

## Séance 11

# Micro-services et architecture serverless



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Rappels

- Définition et services offerts par le cloud computing
  - Modèle standardisé par le NIST
  - Les grands challenges et la situation aujourd'hui
- Description et analyse de différents modèles de services
  - Infrastructure, plateforme et service (IaaS, PaSS, SaaS)
  - Backend pour applications mobiles (BaaS)
  - Sécurité et fonction (SECaaS, FaaS)

# Objectifs

- Définition et principes de la notion de **microservice**
  - Définition et bénéfices de l'architecture microservice
  - Rôle et comportement de l'architecte
  - Modélisation et intégration dans le cadre de microservice
- Architecture **serverless**

*Définition et concepts, FaaS, avantages et inconvénients*

# Tendance

- Domain-Driven Design par Eric Evans

*Importance de représenter le monde réel dans son code*

- Concept de continuous delivery

*Passage plus effectif et efficace en production*

- Machines communicantes à l'aide du web

- Redimensionner des machines à l'aide de la virtualisation

*Permet aussi notamment d'automatiser l'infrastructure*

# Vers le microservice

- Émergence de la notion de **microservice**

*Domain-driven design, continuous delivery, on-demand virtualisation, infrastructure automation, small autonomous teams, systems at scale*

- Nouvelle **tendance/pattern** suite à usage du monde réel

*Existente par l'apparition de tous les éléments précédents*

- Plusieurs **avantages** des microservices cités par les entreprises

- Livrer des softwares plus rapidement
- Adopter plus facilement des nouvelles technologies



**Microservice**

# Microservice

- Petits services autonomes qui travaillent ensemble

*Différences notables avec les services « traditionnels »*

- Petit service qui focuse sur faire une seule chose et bien
  - Pas facile de gérer un gros codebase, même si très modulaire
  - Code similaire à plusieurs fonctions se répand partout
- Entité autonome séparée pouvant être sur une machine propre
  - Pouvant être déployé sur une plateforme PaaS
  - Peut aussi être son propre processus d'exploitation

# Single Responsibility Principle

- Principe de responsabilité unique (SRP)

*Regroupement du code en relation ensemble, création de cohésion*

- Définition du SRP énoncée par Robert C. Martin

*“Gather together those things that change for the same reason,  
and separate those things that change for different reasons.”*

# Petit et focus sur une chose

- Développement de **petits services** indépendants
  - Frontières du service aligné sur frontières business
  - Repérage immédiat d'une fonctionnalité
  - Évitement de la tentation de services grossissants
- Petit service peut être **réécrit en deux semaines** (Jon Eaves)
  - *“Small enough, but not smaller”*
  - Doit pouvoir être géré par une petite équipe
  - Difficile à casser en plus petits blocs

# Autonome et séparé

- Isolation complète du service sur machines différentes
  - Ajout d'un éventuel overhead dû à l'isolation
  - Facilité de distribuer le système, de changer les composants
  - Communication entre services par appels réseaux
- Le service doit exposer une API pour communiquer
  - Découpler le service, permettre changements indépendants
  - Choisir API qui sont technology-agnostic

# Bénéfices (1)

- Plusieurs technologies dans un système avec plusieurs services
  - Choisir le bon outil pour chaque job (performance, DB...)
  - Permettre de tester de nouvelles technologies
- Résistance du système aux pannes et bugs
  - Cloison créée par les frontières des services
  - Dégradation douce d'un système, isolation disfonctionnements
  - Réseau et machines sont les sources de défaillances
- Possibilité de scaling de services le nécessitant

*Existence de systèmes d'approvisionnement à la demande*

## Bénéfices (2)

- Offre une plus grande facilité de développement

*Changement sur un seul service suivi d'un déploiement*

- Alignement organisationnel avec les équipes de développement

*Petite équipe bossant sur petit codebase sera plus productive*

- Réutilisation de fonctionnalités permet la composabilité

- Système optimisé pour faciliter le remplacement

*Ajout/suppression/remplacement de services*

# Architecture orientée service (SOA)

- Plusieurs **services collaboratifs** communiquant par le réseau  
*Appels à travers un réseau plutôt que au sein d'un processus*
- Combat l'application monolithique et promeut la **réutilisabilité**
- Pas de consensus sur comment faire du **bon SOA**
  - Protocole de communication (SOAP), vendor middleware
  - Pas de guidance sur la finesse des services
  - Mauvaise guidance sur les points de découpe d'un système

# Technique de décomposition

- Utilisation d'une **librairie partagée**
  - Partage de code entre plusieurs équipes et services
  - Pas facilement possible d'avoir une hétérogénéité technologique
  - Mise à l'échelle indépendante de services pas évidente
  - Déploiement d'un système doit se faire avec la librairie
- Développement de **modules** indépendants dans le langage
  - Open Source Gateway Initiative (OSGI) pour plugins Java
  - Module Erlang stoppé, redémarré et mis à jour à chaud
  - Mêmes défauts que librairie partagée

# Architecte



# Architecte

- Rôle important de **vision technique** unifiée du projet

*Doit aider à livrer au client le produit qu'il désire*
- Peut travailler sur **un seul projet** ou sur plusieurs
  - Joue aussi le rôle de *technical leader*
  - Collabore avec plusieurs équipes ou sur toute une institution
- L'architecte peut être critiqué pour son **impact important**
  - Impact direct sur la qualité logicielle du produit
  - Influence les conditions de travail des équipes

# Vision évolutive

- Un **logiciel vivant** continue sans cesse d'évoluer

*L'évolution continue même une fois passé en production*

- Grande importance du **retour utilisateur**

*Prise en compte du feedback des utilisateurs/clients*

- **Créer framework** au sein duquel le bon système pourra émerger

*Ne pas vouloir architecturer le produit final parfait dès le départ*

# Planificateur de ville

- Architecte logiciel plutôt comparé à un **planificateur de ville**

*Comme lorsque vous jouez à SimCity*

- Optimiser layout de la ville pour **satisfaire besoins** habitants

*En regardant une multitude d'informations et prévoyant le futur*

- Création naturelle de **zones** délimitées par des frontières

*Zone industrielle, résidentielle, commerciale...*

# Approche sur des principes

- Prise de décision revient à **faire des compromis**  
*Technologie, stockage données, plateforme...*
  - **But stratégique** de la compagnie pour rendre clients heureux  
*Très haut niveau, parfois pas technique (conquérir un marché)*
  - Établir des **principes** pour s'aligner sur les buts stratégiques  
*Plus bas niveau et concerne le développement du software*
- “Rules are for the obedience of fools and the guidance of wise men.”  
— Douglas Bader

# 12 Factors App (1)

- Un **principe** est une chose que l'on décide de choisir

*Contrairement à une contrainte qu'il est impossible de changer*

- Méthodologie pour construire des **apps de type SaaS**
  - Format déclaratif pour setup automation
  - Contrat clair avec l'OS sous-jacent, portabilité maximale
  - Déploiement ais  sur plateforme cloud moderne
  - Divergence dev/prod minimale, d ploiement continu
  - Mise   l' chelle facile

# 12 Factors App (2)

## 1 Codebase

*One codebase tracked in revision control, many deploys*

## 2 Dependencies

*Explicitly declare and isolate dependencies*

## 3 Config

*Store config in the environment*

## 4 Backing services

*Treat backing services as attached resources*

# 12 Factors App (3)

## 5 Build, release, run

*Strictly separate build and run stages*

## 6 Processes

*Execute the app as one or more stateless processes*

## 7 Port binding

*Export services via port binding*

## 8 Concurrency

*Scale out via the process model*

# 12 Factors App (4)

## 9 Disposability

*Maximize robustness with fast startup and graceful shutdown*

## 10 Dev/prod parity

*Keep development, staging, and production as similar as possible*

## 11 Logs

*Treat logs as event streams*

## 12 Admin processes

*Run admin/management tasks as one-off processes*

# Pratique

- Assurer que les principes réalisés à l'aide de la **pratique**

*Ensemble de conseils pratiques pour réaliser tâches*

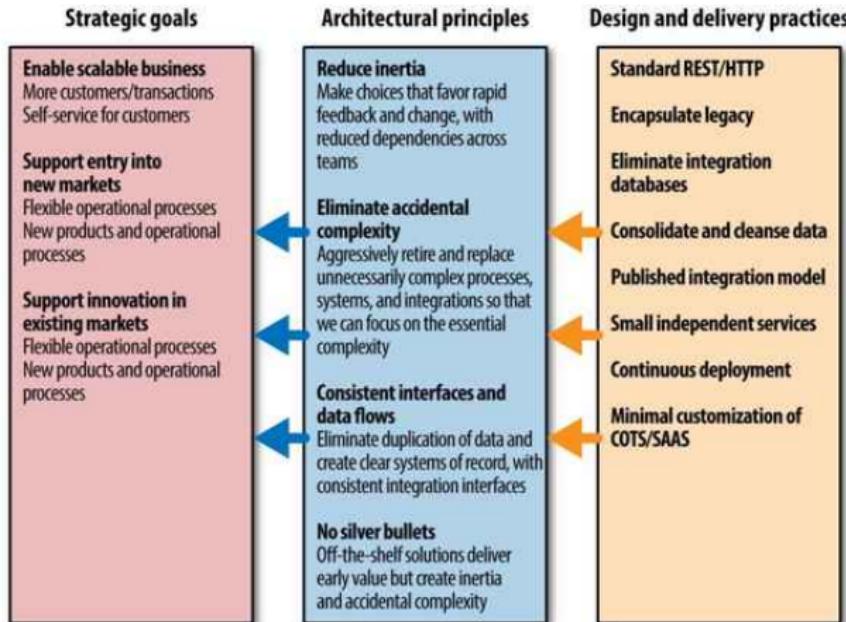
- **Spécifique à une technologie** et suffisamment bas niveau

- N'importe quel développeur doit pouvoir les comprendre
- *Coding guidelines*, logs à capturer de manière centrale, intégration à faire avec HTTP/REST...

- La pratique **changera** plus souvent que le principe

- Étant donné sa nature beaucoup plus technique
- Un principe d'entreprise, deux pratiques pour team Java et C#

# Principe et pratique



Exemple réel de principes et pratiques par Evan Bottcher

# Standard

- Caractéristiques communes d'un **service se comportant bien**

*Service « bon citoyen » au sein du système*

- Importance de pouvoir **monitorer** le système complet

- Proposer un petit nombre d'**interfaces** pour intégrer client

*Technologie, protocole mais aussi façon d'utiliser (HTTP/REST)*

- Bien gérer la **sûreté** et les défauts potentiels

*Savoir si une requête s'est bien déroulée ou non, code d'erreur...*

- “*It needs to be a cohesive system made of many small parts with autonomous life cycles but all coming together.*” — Ben Christensen, Netflix

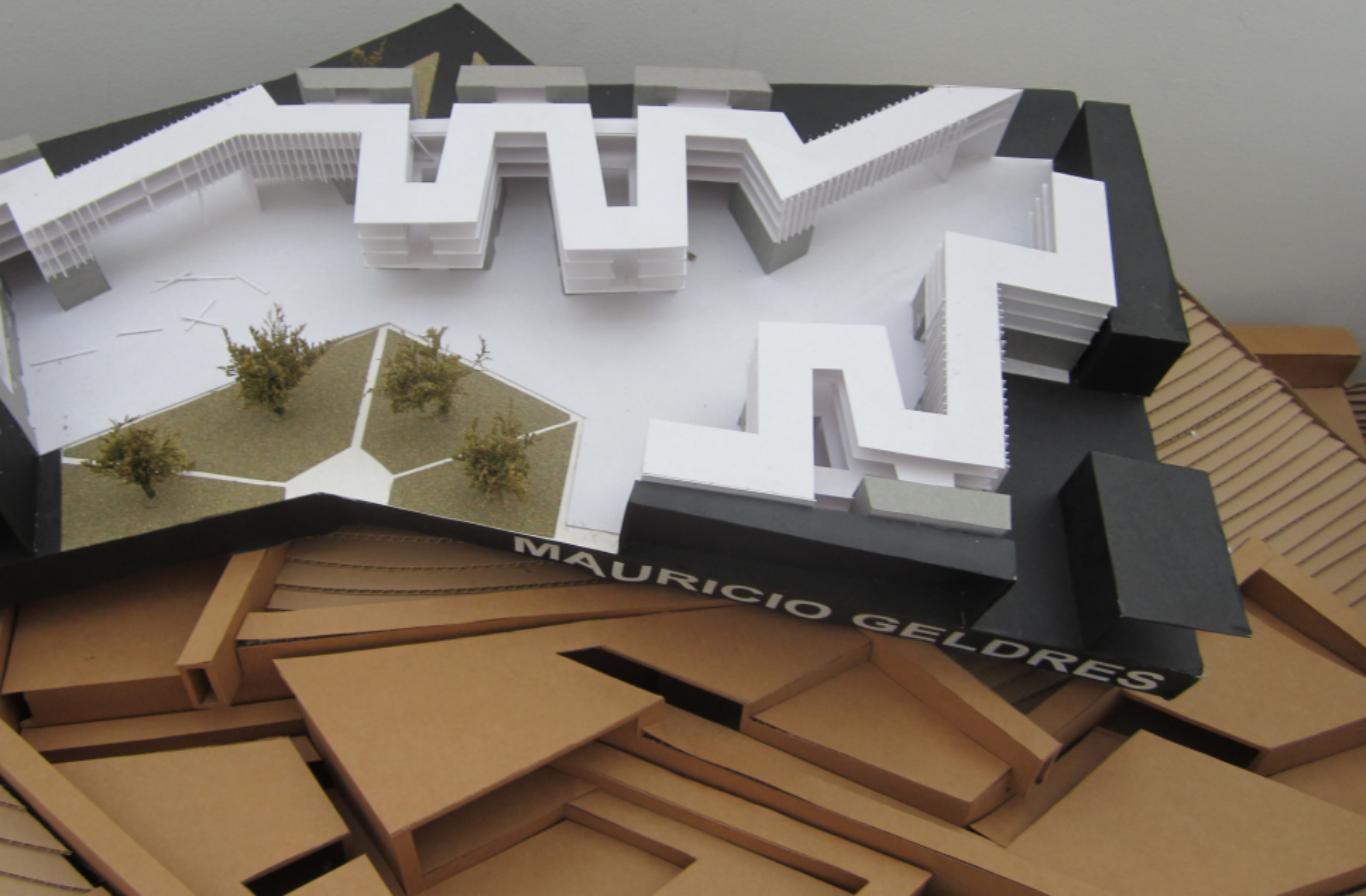
# Monitoring

- Pouvoir connaître l'**état** de chaque service avec métrique  
*Taille des files d'attente, connection aux autres...*
- Obtenir des informations de **santé** de chaque service  
*Le service tourne-t-il toujours, vitesse correcte...*
- Plusieurs outil/plateforme libres existants pour le **monitoring**  
*Graphite fait des métriques et Nagios surveille la santé*

# Gouvernance par le code

- Pas très fun d'assurer que les gens suivent les guidelines  
*Peut sembler lourd à implémenter tous ces standards...*
- Proposer des exemples de code pertinents et illustratifs  
*Pas exemple parfait isolé, mais contextualisé dans real-world case*
- Proposer des tailored service template  
*Template de code de service, remplir avec fonctionnalités désirées*

# Modélisation



# Bon microservice

- Bon microservice avec faible couplage et grande cohésion

*Deux principes très importants en programmation orientée objet*

- Service indépendant avec faible couplage

*Pouvoir changer et déployer un service sans en affecter d'autres*

- Grande cohésion qui rassemble comportements liés

*Localisation à un seul endroit pour changer un comportement*

# Contexte borné

- Plusieurs **contextes bornés** dans un domaine d'application

*Concept proposé par Eric Evans sur le domain-driven design*

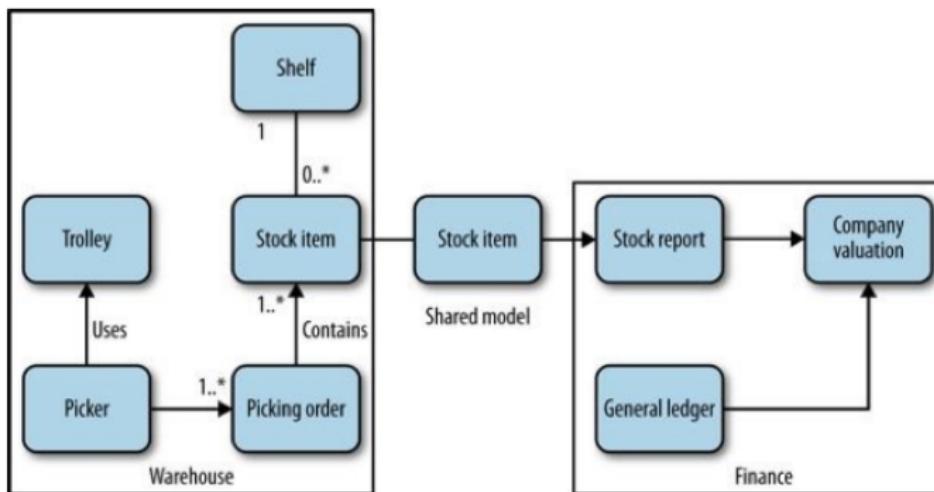
- Un contexte borné comporte **deux types de choses** (modèles)
  - Choses ne devant pas être communiquées à l'extérieur
  - Choses partagées avec les autres contextes bornés
- **Interface explicite** décrivant le partage en dehors de la frontière

*“A specific responsibility enforced by explicit boundaries”*

# Modèle partagé et caché

- MusicCorp composé de deux départements **entrepôt et finance**

*Partage d'une partie d'information concernant le stock*



# Module et service

- Correspondance entre contexte borné et **module**

*Réduction du couplage et identification des partages*

- Correspondance entre modules et **microservice**

- Module au sein d'un système monolithique
- Déplacement vers un service complètement séparé et isolé

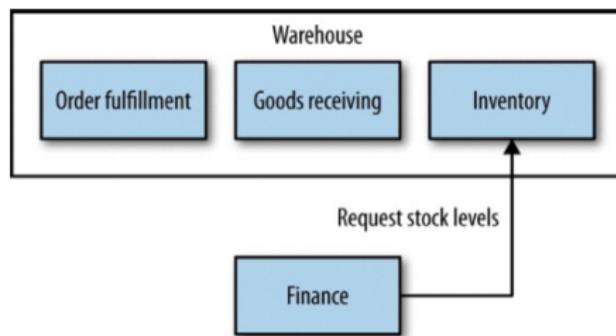
- Contexte borné offre des **capacités**, pas d'échange de données

- Capacités offertes au reste du domaine
- Entrepôt propose d'obtenir l'état actuel du stock
- Ne pas tomber dans le service CRUD par défaut

# Délimitation incrémentale

- Identification de **contexte de haut niveau** avant raffinement

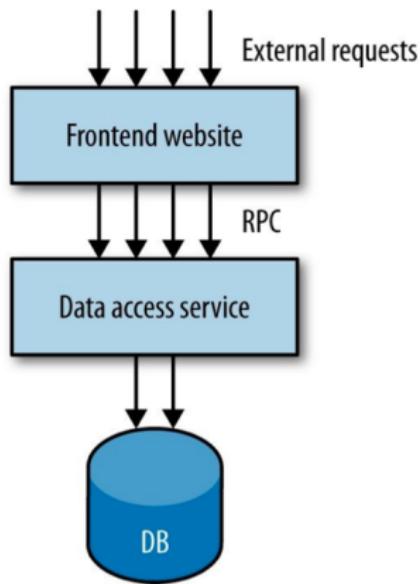
*Approche incrémentale, pouvant être cachée de l'extérieur*



# Frontière technique

- Ajout d'une équipe de développement et **découpe d'un projet**

*Front-end stateless pour site web, backend interface RPC sur DB*



Intégration

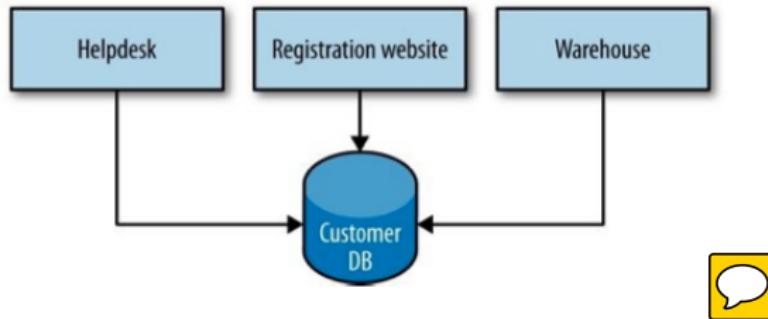


# Technologie d'intégration

- Pléthore de choix pour **communiquer avec microservice**  
*SOAP, XML-RPC, REST, protocol buffers...*
- Plusieurs **propriétés** sur la technologie à prendre
  - Éviter que des changements cassent tout  
*Ajout d'un nouveau champ ne doit pas impacter les clients*
  - Maintenir une API agnostique par rapport à une technologie
  - Rendre l'API simple à utiliser pour le consommateur
  - Cacher tous les détails d'implémentation interne

# Base de données partagée

- **Données centralisées** dans une unique base de données  
*Tout changement est fait par un service direct dans la DB*
- **Couplage fort** entre les différents services et les données
  - Un changement de données impacte tous les services
  - Coincé avec un choix technologique de moteur DB



# Collaboration

- Deux grands **modes de collaboration** de services

*En communication synchrone ou asynchrone*

- Mode synchrone par **requête/réponse**



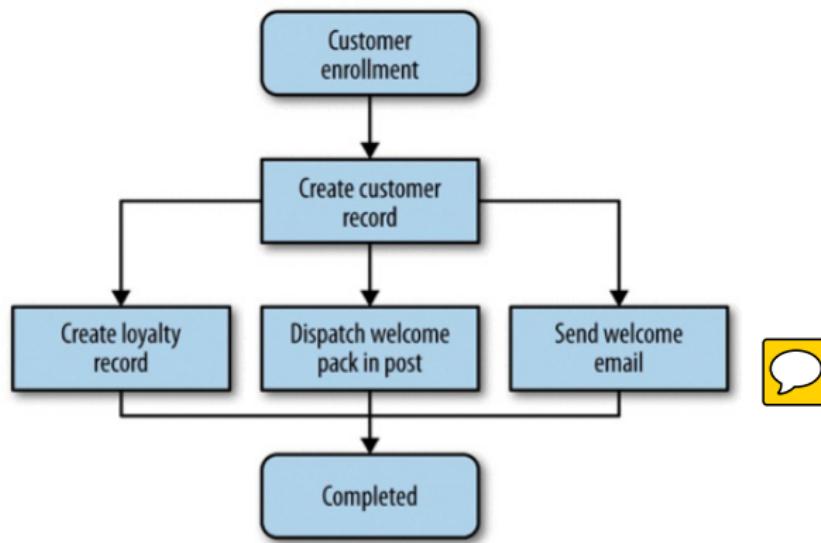
- Appel sur un serveur qui bloque jusqu'à la fin de l'opération
  - Plus facile à raisonner et analyser

- Mode asynchrone basé sur les **événements**

- L'appelant n'attend pas la fin de l'opération et continue
  - Très intéressante pour job long à s'exécuter

# Orchestration ou chorégraphie

- Gérer des **processus business** s'étendant sur plusieurs services  
*Par exemple, inscription nouveau client déclenche plusieurs choses*



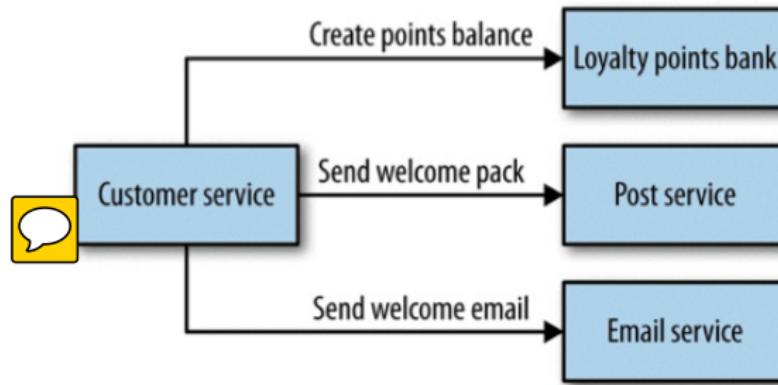
# Orchestration

- **Orchestration** par un élément central

*Cerveau central qui guide et mène le processus*

- Séquence d'**appels requête/réponse** vers différents services

*Trop grand rôle central d'autorité pour le service customer*



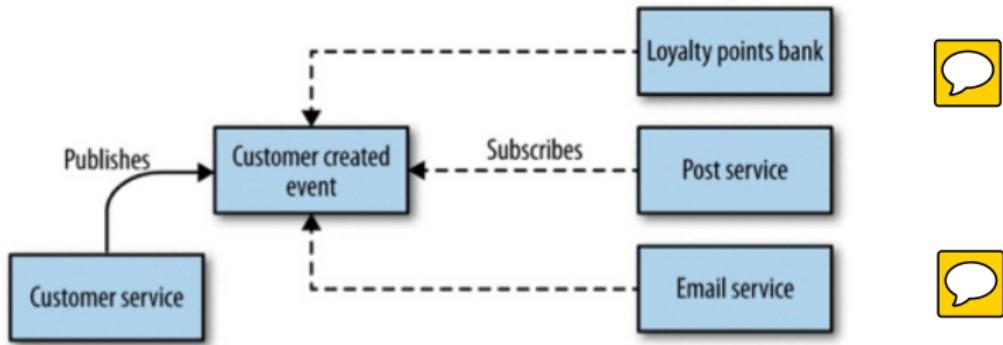
# Chorégraphie

- Chorégraphie de plusieurs acteurs

*Informer les acteurs qu'ils ont un job à réaliser*

- Facile d'ajouter nouveaux traitements et meilleur découplage

*Vue business pas aussi bien reflétée qu'avec l'orchestration*



# Communication requête/réponse

## ■ **Remote Procedure Call (RPC)**

- Faire un appel local qui sera exécuté sur une machine distante
- Interface (SOAP, Thrift, protocol buffers) ou plus couplé (RMI)
- Cacher complexité de l'appel distant, exploitant réseau robuste

## ■ **REpresentational State Transfer (REST)**

- Création de représentation d'objets sur base de requêtes
- Découplage stockage interne et représentation
- Souvent relié au protocole HTTP



# Architecture serverless

# Serverless (1)

- **Architecture serverless** possède forte dépendance externe
  - Services de tiers comme avec Baas
  - Code custom dans container éphémère comme avec Faas
- Déplacement de comportement vers le **front-end**

*Suppression du besoin du serveur qui doit être toujours running*
- **Réduction significative** cout et complexité opérationnelle

*En échange d'une dépendance à un vendeur*

# Serverless (2)

- Deux principales **définitions** mais avec overlapping
  - Logique serveur et état gérés par tiers dans le cloud (BaaS)  
*Application cliente riche comme SPA ou mobile*
  - Logique serveur écrite en partie par développeur (FaaS)  
*Exécuté dans container stateless éphémère géré par tiers*
- Architecture basée sur le **FaaS** est la plus récente vision  
*Par exemple les AWS Lambda proposés par Amazon*

# Histoire

- Toujours existence de **server hardware** et processus serveur  
*Mais outsourcé par le développeur d'une application serverless*
- Premier usage du terme **en 2012** comme indication du futur  
*Ci tourne comme un service, plutôt infrastructure que produit*
- Amazon **AWS Lambda** en 2014, API Gateway en 2015  
*“The Serverless Company using AWS Lambda” à re:Invent 2015*
- **Serverless conference** aujourd’hui et de nombreux vendeurs

# Application orientée UI (1)

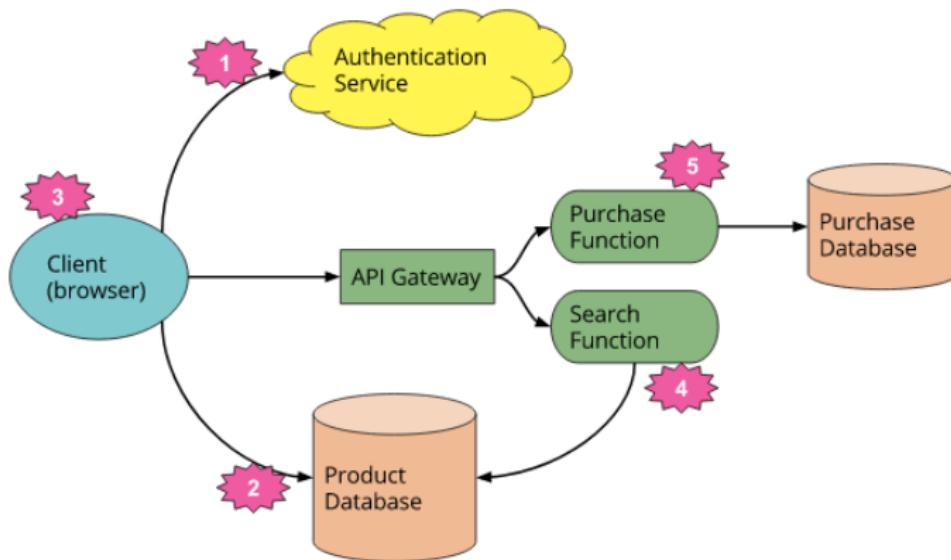
- Exemple d'une **application de e-commerce** de vente d'animaux  
*Implémentation classique 3-Tiers (serveur Java, client HTML/JS)*
- Plupart de la **logique** est dans le backend, client pas intelligent  
*Authentification, navigation entre pages, recherche, transaction...*



# Application orientée UI (2)

- Exemple d'architecture avec les **principes serverless**

*Pas une recommandation ou technique de migration*



# Application orientée UI (3)

- Authentification (1) et accès liste produit (2) par un **BaaS**  
*Auth0 Webtask et Amazon Dynamo, par exemple*
- Le client (3) devient une **Single Page Application** (SPA)  
*Logique user session, page navigation, view from database...*
- Fonctionnalité lourde, chère et critique **côté serveur** (4, 5)  
*Sous forme de FaaS accédée à l'aide d'un API Gateway*

# Application orientée message (1)

- Application devant **processer des données** efficacement  
*Clic sur une pub redirige user et stocke pleins de stats*
- Serveur de pub **poste un message** dans un canal lors d'un clic  
*Traitement asynchrone par l'application click processor*



# Application orientée message (2)

- Fonction FaaS pour **consommer les messages** du canal  
*Comme remplacement de l'application long-lived consumer*
- Exécution dans le contexte **event driven** du vendeur  
*Utilisation du message broker et de l'environnement FaaS*



# AWS Lambda

- Description du produit **AWS Lambda**
  - *AWS Lambda lets you run code without provisioning or managing servers.*
  - *With Lambda, you can run code for virtually any type of application or backend service*
  - *all with zero administration. Just upload your code and Lambda takes care of everything required to run*
  - *and scale*
  - *your code with high availability. You can set up your code to automatically trigger from other AWS services*
  - *or call it directly from any web or mobile app.*

# Function as a Service (FaaS)



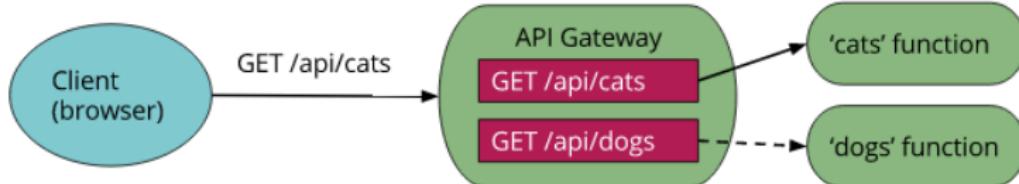
- Exécution de code backend sans devoir déployer un serveur  
*Application standard, JavaScript, Python, Java, Clojure, Scala...*
- Déploiement se fait par **upload de code** chez le fournisseur  
*Scaling horizontal automatique, élastique, géré par fournisseur*
- Fonction déclenchée par un **type d'évènement** à configurer
  - S3 file update, time (scheduled task), message bus (Kinesis)
  - Requête HTTP entrante, directe ou Api Gateway, Webtask...

# Restriction de FaaS

- Pas d'état local sur la machine physique ou instance virtuelle  
*Stateless ou DB, cache (Redis), ou stockage distant (S3)*
- Limitation sur la **durée maximale d'exécution** d'un appel  
*AWS Lambda limite à 5 minutes avant de killer l'exécution*
- Potentielle **latence de démarrage** de la fonction
  - De 10 ms à 2 min sur AWS Lambda, par exemple avec JVM
  - Maintenant la fonction en vie en l'appelant régulièrement

# API Gateway

- Serveur HTTP associant des routes avec une fonction FaaS  
*Fait aussi le mapping des paramètres, et valeur de retour*
- D'autres fonctionnalités sont offertes par API Gateway  
*Authentification, validation des inputs, mapping code réponse*
- HTTP-frontend **microservice** in a serverless way



# Not serverless

- Architecture serverless pas forcément déployable sur PaaS

*“If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless.”*

- Mise à l'échelle d'application FaaS beaucoup plus facile

*Auto-scale au niveau des requêtes, ce qui est difficile avec PaaS*

- Plus grande maturité des PaaS

- Meilleurs outils et maturité des APIs gateway
- In-app readonly cache for optimisation with app 12-Factor

# Procédure stockée

- FaaS pas uniquement un service de **procédures stockées** (SP)

*Ne se limite pas à offrir une procédure d'accès à la DB*

- Différences de **contraintes** par rapport aux SP

- Framework ou (extension de) langage spécifique au vendeur
- Difficile à tester car doit être exécuté avec la DB
- Difficile à versioner, pas vraiment une application séparée

# Avantage

- Réduction du **cout opérationnel** par outsourcing
    - Gestion serveur, base de données, logique applicative...
    - Effet d'économie d'échelle, partage ressources fournisseur
  - Plus **avantageux** que d'autres types de service dans le cloud
    - Cout de développement réduit BaaS (Auth0, Firebase...)
    - Mise à l'échelle des couts FaaS (paie que ce qu'on consomme)
  - Contribue à un **“greener computing”**
- “*Typical servers in business and enterprise data centers deliver between 5 and 15 percent of their maximum computing output on average over the course of the year.*” — *Forbes*

# Inconvénient

- Plusieurs **désavantages sont inhérents** à l'outsourcing
  - Prise de contrôle par un tier sur une partie de l'application
  - Location multiple d'un même hardware à plusieurs clients
  - Vendor lock-in, pas évident de passer d'un vendeur à un autre
  - Pas d'état sur serveur pour les FaaS
- Inconvénients **liés à l'implémentation** pourraient s'amoindrir
  - Pas de configuration, même pas de variable environnement
  - Limitation du nombre d'exécutions sur son compte, self-DoS
  - Durée d'exécution limitée à 5 minutes, latence de démarrage

# Crédits

- <https://www.flickr.com/photos/edwardconde/11439019115>
- <https://www.flickr.com/photos/eugenuity/33363332474>
- <https://www.flickr.com/photos/chasqui/8542554225>
- <https://www.flickr.com/photos/brandsvig/6318694171>
- <https://www.flickr.com/photos/timdorr/217734350>