages du monolithe

- Au fur et à mesure que le monolithe commence à grossir, des problèmes apparaissent.
- La compréhensibilité est réduite et l'intégration devient plus difficile.
- Briser la modularité est plus facile, bien que ce soit encore involontaire, car il n'y a pas de frontières strictes entre les modules.
- Plus la base de code est grande, plus votre IDE et votre conteneur d'applications Web (par exemple, Tomcat ou Flask) sont surchargés.
- La maintenabilité et l'évolutivité peuvent être difficiles.
  - Lorsqu'un composant change, l'ensemble de l'application doit être reconstruit et redéployé.
  - Si un composant doit être mis à l'échelle, l'ensemble de l'application sera mis à l'échelle (de nouvelles ressources seront ajoutées).
- Nécessiter un engagement à long terme envers une pile technologique, ce qui peut entraîner un verrouillage du fournisseur (vendor lock-in).
  - Lorsque notre application est en Java, nous devons utiliser des cadriciels Java et des conteneurs ou serveurs Java, comme Tomcat.

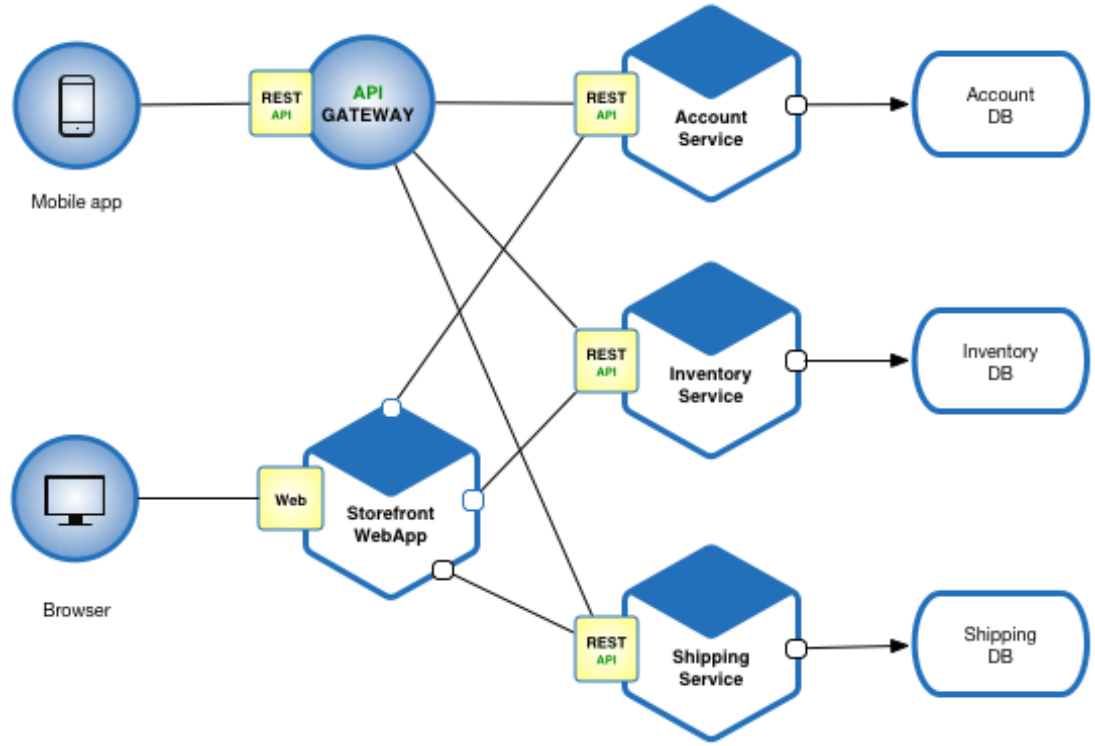
Du monolithe à microservices.





Une application web microservice

- Un module pour chaque composant logique de l'application serveur.
- Utilisation de l'équilibreur de charge / de la passerelle / de l'Appstore pour découpler les demandes.
- Base de données dédiée par module.



Du monolithe à microservices.

Avantages

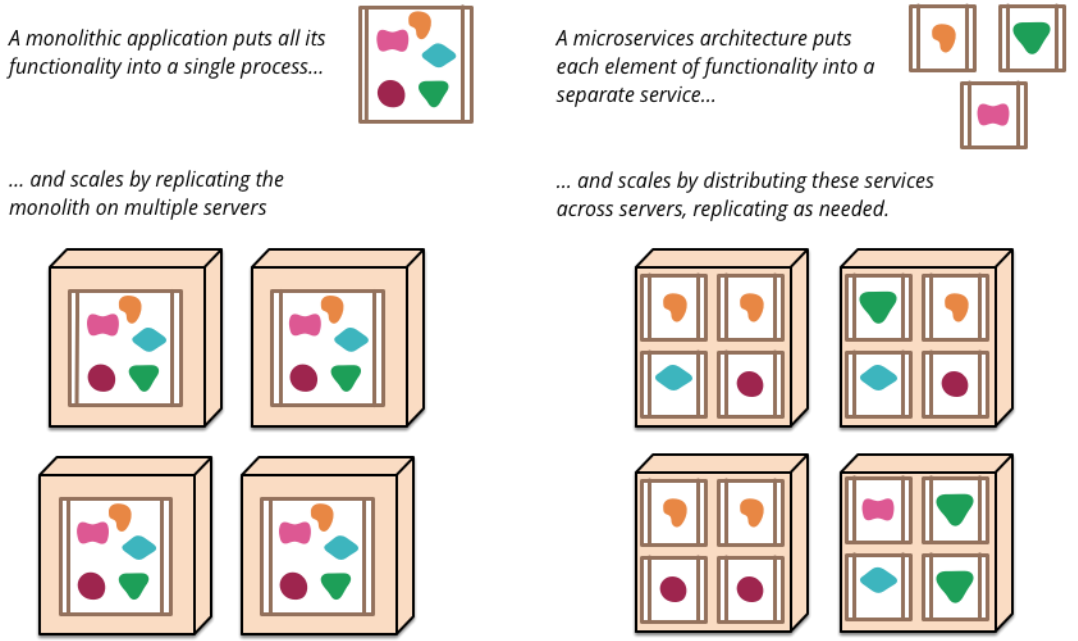
- Amélioration de la maintenabilité et de la testabilité.
  - Des modules plus petits, des équipes plus petites.
  - Des équipes dédiées et expertes.
- Amélioration de la productivité.
  - Allégez le fardeau des IDE
  - Meilleure compréhensibilité, intégration plus facile.
- Isolement d'anomalie.
  - Une panne dans un service n'affectera pas les autres services.
- Éliminer l'engagement à long terme envers les technologies.

Désavantages

- Complexité supplémentaire
  - Développement
  - Testing
  - Déploiement
- Augmentation de la consommation de mémoire.
  - Chaque service doit réserver une partie de la mémoire.

Scaling: Monolithe vs Microservices

- Au lieu de mettre à l'échelle l'ensemble de l'application, nous ne répliquons que les composants surchargés.
- Cela facilite également la maintenabilité et le déploiement continu.
  - Il suffit de redéployer les modules concernés par un changement.
  - Nous pouvons conserver simultanément différentes versions des modules concernés, tout en passant à la nouvelle version (déploiement Canary).



Caractéristiques d’une Architecture Microservice

- Division en composantes via les services
- Décomposition des affaires et du développement
- Produits et non projets
- Points de terminaison intelligents et tuyaux stupides
- Gouvernance décentralisée
- Gestion décentralisée des données
- Automatisation de l'infrastructure
- Concevoir pour l'échec
- Conception évolutive

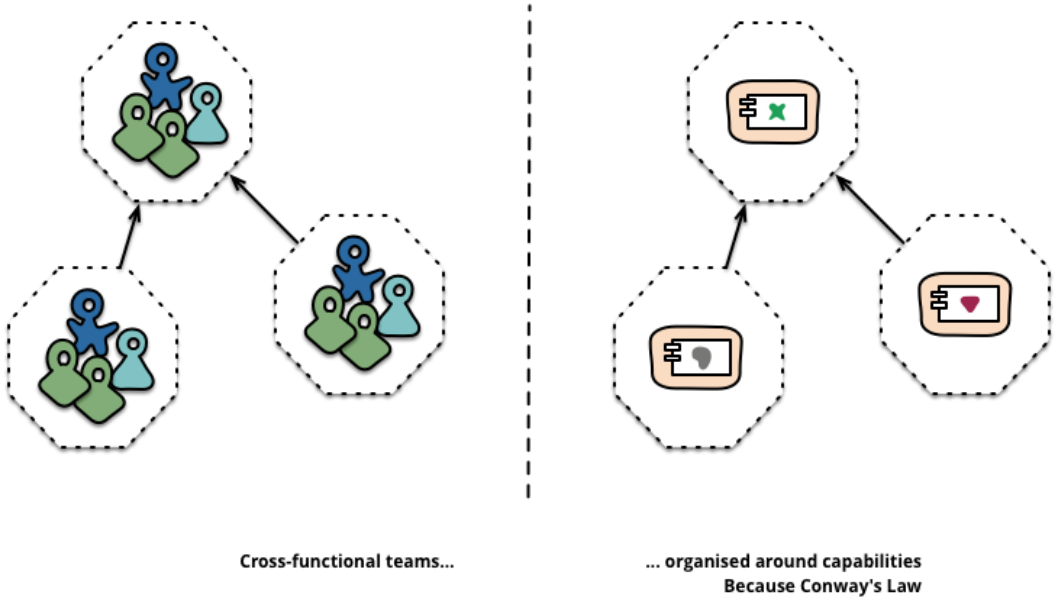
Division en composantes via les services

- Le premier défi de la transition d'une architecture monolithe à une architecture modulaire est de savoir comment diviser l'architecture en composants.
  - Composant: Une unité logicielle remplaçable indépendamment et évolutive.
- Nous pouvons faire des composants de deux manières:
  - Bibliothèques: composants liés à un programme via des appels de fonction en mémoire.
  - Services: composants hors processus avec capacités de communication à distance (demandes de service ou appels de procédure à distance (RPC)).
- Avec les bibliothèques, vous gagnez:
  - Frontières de modularité plus douces.
  - Appels en mémoire plus rapides et moins chers.
- Avec les services, vous gagnez:
  - Déploiement indépendant
  - Interfaces publiques et explicites

Décomposition des affaires et du développement

Toute organisation qui conçoit un système (défini au sens large) produira une conception dont la structure est une copie de la structure de communication de l'organisation.
-- Melvin Conway, 1968

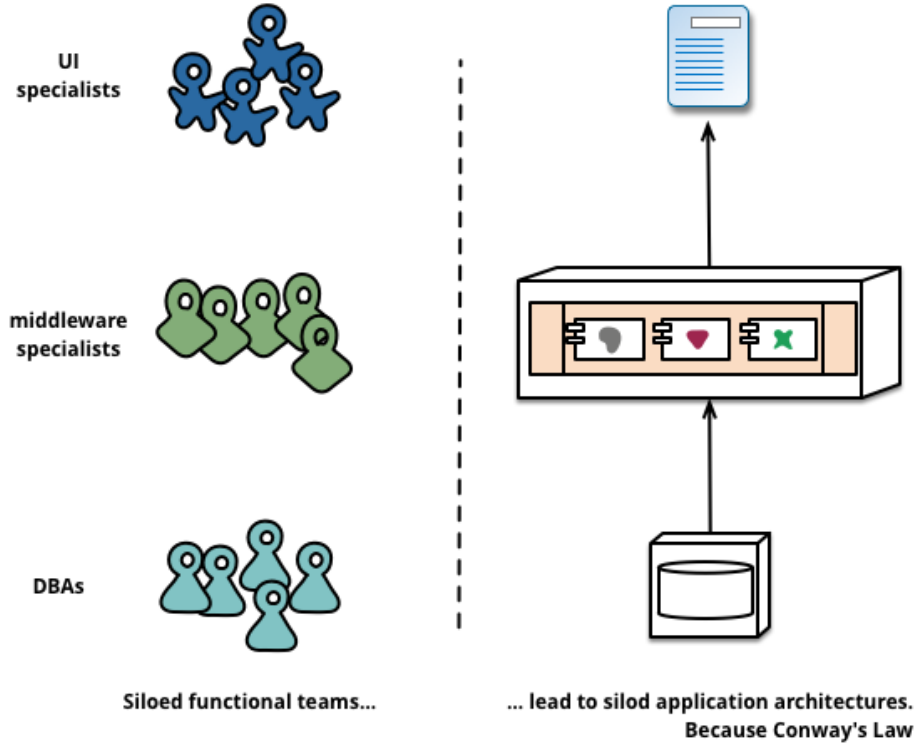
Business oriented



Cross-functional teams...

... organised around capabilities
Because Conway's Law

Development oriented

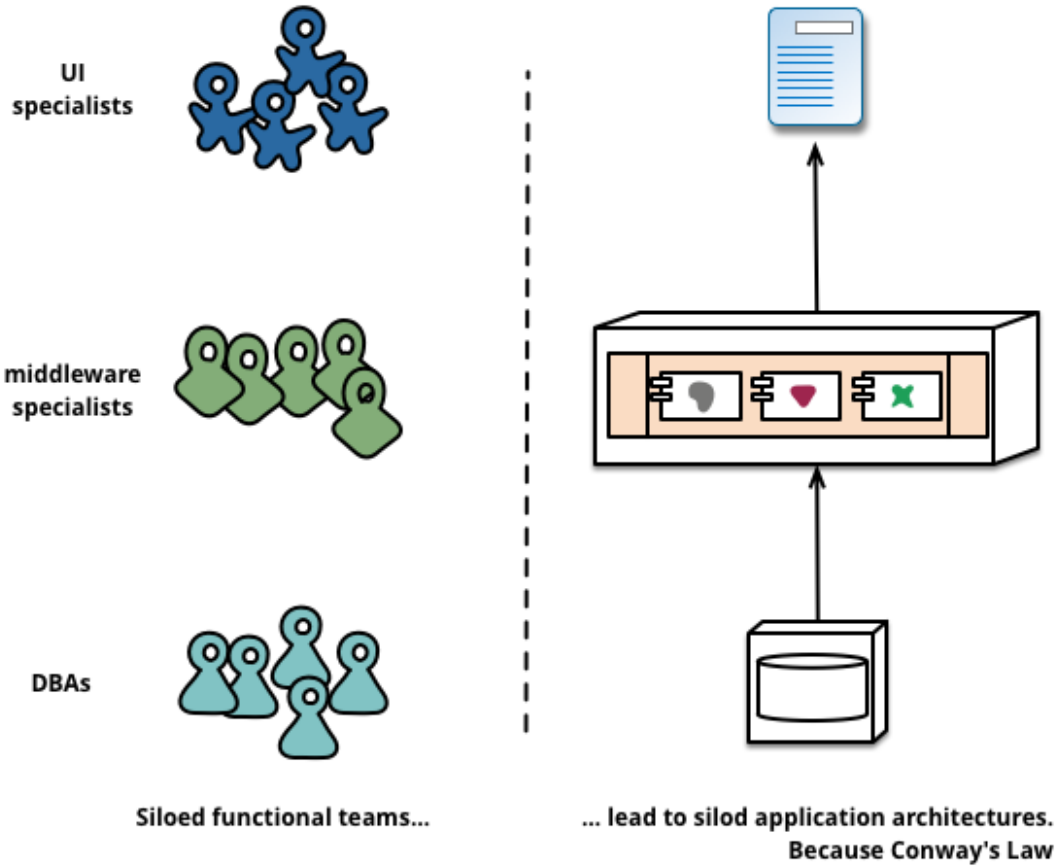


Siloed functional teams...

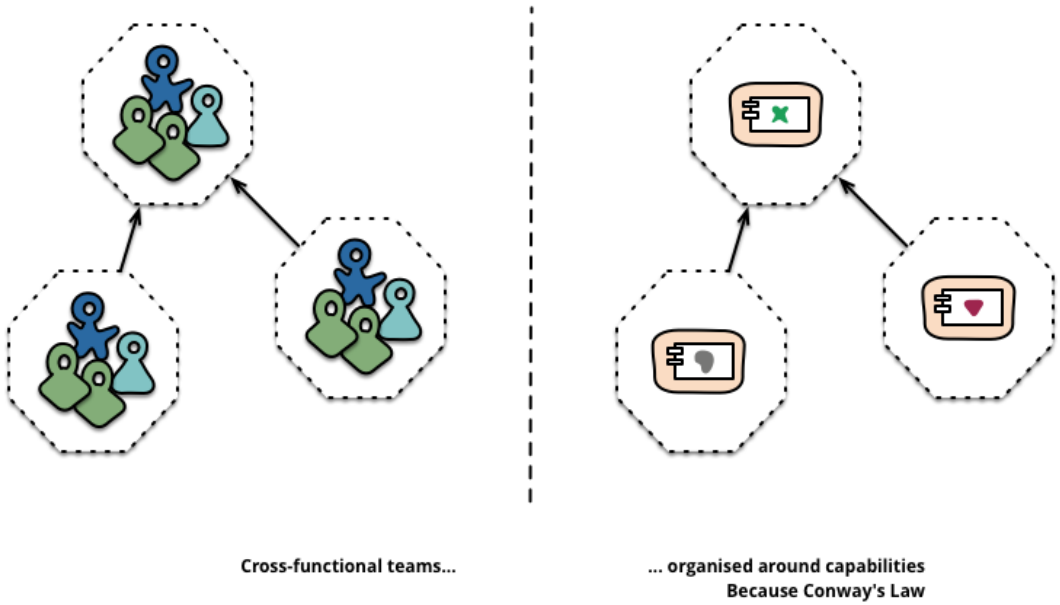
... lead to silod application architectures.
Because Conway's Law

Décomposition du développement

- Nous supposons que nous avons des experts pour les technologies et les couches de l'application.
  - Experts en GUI
  - Experts BD
  - Experts du service
- Étant donné que chaque équipe devra travailler avec TOUS les composants le moment venu, l'architecture est plus monolithique.
- Peut entraîner de fréquentes réunions entre les équipes, des conflits et des violations de budget et de délai.
- Peut conduire à une injection de logique dans toutes les couches.



Décomposition des affaires



- Equipes organisées par module.
- Équipes interfonctionnelles pour inclure tous les experts.
- L'architecture reflète l'organisation de l'équipe de développement.
- Meilleure maintenabilité et productivité.

Produits et non projets

- Le développement orienté projet s’occupe de fournir un ensemble de fonctionnalités.
  - Le projet est ensuite repris par une équipe de maintenance et d'exploitation.
- Dans le développement orienté produit, les développeurs sont également responsables de l'exécution du système, le logiciel en production.
  - Le principe d'Amazon «vous le construisez, vous l'exécutez».
- La mentalité produit est mieux liée aux capacités de l'entreprise.
  - L'utilisateur / client devient une préoccupation pour les développeurs.
  - La qualité d'exécution devient une préoccupation impérative.
  - Comment un logiciel peut-il produire de la valeur?
- La mentalité produit peut être appliquée aux monolithes, mais des composants de plus petite taille et plus ciblés pour des objectifs commerciaux spécifiques sont mieux adaptés.

Points de terminaison intelligents et tuyaux stupides

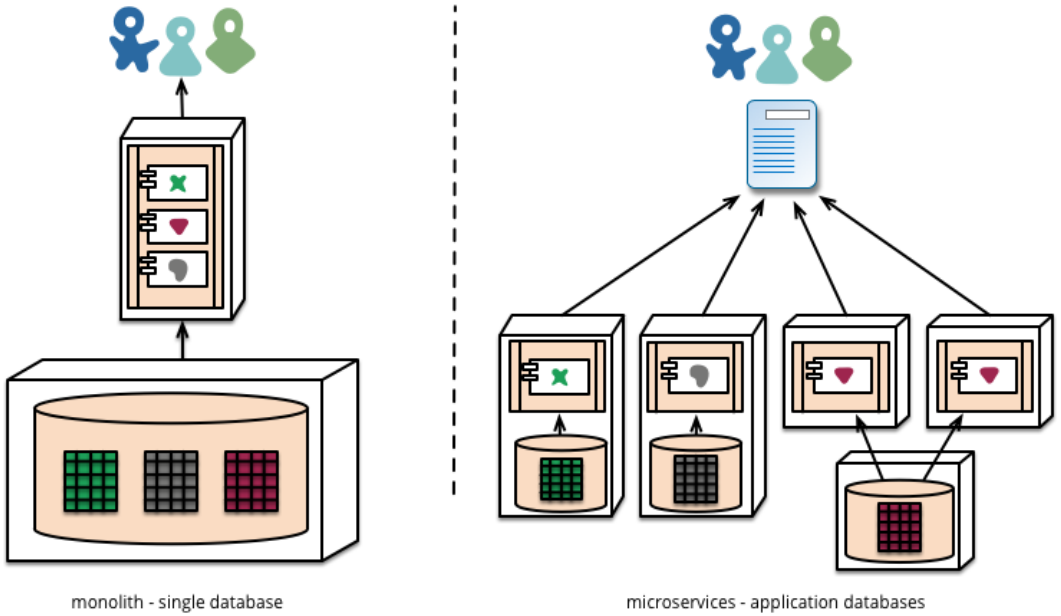
- La SOA traditionnelle utilise des moyens «lourds» pour permettre la communication entre les services.
  - Protocole SOAP
  - Bus de service d'entreprise (Enterprise Service Bus)
- Ces moyens comprennent une logique importante pour permettre le formatage des messages, le rassemblement et le démarshalling, le routage et la chorégraphie.
- Les microservices privilégient des moyens de communication beaucoup plus simples tels que le simple HTTP ou un middleware orienté message avec le moins de logique.
- Dans ce cas, la logique sur la façon de comment gérer le message réside sur le service récepteur et la logique sur la façon de comment préparer un message réside sur le service expéditeur.
- Les services deviennent un peu plus complexes, mais grâce à la modularité et à la spécialisation de chaque service, nous gagnons plus de flexibilité sur les types de messages que nous pouvons traiter.

Gouvernance décentralisée

- Le concept se réfère davantage à la pratique selon laquelle chaque service est responsable de lui-même.
- Premièrement, cela concerne l'utilisation de la technologie.
  - Langage de programmation, bases de données, cadriciels etc.
  - Chaque service peut utiliser sa propre pile technologique et éviter le blocage du fournisseur pour le système.
- Deuxièmement, cela concerne le principe «vous le construisez, vous l'exécutez».
  - Lorsqu'un défaut survient au moment de l'exécution, l'alarme ne se déclenche pas pour l'ensemble du système et l'ensemble de l'équipe de développement ou de l'équipe de gestion.
  - Chaque équipe est responsable du fonctionnement de son propre service.

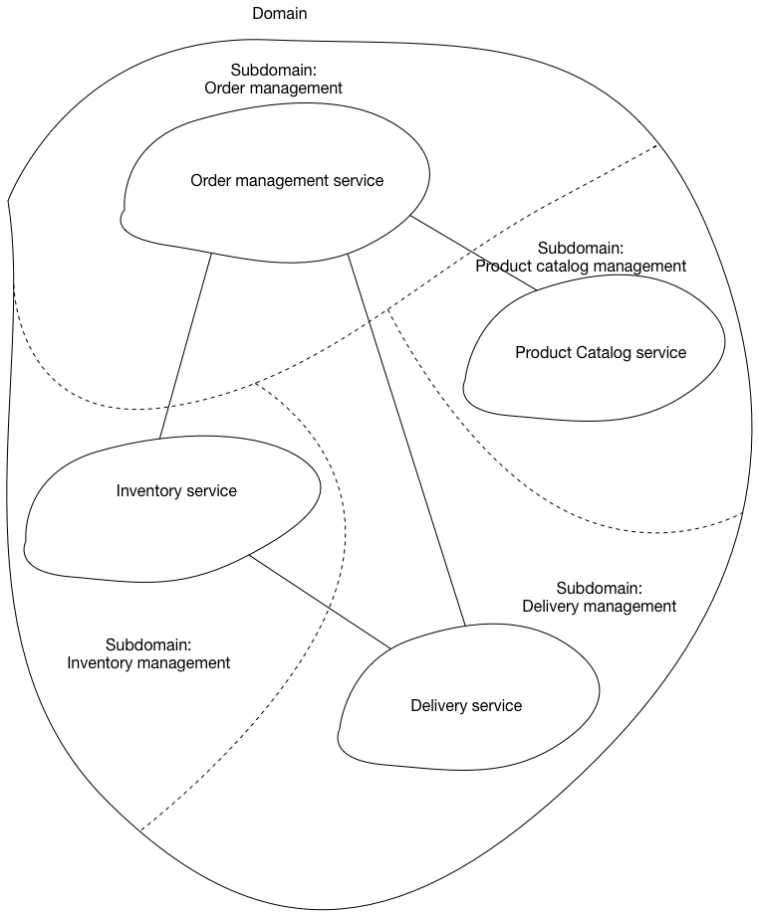
Gestion décentralisée des données

- Une entité peut avoir différentes vues (c'est-à-dire des attributs) au sein d'une grande application ou entre des applications.
  - Client pour les ventes, client pour le support.
- Selon le Domain-Driven Design (DDD), nous pouvons diviser un domaine complexe en plusieurs domaines *délimités* (domaines avec des frontières claires mais transparentes).



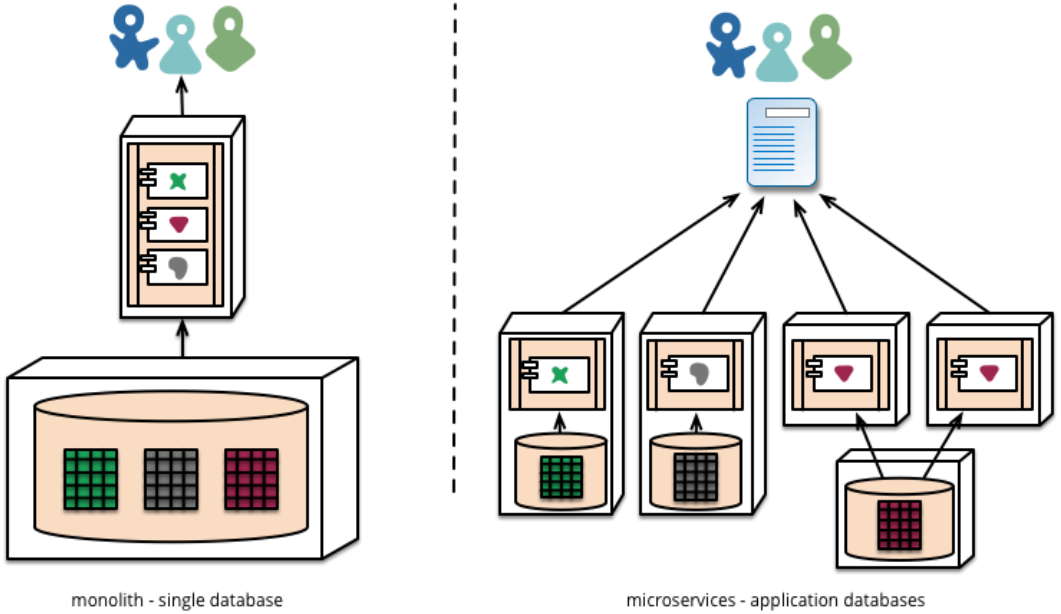
Gestion décentralisée des données

- Une entité peut avoir différentes vues (c'est-à-dire des attributs) au sein d'une grande application ou entre des applications.
  - Client pour les ventes, client pour le support.
- Selon Domain-Driven Design (DDD), nous pouvons diviser un domaine complexe en plusieurs domaines délimités (domaines avec des frontières claires mais transparentes).
- Ceci est à égalité avec la décomposition orientée business et la gouvernance décentralisée.



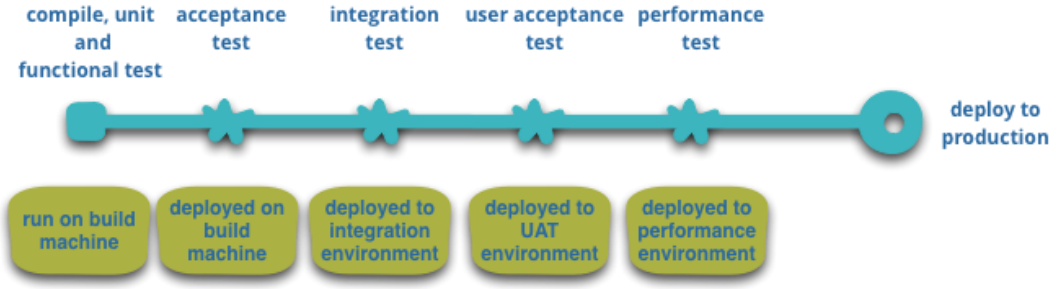
Gestion décentralisée des données

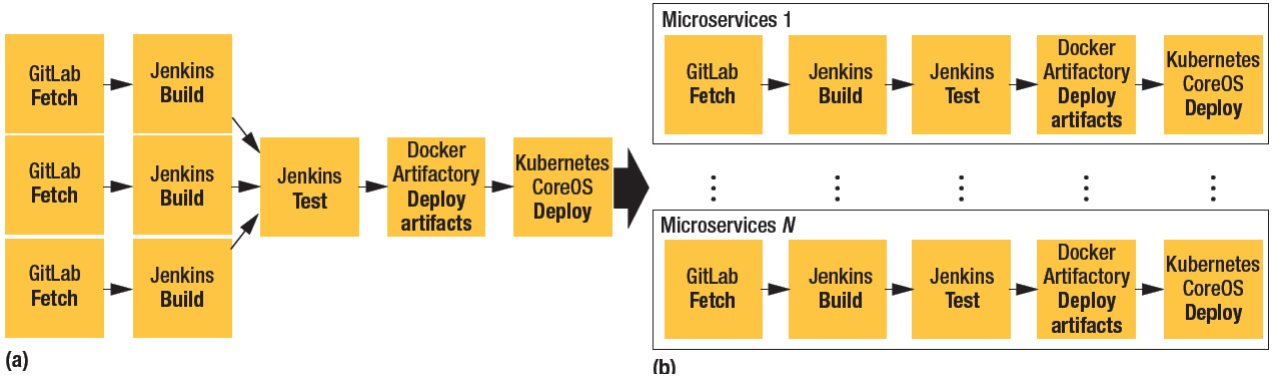
- Une entité peut avoir différentes vues (c'est-à-dire des attributs) au sein d'une grande application ou entre des applications.
  - Client pour les ventes, client pour le support.
- Selon Domain-Driven Design (DDD), nous pouvons diviser un domaine complexe en plusieurs domaines délimités (domaines avec des frontières claires mais transparentes).
- Ceci est à égalité avec la décomposition orientée business et la gouvernance décentralisée.
- Que se passe-t-il si deux services partagent une base de données ou modifient la même entité logique dans deux bases de données différentes?
  - Rappelez-vous le théorème CAP!
  - Disponibilité de base, état souple, cohérence éventuelle (BASE)
  - «Le compromis en vaut la peine tant que le coût de la correction des erreurs est inférieur au coût des affaires perdues avec une plus grande cohérence.»



Automatisation de l'infrastructure

- Investissez beaucoup dans l'automatisation!
- Concentrez-vous sur les tâches répétitives à chaque changement.
  - Testing
  - Déploiement
- Rendez-les ennuyeux!
  - Approche "en un clic" (voir GitLab et GitHub Actions)
- Ces tâches peuvent être automatisées même dans le monolithe.
  - S'ils sont ennuyeux pour une application, ils peuvent être ennuyeux pour plus.



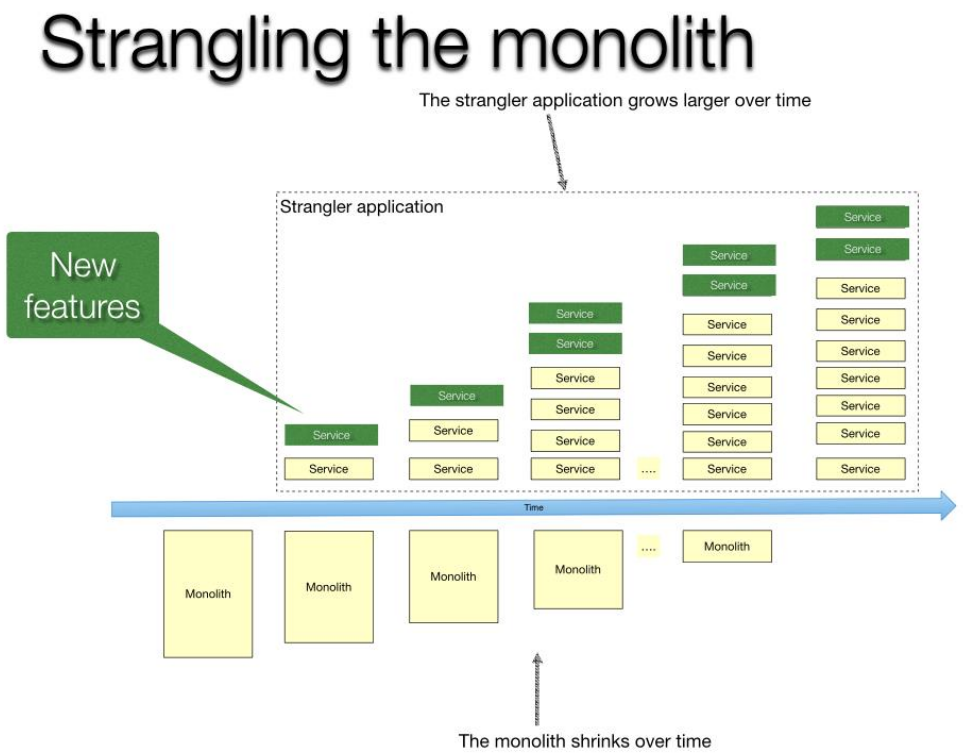


Concevoir pour l'échec

- Tout peut échouer à tout moment!
  - Par exemple, indisponibilité des services.
- Si vous échouez, échouez gracieusement.
  - Minimisez l'impact sur le client / utilisateur.
- Le monitorage est la clé!
  - Au sein du service et entre les services.
  - Être capable de détecter les mauvais comportements émergents à cause de la synergie des services.
  - Monitorage technique (par exemple, throughput des demandes) et monitorage commercial (par exemple, throughput des commandes).
- Ingénierie du chaos (voir LOG8371)
  - La pratique de l'injection aléatoire, inattendue et intentionnelle de pannes dans les services, les nœuds ou même les centres de données.
  - Pour voir comment l'échec est résolu, peut être évité et comment l'équipe réagit.
  - Aide à être prêt pour de véritables échecs.

Conception évolutive

- Lors de la décomposition de notre monolithe, nous devons avoir une règle à l'esprit:
- « Est-il possible de réécrire ou de remplacer complètement un composant sans affecter ses dépendants ou le système dans son ensemble? »
- Dans ce principe, développez votre système en tant que cadriciel.
  - Quels sont les services de base?
  - Quels sont les services qui changent souvent?
  - Y a-t-il des services temporaires qui changeront sous certaines conditions?
- Cette logique s'applique également lorsque nous migrons un monolithe vers un composant de microservice.
  - Le monolithe reste comme un système central qui expose une API.
  - De nouvelles fonctionnalités sont ajoutées en tant que nouveaux services.
  - Nous remplaçons progressivement les fonctionnalités de base par d'autres microservices.
  - Ceci est connu sous le nom de patron «Strangler Application».
- Rappelez-vous l'antipatron «Shotgun Surgery».
  - Si vous modifiez à plusieurs reprises deux services ensemble, fusionnez-les probablement.



Antipatrons d’adoption des architectures microservice

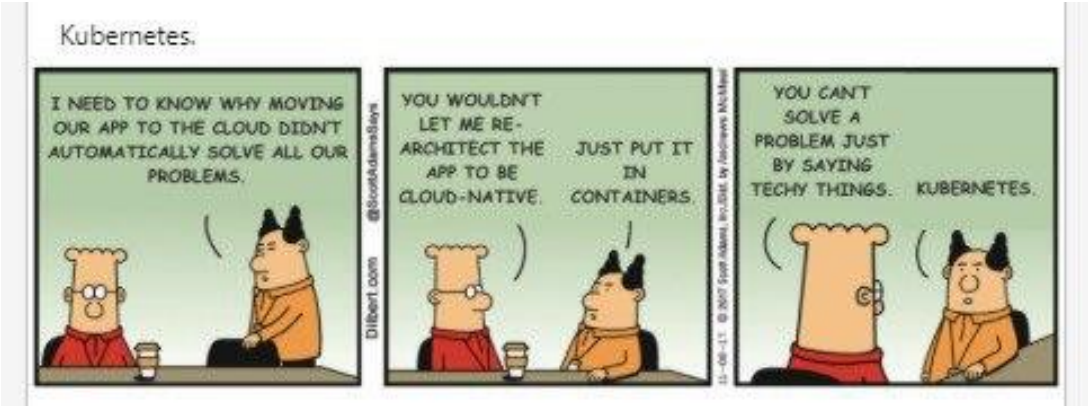
- Magic pixie dust
- Microservices as the goal
- Scattershot adoption
- Trying to fly before you can walk
- Focussing on technology
- More the merrier
- Red flag law

Magic pixie dust

- «Une pincée de microservices résoudra tous nos problèmes de développement.»
- L'adoption d'une architecture doit s'accompagner de l'adoption des principes, patrons, philosophies et cultures appropriés.
- Utilisez-vous déjà les pratiques appropriées pour les architectures de microservices?
  - Testing et déploiement automatisés?
  - Des équipes de développement transversales?
  - Développement et opérations intégrés (DevOps)?
- Avant d'adopter une architecture de microservice, pensez:
  - Quelle est la cause première de votre problème? Développement? Déploiement? Organisation?
  - L'architecture des microservices résout-elle vos problèmes?
  - Avez-vous les conditions préalables en place?

Microservices as the goal

- Un responsable intervient et annonce une initiative visant à transformer tous les systèmes en microservices. «Faites des microservices!»
  - Les développeurs sont-ils préparés et capables de ce changement?
  - Sont-ils bien organisés?
  - Le code est-il préparé (propre et refactoré) pour la transformation?
- Avant d'adopter l'architecture, essayez de réparer et de préparer le monolithe lui-même
  - Améliorez les délais (délai entre le commit et le déploiement).
  - Améliorez la fréquence de déploiement.
  - Réduisez le taux d'échec.
  - Réduisez le temps de récupération.
- Adoptez les tests et le déploiement automatisés, éliminez le travail inutile, améliorez le monitorage.



Scattershot adoption

- C'est une conséquence de l'antipatron précédent: la direction a annoncé l'adoption de microservices, mais elle n'a jamais spécifié de plan ou de stratégie sur la façon de le faire.
- En conséquence, chaque équipe de l'organisation a adopté sa propre stratégie et il y a un manque général de coordination.
- Après avoir décidé d'adopter des microservices:
  - Définir et communiquer la stratégie d'adoption.
  - Établir des indicateurs de performance clés (KPI) pour évaluer les résultats et les progrès vers l'objectif.
  - Mettre en place une équipe d'infrastructure chargée de créer l'infrastructure de développement et d'exécution.
  - Appliquer la stratégie sur une application monolithique candidate.
  - Développez le reste des applications.

Trying to fly before you can walk

- Essentiellement, adoptez une architecture avancée, comme des microservices, avant d'adopter d'abord les bonnes pratiques de développement logiciel et les principes de conception logicielle.
  - Code propre, bonne conception, tests automatisés, assurance qualité des logiciels.
- Enforcer une assurance automatisée de la qualité du code (par exemple, des révisions de code) dans le cadre du pipeline de déploiement.
- Adoptez un processus de développement axé sur les tests pour enforcer des tests automatisés dans le cadre du pipeline d'intégration continue.
- Assurez-vous d'utiliser des principes de conception (SOLID, patrons de conception) ou une conception basée sur le domaine (DDD) pour faciliter la décomposition du monolithe.
- Refactorisez le monolithe avant de migrer.

Focusing on Technology

- Ce n'est pas parce que vous adoptez une technologie spéciale que la migration (ou l'architecture de microservice elle-même) fonctionnera.
  - Docker ou Kubernetes ne transformeront pas automatiquement votre système en microservices.
- L’emphase sur la technologie peut rapidement conduire à un vendor lock-in.
- Avant de choisir une pile technologique:
  - Mettre l’emphase sur la conception (décomposition du monolithe en services)
  - Mettre l’emphase sur l'organisation (une équipe par service)
  - Mettre l’emphase sur les objectifs et les capacités de l'entreprise.
- Lorsque tous ces éléments sont pris en compte, la pile technologique que vous choisirez ne devrait pas faire de différence.
- Votre système devrait fonctionner avec n'importe lequel d'entre eux.

More the merrier

- Microservice n'implique pas littéralement petit.
  - Prendre cela à la lettre peut conduire à un anti-modèle appelé *nanoservice*.
- Des services plus petits impliquent beaucoup plus de services.
  - Plus de dépendances, communication complexe entre les services.
  - Augmentation de la complexité du développement (maintenance, tests, évolution).
  - Communication accrue (peut-être plus que dans le cas du monolithe).
- Définissez un service par équipe.
- N'ajoutez pas de nouveau service à moins qu'il ne résout un problème.
- Divisez les services qui deviennent trop volumineux
  - Taille, complexité, maintenabilité, testabilité
- Divisez les équipes (et leurs services) qui deviennent trop grandes.
  - N'oubliez pas la règle des «deux pizzas»!

Red Flag Law

- «Le nom est tiré des lois sur la circulation du drapeau rouge du 19e siècle qui ont été adoptées aux États-Unis et au Royaume-Uni. Ces lois obligeaient un piéton à marcher devant chaque automobile en agitant un drapeau rouge pour avertir les conducteurs et les cavaliers de chevaux. En conséquence, l'automobile est obligée de rouler à la même vitesse qu'un piéton. »
- Dans notre contexte, une chose similaire se produit lorsqu'une organisation continue d'utiliser des pratiques incompatibles avec les microservices:
  - Processus en cascade, tests manuels, commits pendant des fenêtres de maintenance périodique prédéterminées. Équipes d'experts à travers le système (développeurs, testeurs, assurance qualité)
- Par conséquent, il n'y a aucun avantage à migrer vers une architecture de microservices.
- Avant d'adopter une architecture de microservice:
  - Adoptez les méthodes DevOps pour permettre l'automatisation et l'intégration entre les équipes.
  - Réorganisez vos équipes, des équipes d'experts aux équipes transversales par service.
  - Vous pouvez adopter le modèle «Strangler Application» pour appliquer ces modifications de manière incrémentielle.

Au lieu d’épilogue

- L'adoption d'architectures de microservices a connu des réussites.
  - Amazon, Netflix, The Guardian, Red Hat, IBM.
- La dégradation de l'architecture et l'érosion de la conception ne sont pas le privilège des monolithes.
- Le succès d'une architecture de microservices dépend de bonnes décisions de conception et d'équipes compétentes.
- Considérez toujours les avantages et les inconvénients et si vous êtes prêt à adopter.

La prochaine fois