Architecture MicroService

ARCHITECTURES MICROSERVICES

- Limites des architectures monolithiques
- Architectures Micro-Services
- Mise en œuvre d'un Micro-service avec Spring Boot.
  - Data Access Layer: JPA, Hibernate, Spring data
  - Business Layer
  - Web Layer with: (REST API, SOAP API, Graph QL API)

INTRODUCTION À L'ARCHITECTURE MICROSERVICES

Définition & Principes

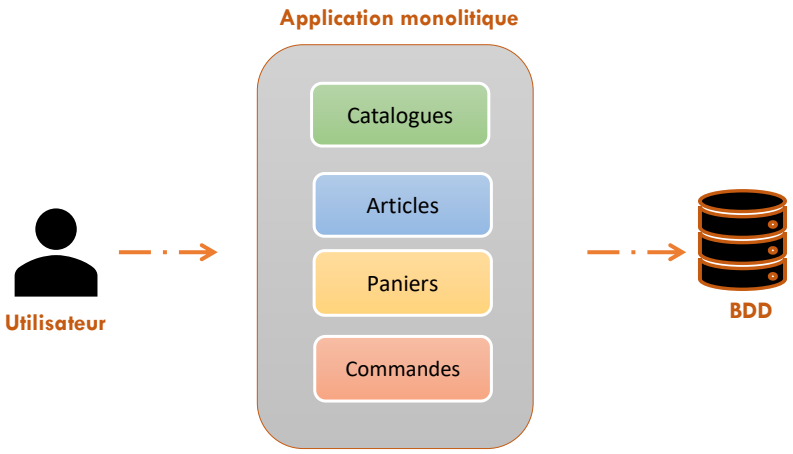
Microservices are small, autonomous services that work together to form a complete application.

Sam Newman

In short, the microservice architectural style is an approach to developing a single application as a **suite of small services,** each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API

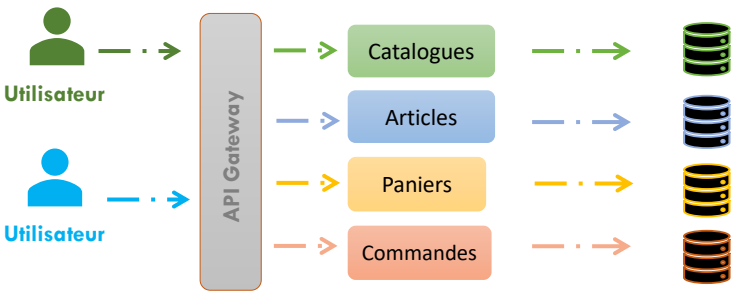
James Lewis and Martin Fowler (2014)

ARCHITECTURE MONOLITHIQUE



ARCHITECTURE MICROSERVICE

Application distribuée



INTRODUCTION À L'ARCHITECTURE MICROSERVICES

Définition & principes

- Les microservices désignent à la fois une architecture est une approche de développement logiciel qui consiste à décomposer les applications en éléments les plus simples indépendants les uns des autres

- Contrairement à une approche monolithique classique, les microservices fonctionnent en synergie pour accomplir les mêmes tâches, tout en étant séparés.
  → Chacun de ces composants ou processus est un microservice

CARACTÉRISTIQUES DES MICROSERVICES

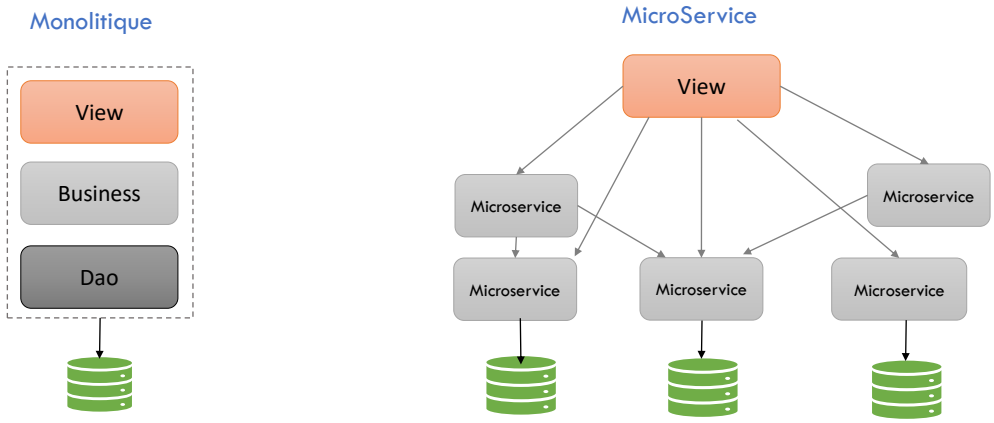
- Indépendance : Chaque service est autonome et encapsule une fonctionnalité spécifique du système. Cela permet de développer, tester, déployer et mettre à l'échelle chaque service indépendamment.
- Isolation des pannes : Une défaillance dans un service n'affecte pas nécessairement les autres, ce qui augmente la résilience globale du système.
- Flexibilité technologique : Chaque service peut être développé avec la technologie qui lui convient le mieux (Java, Python, Node.js, etc.), facilitant l'utilisation des meilleures pratiques et technologies pour chaque cas d'usage.
- Scalabilité : Les microservices permettent une mise à l'échelle horizontale, où chaque service peut être ajusté indépendamment pour répondre à des besoins de charge spécifiques.

CARACTÉRISTIQUES DES MICROSERVICES

Un microservice est donc un service qui ne partage jamais son contexte

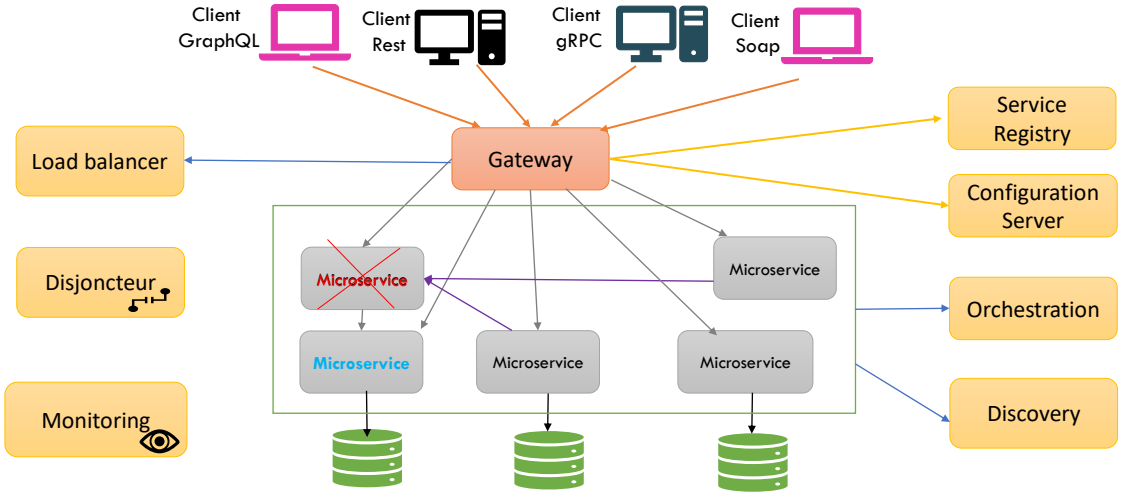
d'exécution avec d'autres microservices mais qui peut communiquer avec eux

AMS VS MONOLITIQUE



Un microservice est donc une fonction principale d'une application qui est exécuté indépendamment des autres services

AMS VS MONOLITIQUE



COMPOSANTS DES MICROSERVICES

- **Gateway :** Un point d'entrée pour les demandes des clients qui achemine les demandes vers les microservices appropriés , effectue l'authentification et l'autorisation et gère d'autres tâches telles que la mise en cache et le mappage demande-réponse.
  - **Outils associés** : Kong, Zuul, Apigee.
- **Configuration Server** : Un référentiel centralisé pour stocker les informations de configuration accessibles à tous les microservices.
  - **Outils associés** : Spring Cloud Config, Vault.
- **Service Registry :** Enregistre les services actifs pour permettre leur découverte.
  - **Outils associés** : Consul, Eureka.
- **Service Orchestration:** Gère le déploiement, la mise à l'échelle et la résilience des conteneurs hébergeant les services.
  - **Outils associés** : Kubernetes, Docker Swarm.

COMPOSANTS DES MICROSERVICES

- **Disjoncteur :** Un mécanisme qui permet d'éviter les pannes en cascade en interrompant la communication entre les services lorsqu'un service ne répond pas.
  - **Outils associés** : Resilience4j, Hystrix.
- **Load balancer** : Répartit le trafic entre les différentes instances d'un service pour assurer une répartition équilibrée des charges.
  - **Outils associés** : HAProxy, NGINX, AWS Elastic Load Balancer.
- **Discovery** : Permet aux services de localiser dynamiquement d'autres services sans dépendre d'adresses IP statiques.
  - **Outils associés** : Eureka, Consul, Zookeeper.
- **Monitoring :** Un système qui suit la santé et les performances des microservices et génère des alertes en cas de panne ou de dégradation des performances.
  - **Outils associés** : Prometheus, Grafana, ELK Stack.

LES DÉFIS SOULEVÉS

Passer d'une architecture monolithique à l'architecture microservices soulève quelques défis:

- Comment identifier l'application d'entreprise qui s'adapte à cette architecture?
- Comment définir les frontières d'un microservice?
- Comment faire (auto) découvrir ces microservices?
- Comment permettre au client de découvrir ces microservices sans rien connaitre de leur contexte (couplage faible)?
- Comment faire communiquer deux microservices déployés dans des serveurs distincts sans écrire du code supplémentaire et sans l'overhead du http/Rest?
- Comment faire pour que l'application web qui consomme divers microservices puisse paraître comme une application uniforme et homogène?

ARCHITECTURE ET CARACTÉRISTIQUES

Cohésion interne d'un microservice

Un **microservice** est censé bénéficier d'une forte cohésion interne:

→ le périmètre fonctionnel des fonctionnalités qu'il implémente doit être réduit, et essentiellement dédié à une fonctionnalité élémentaire.

ARCHITECTURE ET CARACTÉRISTIQUES

Découplage entre les microservices

- Dans une architecture microservice, le logiciel est décomposé en un ensemble de services hautement indépendants.

- Chaque microservice peut être:
  - Déployé indépendamment
  - Conçu et développé indépendamment
  - Testé indépendamment

- En conséquence, chaque microservice peut évoluer de façon plus indépendante que dans une approche « monolithique ».

ARCHITECTURE ET CARACTÉRISTIQUES

Distribution des microservices

- Afin d'atteindre ce niveau de découplage, chaque microservice doit représenté un **processus système indépendant**, isolé sur une machine distincte ou déployé dans un conteneur.

- La communication entre les clients et les microservices, ou entre les microservices eux-mêmes, est implémentée par des services web ou un protocole d'échanges de messages avec ou sans modèle d'acteurs.

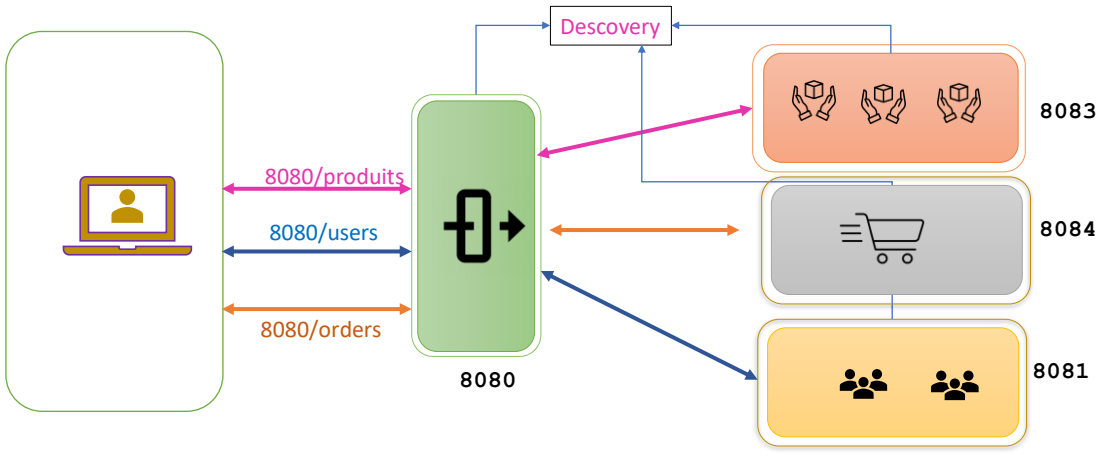
ARCHITECTURE ET CARACTÉRISTIQUES

Distribution des microservices

- Monolithe
  - 1 processus
  - Interaction locale

- Micro-services
  - N-processus distribués
  - Interaction à distance

APPLICATION – GATEWAY CONFIGURATION



APPLICATION – DESCOVERY CONFIGURATION

i. Créer une application Spring boot et ajouter les dépendances suivantes:

ii. Configurer le fichier application.yml

```yaml
server:
  port: 8761
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

iii. Ajouter l'annotation suivante à la classe principale : `@EnableEurekaServer`

APPLICATION – GATEWAY CONFIGURATION

i. Créer une application Spring boot et ajouter les dépendances suivantes:

ii. Configurer le fichier application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: r1
          uri: lb://product-service
          predicates :
            - Path= /produits/**
        - id: r2
          uri: lb://user-service
          predicates:
            - Path= /users/**
  application:
    name: gateway
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
```

APPLICATION – GATEWAY CONFIGURATION

❖ Version 1: Configuration statique

Configuration de Gateway via le fichier application.proporties ou application.yml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id : r1
          uri : lb://user-service
          predicates :
            - Path= /users/**
        - id : r2
          uri : http://product-service
          predicates :
            - Path= /produits/**
    application:
      name: gateway

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
```


APPLICATION – GATEWAY CONFIGURATION

❖ Version 2:

Configuration de Gateway via la classe Application

```java
@SpringBootApplication
public class Tp2GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(Tp2GatewayApplication.class, args);
    }
    @Bean
    public RouteLocator myRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(r->r.path("/products/**").uri("http://localhost:8084/"))
                .route(r->r.path("/etudiants/**").uri("http://localhost:8083/"))
                .build();
    }
}
```

APPLICATION : AJOUT DES MICROSRVICES

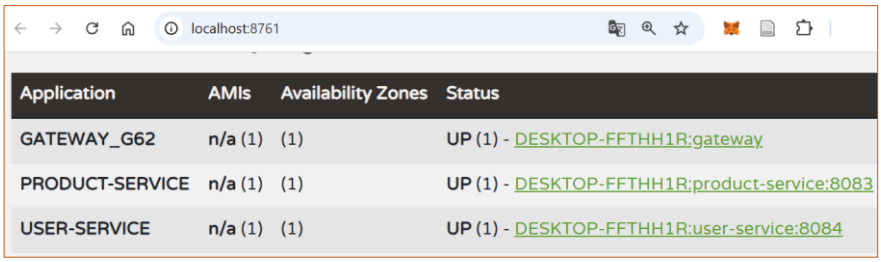
- Ajouter deux microservices avec les dépendances nécessaires y compris : « eureka discovery client »
  - User-service
  - Product-service

```yaml
server:
  port: 8084
spring:
  application:
    name: user-service
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
```

```yaml
server:
  port: 8083
spring:
  application:
    name: product-service
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
```


TESTE

1. Démarrer le descovery
2. Puis démarrer les autres microservices
3. Démarrer le gateway



TESTE

APPLICATION – GATEWAY CONFIGURATION

Test :

http://localhost:8080/etudiants

http://localhost:8080/produits

TESTE

1. Démarrer le descovery
2. Puis démarrer les autres microservices
3. Démarrer le gateway

Instances actuellement enregistrées auprès d'Eureka

Application	AMI	Zones de disponibilité	Statut
GATEWAYSERVER	n/d (1)	(1)	UP (1) - 192.168.56.1 : GatewayServer :8080
SERVICE-CLIENT	n/d (1)	(1)	UP (1) - 192.168.56.1 :service-client :8082
SERVICE-PRODUIT	N/A (2)	(2)	UP (2) - 192.168.56.1 :service-produit :8085 , 192.168.56.1 :service-produit :8081