LES MICROSERVICE

Une introduction de B. Adanlessossi

PRINCIPES ET PHILOSOPHIE

DU MONOLITH A L'ARCHITECTURE MICROSERVICE.

AVANTAGES ET DESAVANTAGES

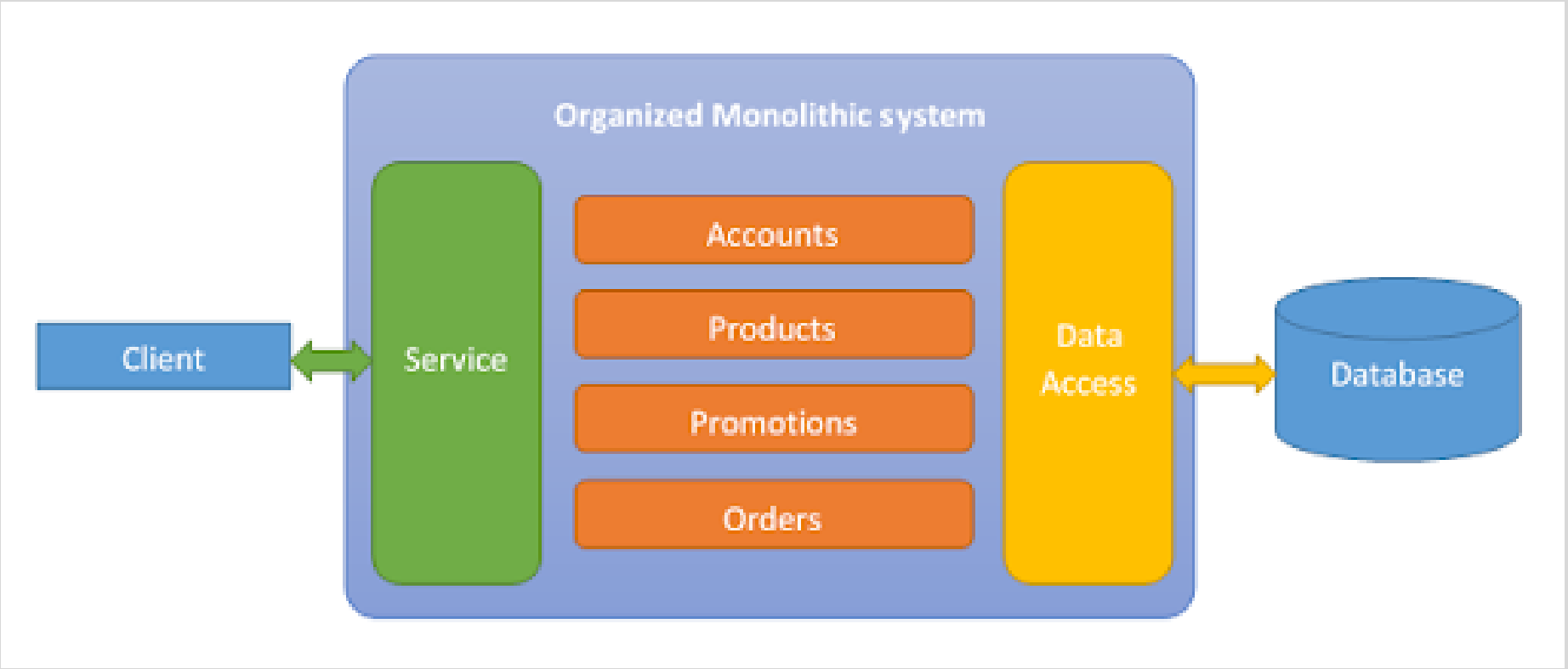
COMMENT ET QUAND UTILIZER LES MICROSERVICES.

DU MONOLITHE A L'ARCHITECTURE MS

- Qu'est ce qu'un Monolithe?
- Selon Wikipedia,
- "Un monolithe est un bloc de pierre massif monumental de grande dimension, constitué d'un seul élément, naturel ou taillé, éventuellement déplacé.
- L'architecture monolithe (ou monolithique) est une construction réalisée dans un bloc unique d'un seul matériau, et l'industrie lithique désigne l'ensemble des objets en pierre taillée par les humains.

L'ARCHITECTURE MONOLITHE

- En genie logiciel, nous avions jusqu'à present malheureusement construit des monolithes!!!



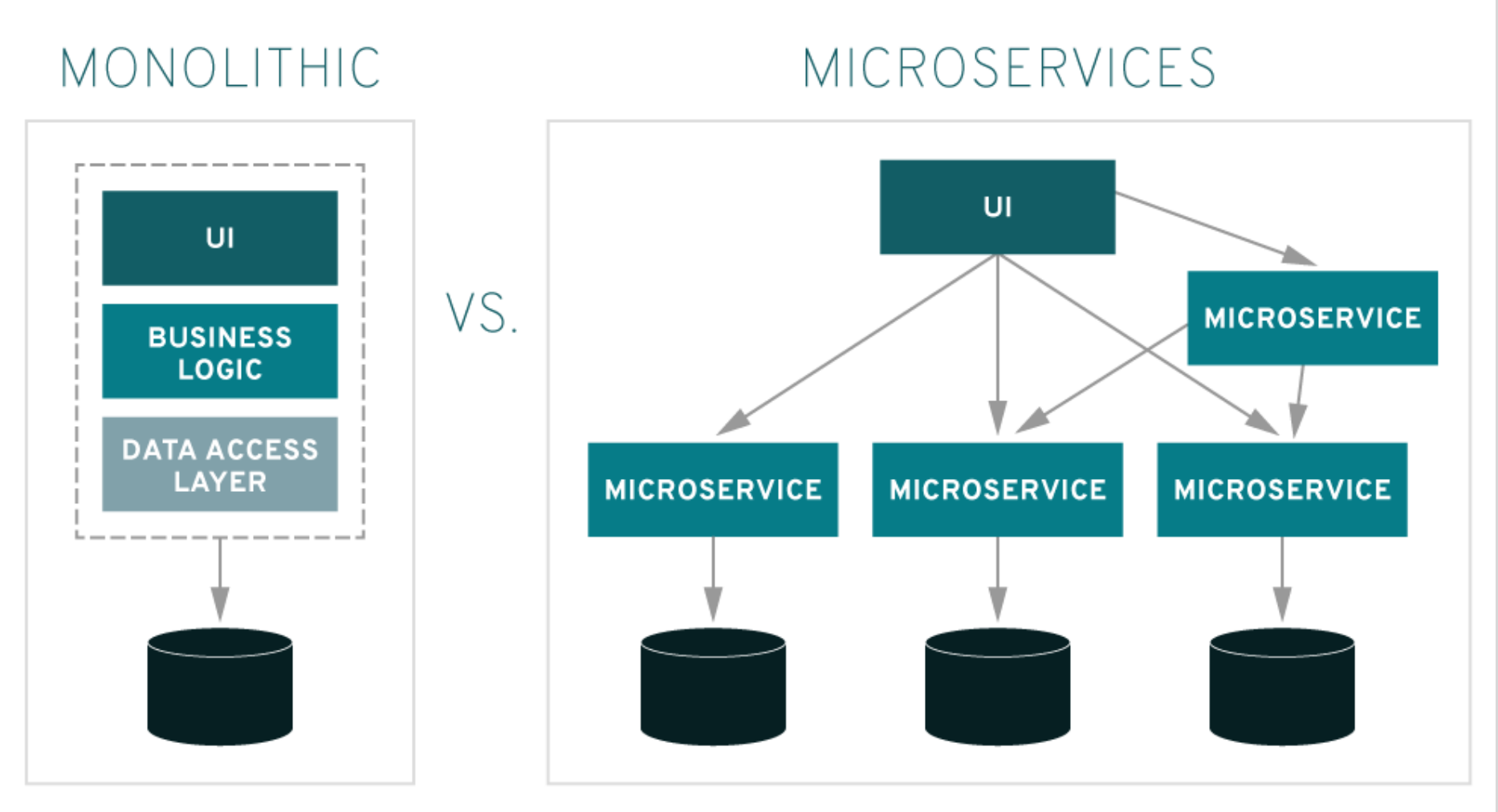
PRINCIPALES CARACTERISTIQUES D'UNE L'ARCHITECTURE MONOLITHE

- Design, developpé et deployé comme une seule unite
- Complex et difficile à maintenir
- Difficile de pratiquer la méthodologie agile
- En cas de changement, il faudra redeployer toute l'application
- Au fur et à mesure que le monolithe grandit, des problems de performance se présentent
- Une seule function erronnée peut rendre toute l'application inutilisable
- Difficulté d'adopter de Nouvelles technologies ou frameworks
- Etc.

QU'EST UN MICROSERVICE

- Avertissements:
  - Micro ne veut pas dire "petit". Un monolithe peut valablement se pretendre microservice.
- Un microservice est une application construite pour exécuter un seul service, mais de manière bien exécutée.
- Un site de eCommerce sera comprise de multiples microservices communicant entre elles pour executer toutes les fonctions d'un commerce en ligne: gestion des clients, payments, produits, commandes, livraison, approvisionnement, etc.
- Un microservice doit faire une seule chose à la fois et le faire très bien
- Un microservice possède sa propre base de donnée
- Un microservice communique avec les autres microservices par des événements
- Nous élaborerons au fur et à mesure que nous avançons.

MONOLITHE CONTRE MICROSERVICE



CARACTERISTIQUES D'UN MICROSERVICE

- Un microservice est développé autour du métier (DDD)
- Autonome: développé, déployé et scaled de manière autonome
- Tolérance à la faute
- Gestion décentralisée des données
- Gouvernance des services
- Observabilité

AVANTAGES DU MICROSERVICE

- Développement rapide et agile des fonctionnalités métier
- Remplaçabilité
- Isolation et prédictibilité
- Déployement agile et à échelle
- Alignement avec la structure de l'organisation

DESAVANTAGES DU MICROSERVICE

- Communication entre les services (Evénements)
- Difficulté de gouvernance du service (gestion du cycle de vie, tests, métriques, observabilité, découverte, qualité du service, etc.)
- Complexité des données transactions distribuées (consistence éventuelle)
- Dépend abondament de la méthodologie de déployement (conteneurs docker et système d'orchestration.

COMMENT ET QUAND LES MICROSERVICES

- L'architecture microservice est idéale si l'architecture de votre entreprise requiert de la modularité
- Si le problème métier que vous devez résoudre paraît simple, vous n'auriez pas envie d'utiliser une architecture microservice.
- Si votre logiciel veut embrasser le monde des "conteneurs" - > microservice
- Si votre système est trop complexe pour être ségrégué en microservice, vous devriez isoler des zones susceptibles d'être développées en microservices, avec un impact minimal sur l'ensemble.
- La comprehension des domaines du métier est primordiale dans le développement d'un microservice
- Chaque domaine, son microservice! Nous y reviendrons.

DESIGN D'UN MICROSERVICE

DOMAIN DRIVEN DESIGN.

PRINCIPES DE DESIGN

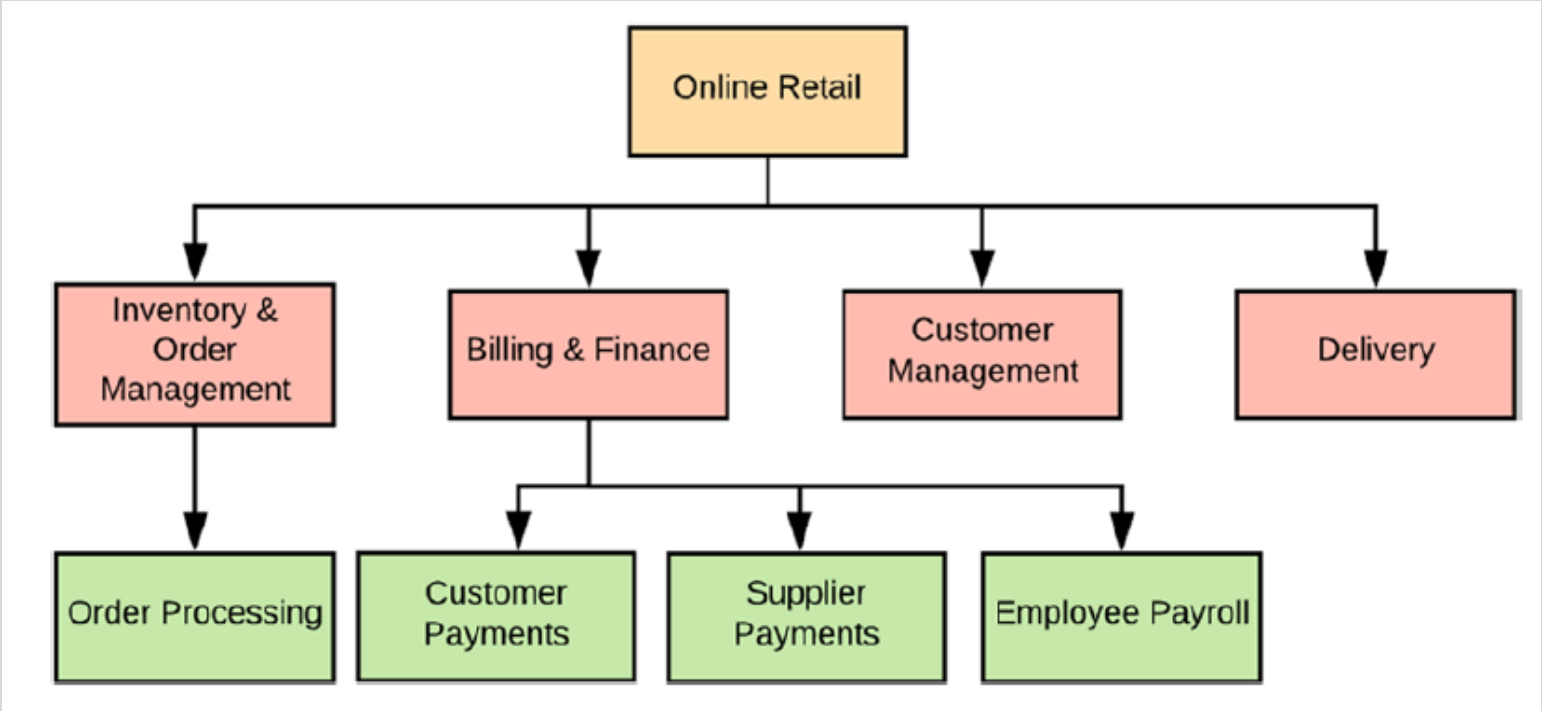
APPLICATION 12 FACTEURS.

DOMAIN DRIVEN DESIGN

- Développement axé sur le métier
- Chaque logiciel que nous développons est d'une façon ou d'une autre lié à l'activité d'un utilisateur ou dans son intérêt.
- Le context dans lequel l'utilisateur utilise l'application est appellé "domaine".
- Certains domaines concernent des mots reels comme acheteur, vendeur, partenaires, etc
- D'autres referent des valeurs intangibles comme crypto-monnaie, portefeuille bitcoin, etc.
- Un logiciel peut également être le domaine quand vous construisez un logiciel pour les logiciels. Exemple un outils de gestion de configuration de logiciels.

DIVISER POUR CONQUERIR

- Le premier principe du Domain Driven Design est “diviser pour conquérir”!!
- Exemple d’une application de vente en ligne.



DIVISER POUR CONQUERIR - BIS

- Dans l'exemple precedent, chaque département de l'entreprise est traité comme un sous-domaine.
- L'identification du domaine et sous-domaines du métier est d'une importance critique et capitale dans l'elaboration d'une architecture
- Le plus important challenge auquel sont confrontés les architectes, est de proposer pour chaque service, le juste degré de granularité!
- Reprenons la loi de Conwaw: "toute organization qui construit un système, tend à reproduire une copie de la structure de commnunication de l'organization dans le système".
- Nous verrrons plus loin comment cette communication se réflète…

PRINCIPES DE DESIGN

- Ces principes devraient nous aider à construire une architecture solide.
- Le temps du développement à la production, la scalabilité, la complexité, la localisation et la résilience sont les éléments clés du design.
  - Forte cohésion et couplage faible entre les composants
  - Résilience:
    - Timeout
    - Circuit breaker
    - Bulkhead (compartiments)
    - Steady state (éloigner toute intervention humaine)
    - Fail fast (no retry, use circuit breaker)
    - Let it crash (il vaut mieux abandoner la partie fautive)
    - Handshaking (permet d'établir une communication de confiance)
  - Observabilité (Collecter des données pour prédire/prévoir/auditer le comportement du système)
  - Automation (déployements frequents – ne peut être attaint sans automation)

Les 12-facteurs d'une application

- L'architecture microservice n'est pas uniquement basée sur ces principes. Certains parlent de "CULTURE", "PHILOSOPHIE".
- Bien sûr que le design est la pierre angulaire, mais c'est surtout une question de communication et de collaboration entre des ingénieurs en informatique et les experts du domaine.
- Le "12 factor App" est un manifeste publié par Heroku en 2012 et est une collection de bonnes pratiques pour construire et maintenir des application robustes, scalables et portables.
- Aujourd'hui ce manifeste est devenu un "mantra" pour tout développement et deployement d'applications basées sur l'architecture microservice.

La base du code - Codebase

- Pour developer avec succès, il faut avoir la même base pour le code source de l'application
- Cette base commune permet de tracer le cycle de développement, les changements au cours du temps et les développeurs impliqués.
- Tout développeur devrait pouvoir accéder à cette base commune
- Avoir un repository par microservice
- Exemples: Git, GitLab, CodeCommit

Les dépendences

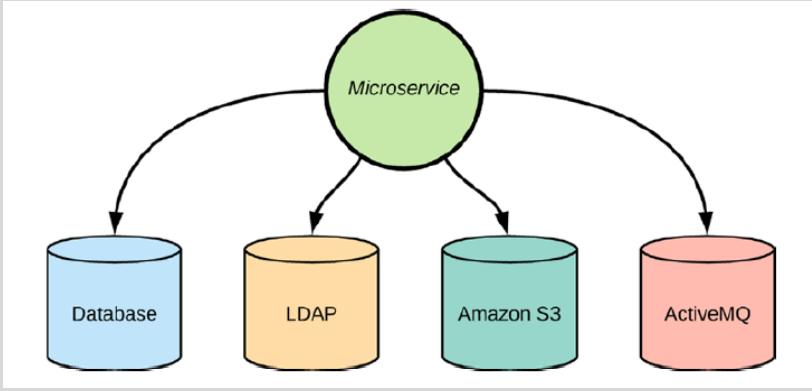
- Ce facteur dit que si votre dépend d'autres librairies vous devriez les declarer et les gérer dans un fichier unique
- Pour une application Java, ce serait des declarations dans les fichiers pom.xml
- Mais de nos jours, les déployements font confiance aux conteneurs docker pour gérer leurs dépendences.

Configuration

- Il faut configure le maximum possible d'éléments "codés en dur" de votre application et les sécuriser avec des clés.
- Exemple: les URL, les noms des bases de données, les environnements, etc.
- Une erreur commune est de ne pas sécuriser ces éléments.

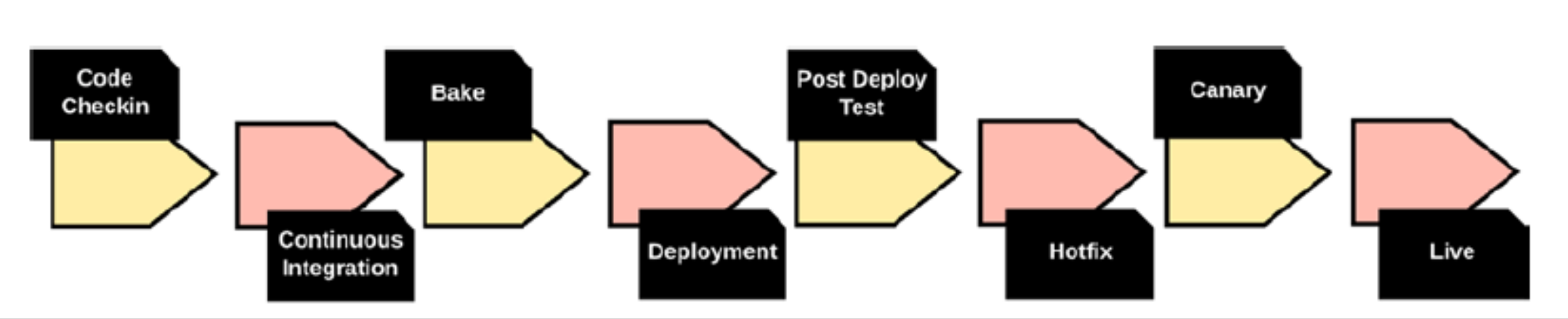
Services annexes – Backend Services

- Ce sont des services annexes dont votre application a besoin pour son fonctionnement normal.
- Il peut s'agir d'une base de données, un LDAP ou DNS, un système de cache, un broker de messages, un service distant de paiement, etc.
- Ce facteur dit qu'il "*faut traiter les Backend Services comme des ressources attachables/détachables à volonté*".
- En d'autres mots, nous devrions être capables de changer de base de données sans impacter le système, de la même manière que pour le DNS, le Système de Messagerie et tout service annexe.



Build, release and run!

- Ce facteur insiste sur la separation entre les activités de construction, la publication et l'execution d'une application.
- Pour assurer la qualité du développement, il faut avoir une claire separation entre ces phase d'activité, depuis le premier checking du code, jusqu'à la mise en production "Live"!



Les processus

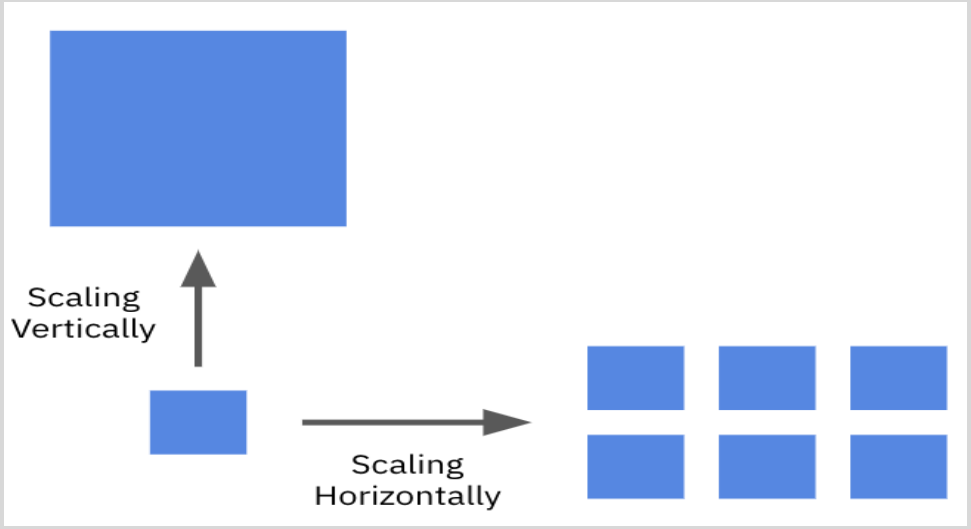
- Le sixième facteur insiste sur le fait que les processus doivent être stateless et doivent éviter ce qu'on appelle "sticky session".
- Stateless veut dire que l'application ne doit pas compter sur aucune données en mémoire avant et après execution.
- Si nous devrions maintenir un "état" en mémoire et le répliquer sur tous les services, l'architecture serait complexe à maintenir.
- Une architecture stateless permet de répliquer facilement l'application sans nécessité de coordination.
- Ce principe est donc traité comme: "***Share Nothing Architecture***", qui dit qu'il ne faut jamais partager le disque dur our la mémoire avec un autre service.

Définition du port

- Le septième facteur dit que votre application doit avoir la possibilité de définir sur quell port elle s'execute et rendre ce port découvrable par les autres applications.
- Votre application ne doit pas dépendre du port assigné par votre serveur d'application.

La concurrence

- Il y a 2 façons de faire le scaling d'une application:
  - Scaling vertical: dans ce cas, on rajoute plus de mémoire CPU et de disque dur
  - Scaling horizontal: la même application est installée en plusieurs exemplaire sur différents nodes.
- Ce facteur dit qu'une application doit être capable de scaler horizontalement, in an out.
- C'est un important aspect, car en cas d'affluence, votre application doit être capable de se démultiplier horizontalement pour servir tous les clients, puis réduire le nombre d'instances dès que l'affluence diminue.



Disposabilité

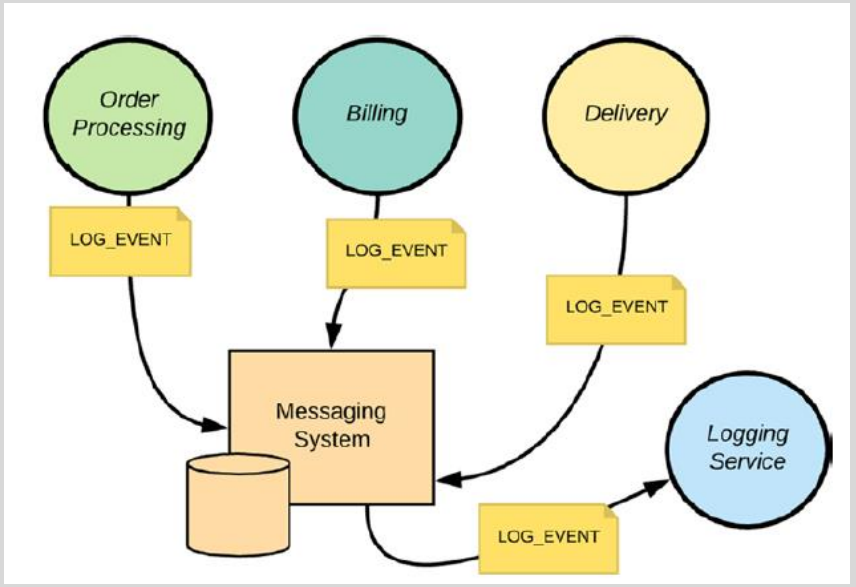
- Votre application doit être capable de démarrer et de stopper rapidement.
- Si vous architectez un microservice, assurez vous que les services ne soient pas bloqués par d'autre processus lors du démarrage.
- Google utilise par exemple des milliers d'instances pour servir ses moteurs de recherche.
- Ceci encourage le principe d'***un microservice par host***.
- Le hardware ne coute plus rien! On peut en disposer comme de tout autre disposable.

Parité Développement/Production

- Ce facteur insiste sur le fait que l'environnement de développement, de tests, de recette et de production doivent être aussi identiques que possible.
- En mode développement, nous disposons par exemple souvent d'une base de données MySQL, alors que la production est une base de données Oracle.
- Les problèmes seront découverts seulement en production, ce qui est très mauvais!
- Quelques fois le manque de ressources ne nous permettent pas d'avoir l'infrastructure necessaire pour chacune des phases de développement, de test et de production.
- La norme de développement et de déploiement de microservices est d'utiliser des conteneurs docker.
- Le principal but de l'architecture microservice et le développement et déploiement rapide d'application!

Log

- Les logs de l'application doivent être considérés comme des flux d'évenements de notre application.
- Ils permettent d'identifier et d'isoler les problems survenant au niveau de l'application et peuvent être utilisés comme un instrument d'audit.
- Dans un environnement de microservice, il est important de tracer et de corréler les différents appels entre services du système.



Processus d'administration

- Ce dernier facteur met en lumière la nécessité d'avoir un interface administratif pour le service.
- Ces taches administratives peuvent être une migration de base de données ou bien un script shell qui peut être execute de manière interactive.

Au-delà du 12-Facteur

- API en premier: commencer par définir un format de communication API REST (contract first)
- Télémétrie: collection de données classées en 3 catégorie, performance, santé et données spécifiques au métier:
  - Nombre de requêtes (réussies, fautives, etc.)
  - Durée d'une requête/réponse
  - Etc.
- Sécurité: chaque microservice doit prendre la sécurité au sérieux dès les premières phases de développement.

COMMUNICATION ENTRE SERVICES

- Dans l'exemple du slide 15, chaque département de l'entreprise est traité comme un sous-domaine.
- L'identification du domaine et sous-domaines du métier est d'une importance critique et capitale dans l'elaboration d'une architecture
- Le plus important challenge auquel sont confrontés les architectes, est de proposer pour chaque service, le juste degré de granularité!
- Reprenons la loi de Conwaw: "toute organization qui construit un système, tend à reproduire une copie de la structure de commnunication de l'organization dans le système".
- Voyons voir comment cette communication se réflète…

COMMUNICATION ENTRE SERVICES

SYNCHRONE OU ASYNCHRONE.

FORMATS/TYPES DE MESSAGE

DEFINTION DE SERVICE ET CONTRATS.

INTRODUCTION

- Dans une architecture microservice, les services communiquent entre eux à travers des appels reseaux.
- Un système est collection de services collaborant pour servir les desseins d'un métier spécifique
- De ce fait, une application basée microservice peut être distribuée sur plusieurs reseaux et plusieurs hôtes.
- Un service est orienté sur une capabilité métier mais les interactions entre les service forment un système ou un produit concernant un cas de métier spécifique.
- Une application basée microservice consiste en une collection de services independants qui communiquent entre eux en s'envoyant des messages.

COMMUNICATION SYNCHRONE

- Dans le style synchrone, le client envoie un message au service et attend que la réponse retourne.
- Toutes les 2 parties doivent maintenir la connection en attendant que le client reçoive sa réponse.
- La logique d'exécution du client est bloquée en attendant la réponse.
- Exemples de communication synchrone: REST, gRPC, graphql, Websockets, etc..

COMMUNICATION ASYNCHRONE

- Dans le style asynchrone, le client envoie un message mais n'attend pas de réponse et peut vaquer à autre chose. "Fire and Forget".
- Le client peut bien reçevoir la réponse, "un jour", ou pas du tout de réponse, en fonction du service appellé.
- La réponse peut même parvenir au client par un canal different!
- On distinguera 2 styles de communication asynchrone:
  - Single Receiver: un message donnée est délivré de manière fiable à un seul consommateur, le client. Apache MQ est un bon exemple de Single receiveir.
  - Multiple Receivers: un message donné peut être délivré fiable à plusieurs consommateurs souscrits, introduisant l'idée de Publieur/Souscriveur. Dans cette gamme, on trouvera Apache Kafka

FORMATS DE MESSAGE

- La communication entre les microservices se fait par l'échange de messages.
- La determination du format à utiliser est d'une importance capitale.

- JSON/XML
  - JSON est fortement utilize comme format de message. XML perd gentiment du terrain
- Protocol Buffer
  - Très utilisé dans les application utilisant gRPC.
- AVRO
  - Apache Avro est un système de sérialisation fournissant une riche structure de données pour la représentation des données, format compact, grand support pour la plupart des languages de programmation, etc.

DEFINITION DE SERVICE ET CONTRATS

- Lorsque vous êtes en charge d'implémenter un processus métier comme un service, vous devez définir un contrat de service.
- Comme la construction de microservice est basé sur REST, nous pouvons utiliser le même style de communication pour définir le contrat du service.
- L'architecture microservice se base sur les API standard de REST, comme openAPI pour la definition des contrats

CONCLUSION

- Ceci n'est qu'une introduction à l'architecture microservice
- Plus de renseignements:
- https://microservices.io/
- https://martinfowler.com/microservices/

https://openclassrooms.com/fr/courses/4668056-construisez-des-microservices/5122300-apprehendez-larchitecture-microservices