

# ARCHITECTURE MICROSERVICES

# PLAN

- Introduction
- Définition
- Architecture micro service
- Mise en œuvre avec springboot
- Tps

# Introduction

- Avec l'essor de l'utilisation des applications gourmandes en ressources, les entreprises sont confrontées à des challenges liés à la performance de ces services, au coût de l'infrastructure technique et au coût du développement et de la maintenance.
- **Problème n° 1** : comment faire en sorte que les applications proposées en ligne soient **toujours disponibles** et ne souffrent jamais de coupure ou de ralentissement, quelle que soit l'affluence des utilisateurs sur celles-ci ?



# Introduction

- il faut que l'application soit conçue de façon à ce qu'elle puisse être **scalable**, c'est-à-dire **capable de s'étendre sur plus de ressources** (plus de serveurs, de disques durs, de bases de données)
  - **Problème n°2** : les entreprises se livrent à une "guerre de la mise à jour". Il faut que l'entreprise soit capable de faire évoluer son application de façon très fréquente et de répondre rapidement aux nouvelles fonctionnalités que propose la concurrence.

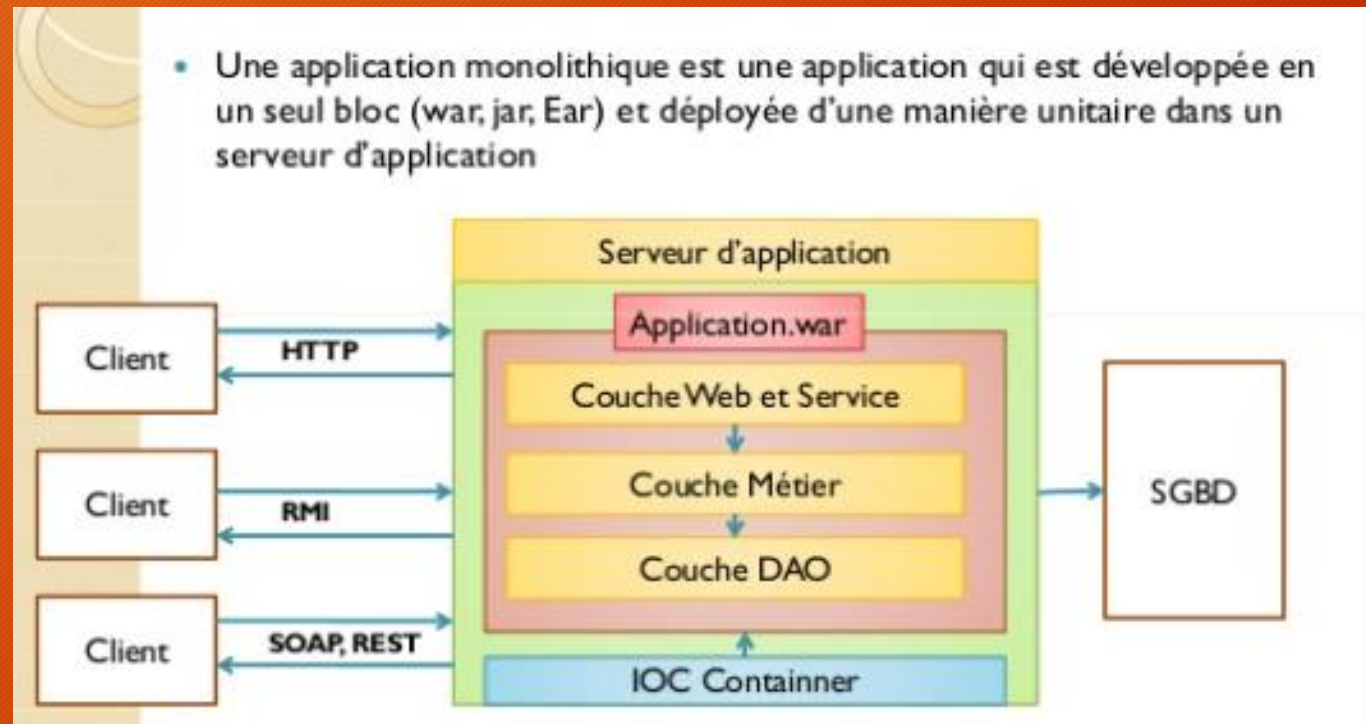
# Introduction

- Les entreprises sont donc obligées de passer à des architectures qui permettent de faire des mises à jour qui visent des **composants ciblés** , très rapidement, de façon fiable et sans se soucier des éventuelles conséquences sur le reste de l'application.
  - **Problème n°3** : Les technologies utilisées pour développer ces applications **évoluent très vite** et les nouveautés offrent parfois des avantages énormes. Comment les entreprises peuvent-elle s'adapter rapidement pour tirer profit de ces évolutions ?



# Introduction

- Architecture monolithique



# Introduction

## Problèmes des applications monolithiques

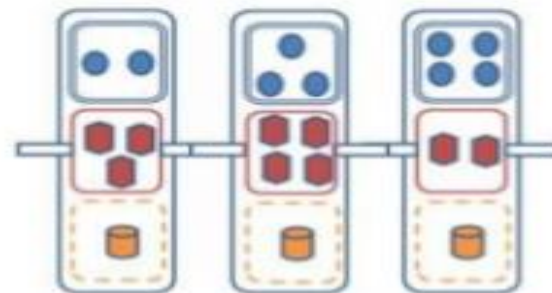
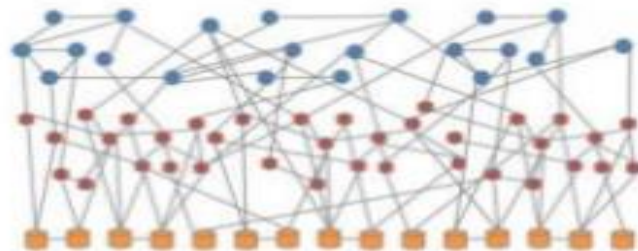
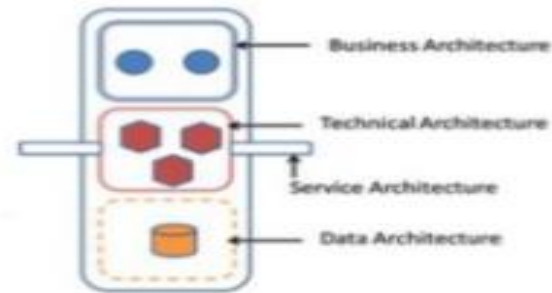
- Les principaux problèmes des applications monolithiques sont :
  - Elles centralisent tous les besoins fonctionnels
  - Elles sont réalisées dans une seule technologie.
  - Chaque modification nécessite de :
    - Tester les régressions
    - Redéployer toute l'application
  - Difficile à faire évoluer au niveau fonctionnel
  - Livraison en bloc (Le client attend beaucoup de temps pour commencer à voir les premières versions )

# Définition

- Architecture micro service =

## Micro service

- Un micro service combine les trois couches « **métier**, **technique** et **données** » en une seule, accessible via des API.



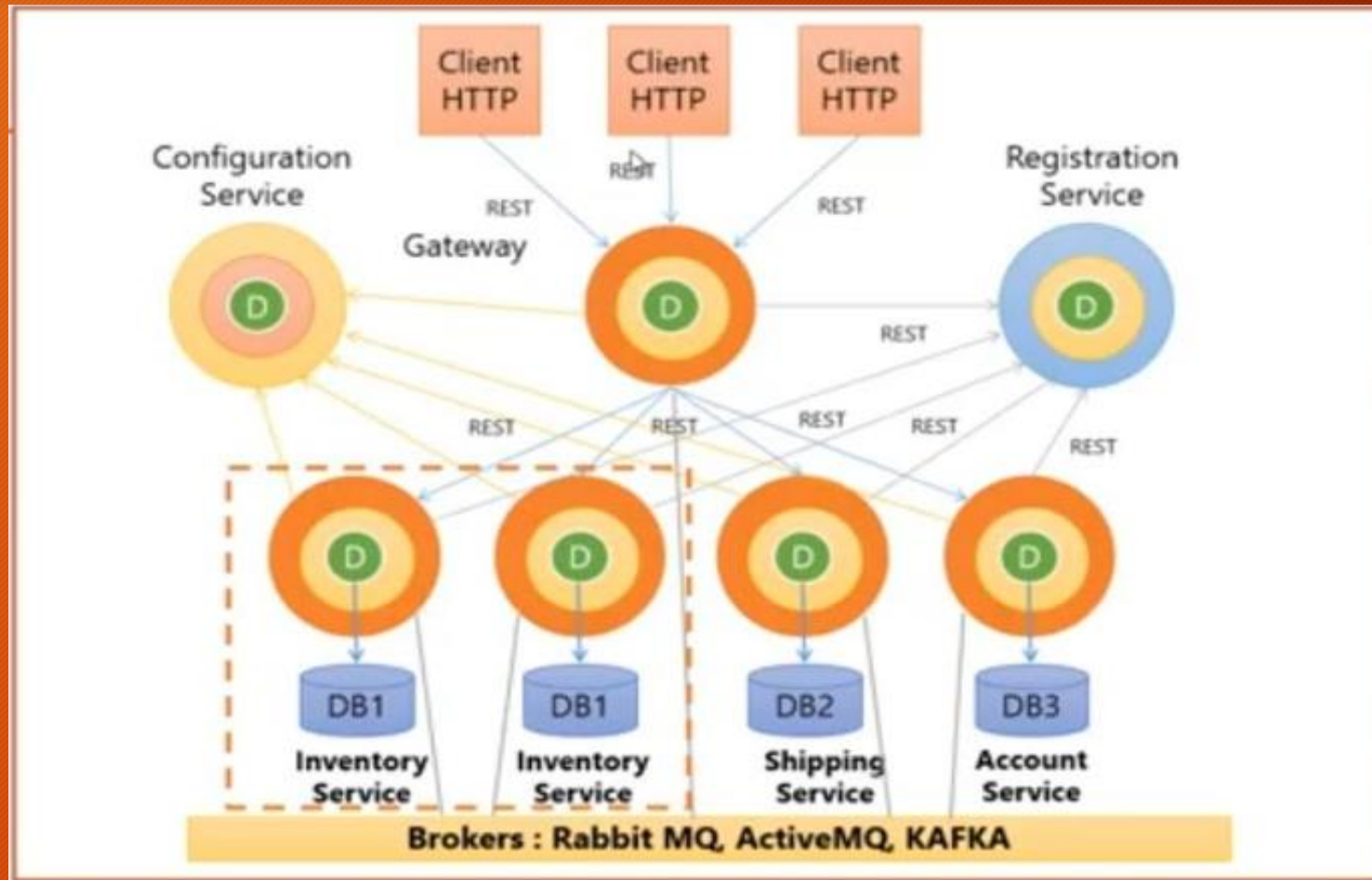


# Définition

## Aspects clef des micro services

- Chaque micro service peut être conçu à l'aide de n'importe quel **outil** et développé avec n'importe quel **langage et technologie**.
- Ils sont **faiblement couplés** puisque chaque micro service est physiquement séparé des autres,
- **Indépendance relative entre les différentes équipes** qui développent les différents micro services (en partant du principe que les APIs qu'ils exposent sont définis à l'avance).
- **Facilité des tests** et du **déploiement** ou de la livraison continue.

# Architecture micro service





# Architecture micro service

- Les différents types de micro service:
  - service métier: découpage de la problématique métier en plusieurs mini projets
  - Registration service: enregistrement des micro services dans un annuaire
  - Configuration service: centralisation de la configuration
  - Service Proxy / Service Gateway: centralisation des entrées clientes
  - Monitoring service : service de surveillance des micro services
  - External service : intégration de services externes



# Architecture micro service

- Chaque micro service est déterminé par:
  - Son mon
  - Son adresse IP
  - Son numéro de port

```
Customer-service : CustomerServiceApplication.java

package org.id.customerservice;
import lombok.AllArgsConstructor; import lombok.Data; import lombok.NoArgsConstructor; import lombok.ToString; import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.context.annotation.Bean;
import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.rest.core.annotation.RepositoryRestResource; import javax.persistence.Entity;
import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id;

@Entity @Data @NoArgsConstructor @AllArgsConstructor @ToString
class Customer {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; private String name; private String email;
}

@RepositoryRestResource
interface CustomerRepository extends JpaRepository<Customer, Long> { }

@SpringBootApplication
public class CustomerServiceApplication {

    public static void main(String[] args) { SpringApplication.run(CustomerServiceApplication.class, args); }

    @Bean
    CommandLineRunner start(CustomerRepository customerRepository){
        return args -> {
            customerRepository.save(new Customer(null, "Enset", "contact@enset-media.ma"));
            customerRepository.save(new Customer(null, "FSTM", "contact@fstm.ma"));
            customerRepository.save(new Customer(null, "ENSAM", "contact@ensam.ma"));
            customerRepository.findAll().forEach(System.out::println);
        };
    }
}
```

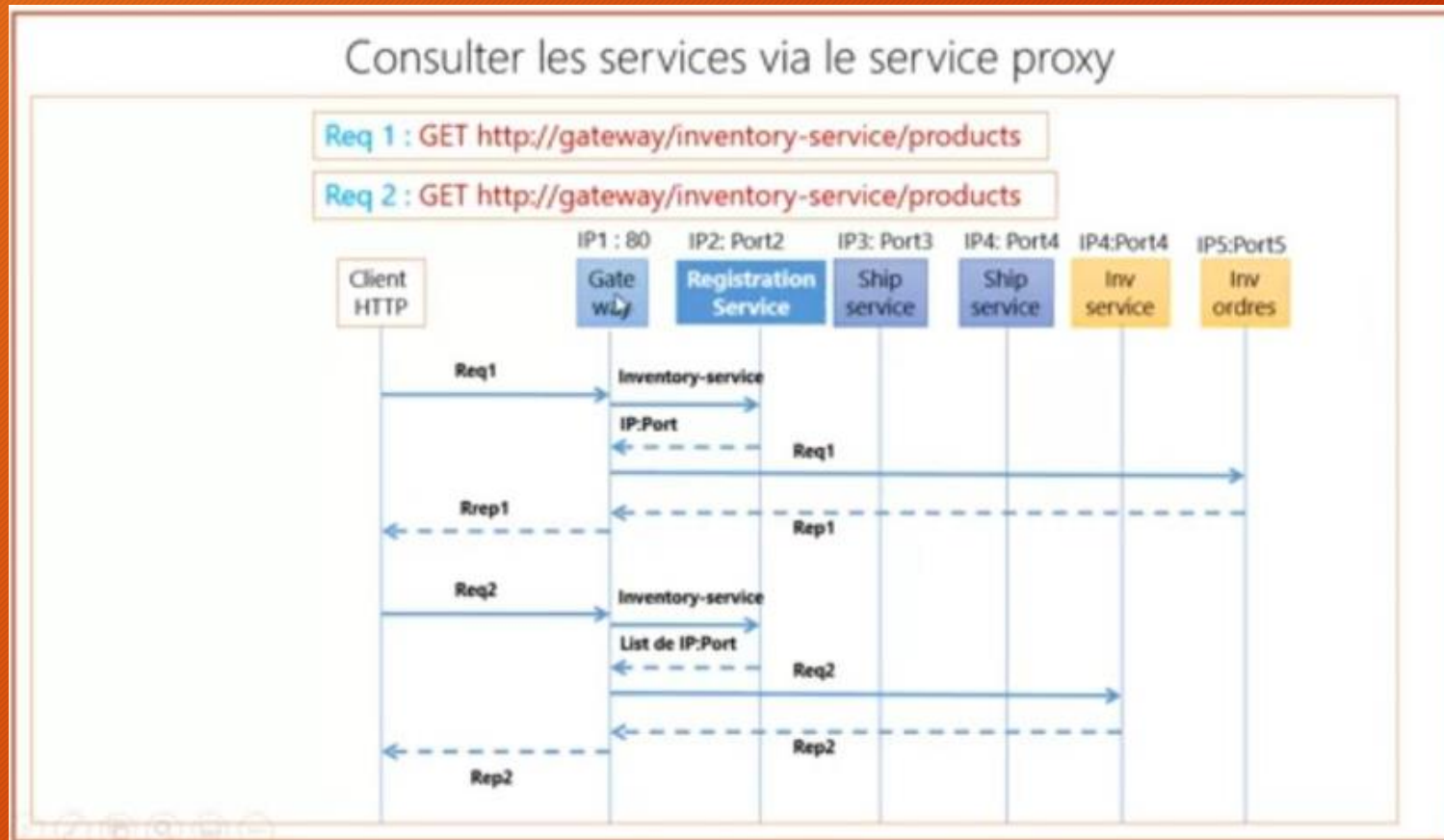
**application.properties**

```
spring.cloud.discovery.enabled=false
server.port=8081
spring.application.name=customer-service
management.endpoints.web.exposure.include=*
```

**@Projection(name = "fullCustomer", types = Customer.class)**

```
interface CustomerProjection extends Projection {
    public Long getId();
    public String getName();
    public String getEmail();
}
```

# Architecture micro service





# Mise en œuvre avec springboot

## Application

- Créer une application basée sur deux services métiers:
  - Service des clients
  - Service d'inventaire
  - Service Facturation
  - Services Externes : RapidAPI
- L'orchestration des services se fait via les services techniques de Spring Cloud :
  - Spring Cloud Gateway Service comme service proxy
  - Registry Eureka Service comme annuaire d'enregistrement et de découverte des services de l'architecture
  - Hystrix Circuit Breaker
  - Hystrix DashBoard



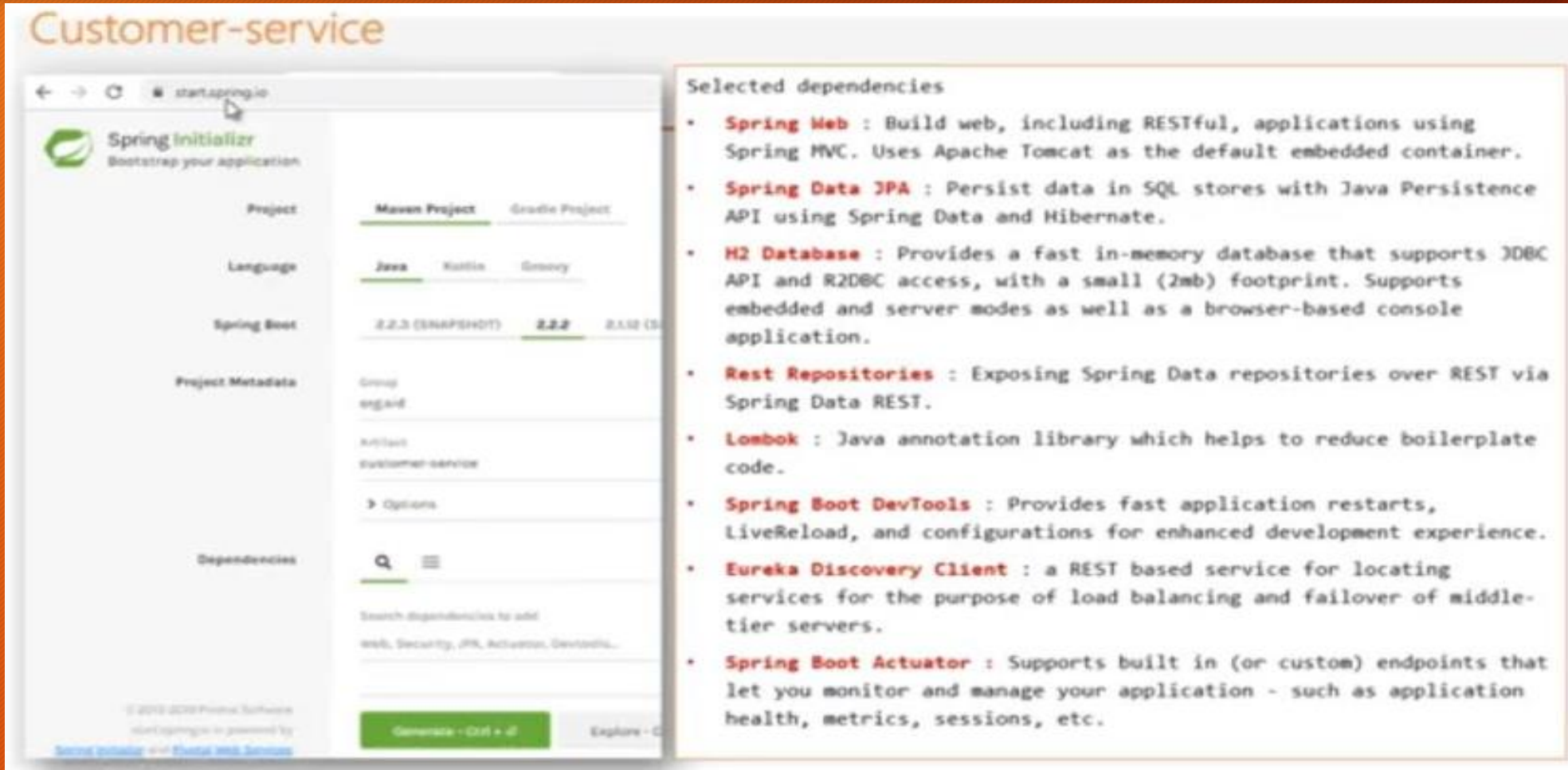


# Mise en œuvre avec springboot

- service métier: projet springboot java, avec springData ou SpringDataRest
- Registration service: Eureka
- Service Proxy / Service Gateway: Spring Cloud
- Monitoring service : Hystrix dashboard
- External service : rapidAPI

# Mise en œuvre avec springboot : service métier customer

## Customer-service



← → ↻ start.spring.io

**Spring Initializer**  
Bootstrap your application

**Project**

Maven Project    Gradle Project

**Language**

Java    Kotlin    Groovy

**Spring Boot**

2.2.3 (SNAPSHOT)    **2.2.2**    2.1.10 (LTS)

**Project Metadata**

Group  
org.aid

Artifact  
customer-service

Options

**Dependencies**

Search dependencies to add  
Web, Security, JPA, Actuator, DevTools...

Generate - Ctrl + G    Explore - C

© 2012-2019 Pivotal Software  
All rights reserved. Powered by  
Spring Initializer and Pivotal Web Services

### Selected dependencies

- **Spring Web** : Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- **Spring Data JPA** : Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- **H2 Database** : Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser-based console application.
- **Rest Repositories** : Exposing Spring Data repositories over REST via Spring Data REST.
- **Lombok** : Java annotation library which helps to reduce boilerplate code.
- **Spring Boot DevTools** : Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- **Eureka Discovery Client** : a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
- **Spring Boot Actuator** : Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.



# Mise en œuvre avec springboot : service métier customer

## Customer-service : CustomerServiceApplication.java

```
package org.id.customerservice;
import lombok.AllArgsConstructor; import lombok.Data; import lombok.NoArgsConstructor; import lombok.ToString; import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.context.annotation.Bean;
import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.rest.core.annotation.RepositoryRestResource; import javax.persistence.Entity;
import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id;

@Entity @Data @NoArgsConstructor @AllArgsConstructor @ToString
class Customer {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; private String name; private String email;
}

@RepositoryRestResource
interface CustomerRepository extends JpaRepository<Customer, Long> { }

@SpringBootApplication
public class CustomerServiceApplication {

    public static void main(String[] args) { SpringApplication.run(CustomerServiceApplication.class, args); }

    @Bean
    CommandLineRunner start(CustomerRepository customerRepository){
        return args -> {
            customerRepository.save(new Customer(null, "Enset", "contact@enset-media.ma"));
            customerRepository.save(new Customer(null, "FSTM", "contact@fstm.ma"));
            customerRepository.save(new Customer(null, "ENSAM", "contact@ensam.ma"));
            customerRepository.findAll().forEach(System.out::println);
        };
    }
}
```

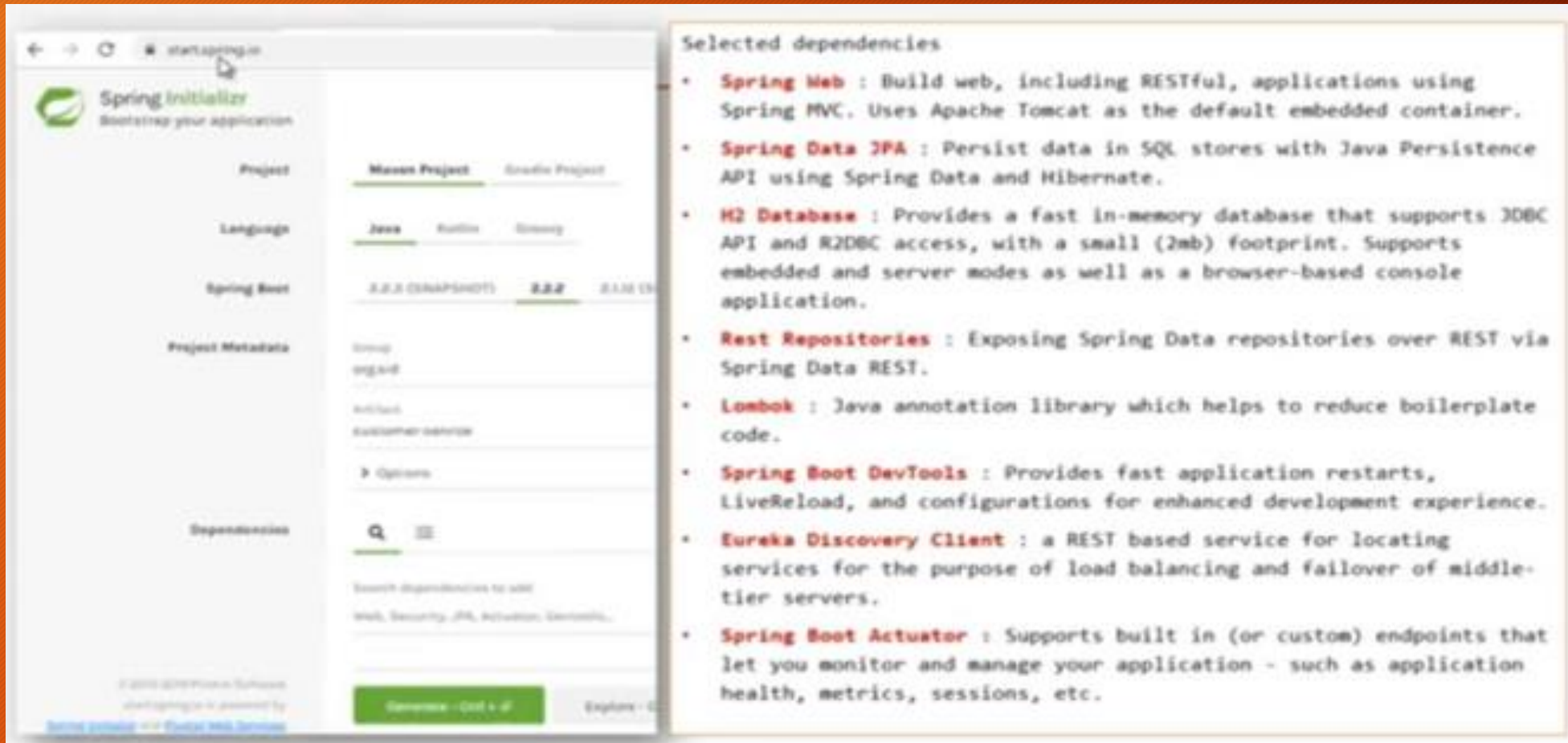
### application.properties

```
spring.cloud.discovery.enabled=false
server.port=8081
spring.application.name=customer-service
#management.endpoints.web.exposure.include=*
```

```
@Projection(name = "fullCustomer", types =
Customer.class)
interface CustomerProjection extends Projection {
    public Long getId();
    public String getName();
    public String getEmail();
}
```



# Mise en œuvre avec springboot : service métier inventory



The screenshot displays the Spring Initializr web application interface. On the left, a sidebar contains navigation links: Project, Language, Spring Boot, Project Metadata, and Dependencies. The main content area is divided into two sections. The top section, titled 'Project', shows configuration options for 'Maven Project' and 'Gradle Project', with 'Maven Project' selected. Below this, the 'Language' section shows 'Java' selected. The 'Spring Boot' section shows version '2.2.2 (SNAPSHOT)' selected. The 'Project Metadata' section shows 'Group' as 'org.springframework' and 'Artifact' as 'spring-web'. The 'Dependencies' section shows a search bar and a list of dependencies. On the right, a box titled 'Selected dependencies' lists the following:

- **Spring Web** : Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- **Spring Data JPA** : Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- **H2 Database** : Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser-based console application.
- **Rest Repositories** : Exposing Spring Data repositories over REST via Spring Data REST.
- **Lombok** : Java annotation library which helps to reduce boilerplate code.
- **Spring Boot DevTools** : Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- **Eureka Discovery Client** : a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
- **Spring Boot Actuator** : Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

# Mise en œuvre avec springboot : service métier inventory

```
package com.huios.springboot;

import org.springframework.beans.factory.annotation.Autowired;

@SpringBootApplication
public class ProductServiceApplication {
    @Autowired
    private RepositoryRestConfiguration restConfiguration;

    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }

    @Bean
    CommandLineRunner start(ProductRepository productRepository) {
        restConfiguration.exposeIdsFor(Product.class);
        return args -> {
            productRepository.save(new Product(null, "ordinateur", 350));
            productRepository.save(new Product(null, "souris", 8));
            productRepository.save(new Product(null, "clavier", 15));
            productRepository.findAll().forEach(System.out::println);
        };
    }
}
```

```
package com.huios.springboot.dao;

import org.springframework.data.jpa.repository.JpaRepository;

@RepositoryRestResource
public interface ProductRepository extends JpaRepository<Product, Long>{

}
```

```
package com.huios.springboot.domaine;


import javax.persistence.Entity;

@Entity
@Data @NoArgsConstructor @AllArgsConstructor @ToString
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double price;
}
```

```
server.port=8082
spring.application.name=inventory-service
spring.cloud.discovery.enabled=false
```

# Mise en œuvre avec springboot : service gateway

## Gateway-service



Spring Initializr  
Bootstrap your application

Project	Maven Project	Gradle Project
Language	Java	Kotlin Groovy
Spring Boot	2.2.3 (SNAPSHOT)	2.2.2
Project Metadata	<div>Group org.id</div> <div>Artifact gateway-service</div> <div>&gt; Options</div>	
Dependencies	<div>Q</div> <div>≡</div>	

### Selected dependencies

- **Gateway** : Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency.
- **Spring Boot Actuator** : Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- **Hystrix** : Circuit breaker with Spring Cloud Netflix Hystrix.
- **Eureka Discovery Client** : a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

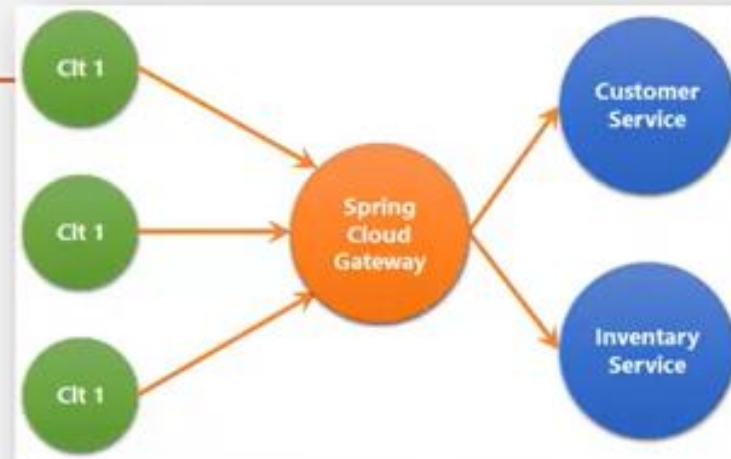


# Mise en œuvre avec springboot : service gateway

## Static routes configuration: application.yml

application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id : r1
          uri : http://localhost:8081/
          predicates :
            - Path= /customers/**
        - id : r2
          uri : http://localhost:8082/
          predicates :
            - Path= /products/**
      discovery:
        enabled: false
  server:
    port: 8888
```



localhost:8888/customers/1

```
{
  "name": "Enset",
  "email": "contact@enset-media.ma",
  "_links": {
    "self": {
      "href": "http://localhost:8881/customers/1"
    },
    "customer": {
      "href": "http://localhost:8081/customers/1"
    }
  }
}
```

localhost:8888/products/1

```
{
  "name": "Computer Desk Top HP",
  "price": 900,
  "_links": {
    "self": {
      "href": "http://localhost:8082/products/1"
    },
    "product": {
      "href": "http://localhost:8082/products/1"
    }
  }
}
```

# Mise en œuvre avec springboot : service gateway

## Static routes configuration: Java Config Class

```
@Bean
RouteLocator gatewayRoutes(RouteLocatorBuilder builder){
    return builder.routes()
        .route(r->r.path("/customers/**").uri("http://localhost:8081/").id("r1"))
        .route(r->r.path("/products/**").uri("http://localhost:8082/").id("r2"))
        .build();
}
```

localhost:8088/products/1

```
{
  "name": "Computer Desk Top HP",
  "price": 900,
  "_links": {
    "self": {
      "href": "http://localhost:8082/products/1"
    },
    "product": {
      "href": "http://localhost:8082/products/1"
    }
  }
}
```

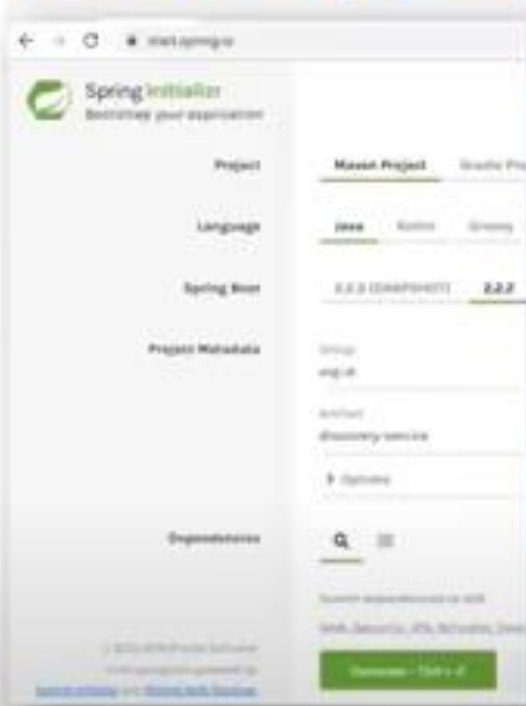
localhost:8088/customers/1

```
{
  "name": "Enset",
  "email": "contact@enset-media.ma",
  "_links": {
    "self": {
      "href": "http://localhost:8081/customers/1"
    },
    "customer": {
      "href": "http://localhost:8081/customers/1"
    }
  }
}
```




# Mise en œuvre avec springboot : service d'enregistrement eureka

## Eureka Discovery Service : Dynamic Routing



Selected dependencies

- **Eureka Server** : `spring-cloud-netflix Eureka Server`.



```
package org.id.discovery.service; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer; @SpringBootApplication @EnableEurekaServer public class DiscoveryServiceApplication { public static void main(String[] args) { SpringApplication.run(DiscoveryServiceApplication.class, args); } }
```

**application.properties**

```
server.port=8761  
eureka.client.fetch-registry=false  
eureka.client.register-with-eureka=false
```



# Mise en œuvre avec springboot : autoriser aux services de s'enregistrer chez Eureka

Permettre à Customer-service et Invotory-service de s'enregistrer chez Eureka server

## Customer-service

```
spring.cloud.discovery.enabled=true
server.port=8081
spring.application.name=customer-service
management.endpoints.web.exposure.include=*
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

### application.properties

```
spring.application.name=gateway-service
server.port=8888
spring.cloud.discovery.enabled=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

## Inventory-service

```
spring.cloud.discovery.enabled=true
server.port=8082
spring.application.name=inventory-service
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

### application.properties

# Mise en œuvre avec springboot : modification de la gateway statique

```
@Bean
RouteLocator staticRoutes(RouteLocatorBuilder builder){
    return builder.routes()
        .route(r->r.path("/customers/**").uri("lb://CUSTOMER-SERVICE").id("r1"))
        .route(r->r.path("/products/**").uri("lb://INVENTORY-SERVICE").id("r2"))
        .build();
}
```



# Mise en œuvre avec springboot : modification de la gateway dynamique

## Dynamic routes configuration with Discovery Service

### application.properties

```
spring.application.name=gateway-service
spring.cloud.discovery.enabled=true
server.port=8888
```

@Bean

```
DiscoveryClientRouteDefinitionLocator dynamicRoutes(ReactiveDiscoveryClient rdc,
DiscoveryLocatorProperties dlp){
    return new DiscoveryClientRouteDefinitionLocator(rdc,dlp);
}
```

localhost:8888/CUSTOMER-SERVICE/customers/1

```
{
  "name": "Enset",
  "email": "contact@enset-media.ma",
  "_links": {
    "self": {
      "href": "http://localhost:8081/customers/1"
    },
    "customer": {
      "href": "http://localhost:8081/customers/1"
    }
  }
}
```

localhost:8888/INVENTORY-SERVICE/products/1

```
{
  "name": "Computer Desk Top HP",
  "price": 900,
  "_links": {
    "self": {
      "href": "http://localhost:8082/products/1"
    },
    "product": {
      "href": "http://localhost:8082/products/1"
    }
  }
}
```

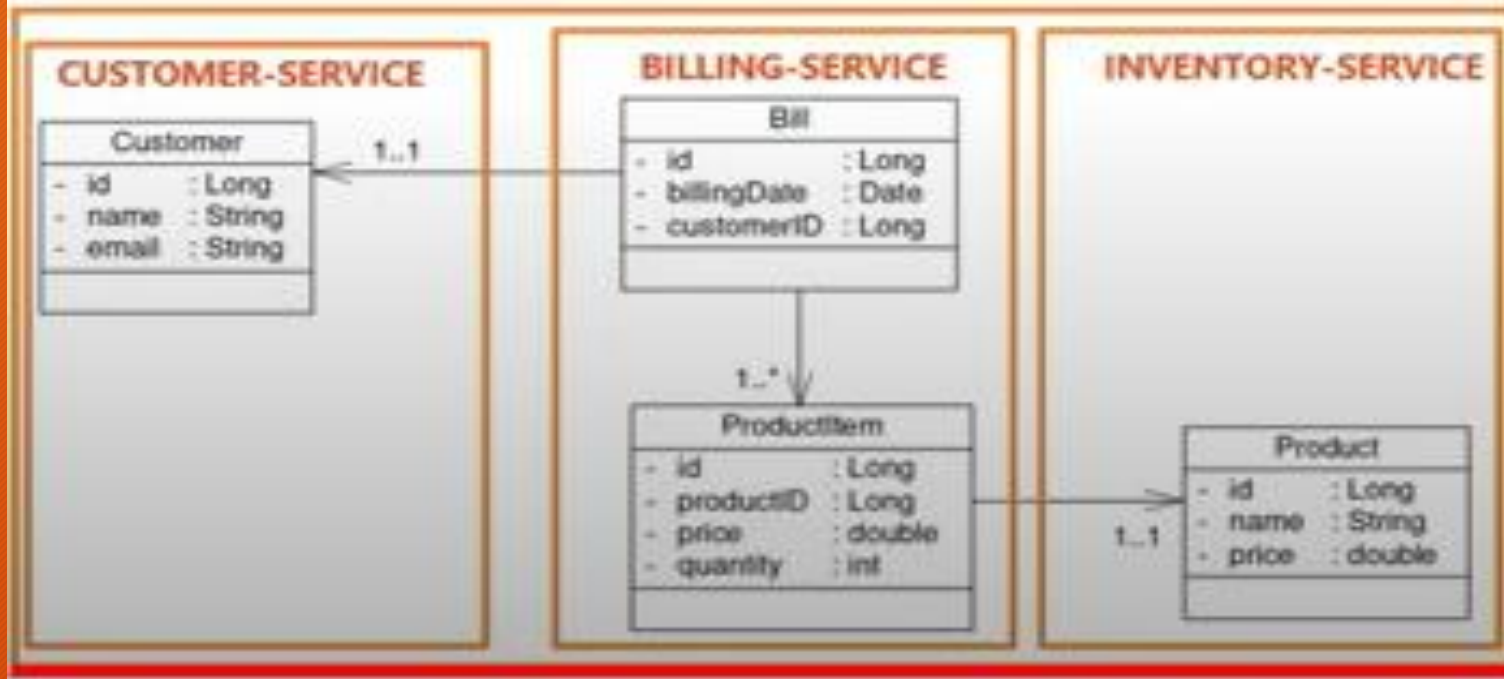




# Mise en œuvre avec springboot : cas de communication entre micro services

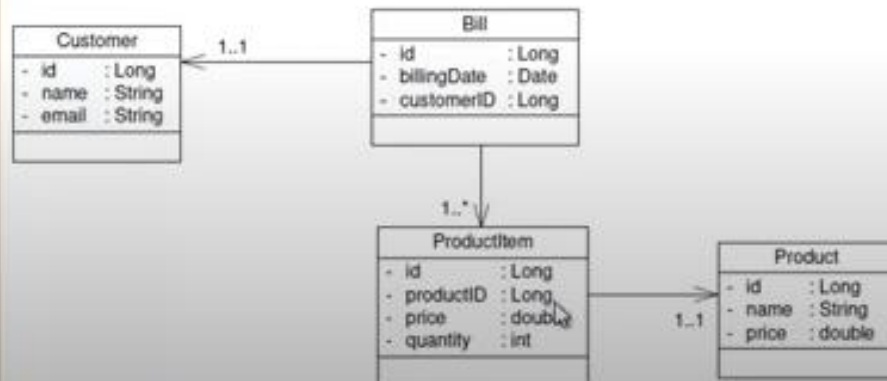


Ajouter un service de facturation (Billing Service), qui communique avec les services Clients et Inventaire en utilisant Spring cloud OpenFeign Rest Client



# Mise en œuvre avec springboot : cas de communication entre micro services

```
@Entity @Data @NoArgsConstructor @AllArgsConstructor
class Bill{
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; private Date billingDate;
    @Transient @OneToMany(mappedBy = "bill")
    private Collection<ProductItem> productItems;
    @Transient private Customer customer;
    private long customerID;
}
@RepositoryRestResource
interface BillRepository extends JpaRepository<Bill,Long>{}
```



## Billing-service

```
@Entity @Data @NoArgsConstructor @AllArgsConstructor
class ProductItem{
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Transient
    private Product product; private long productID;
    private double price; private double quantity;
    @ManyToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Bill bill;
}
@RepositoryRestResource
interface ProductItemRepository extends
JpaRepository<ProductItem,Long>{
    List<ProductItem> findByBillId(Long billID);
}
```





# Mise en œuvre avec springboot : cas de communication entre micro services

## Selected dependencies

- **Spring Web** : Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- **Spring Data JPA** : Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- **H2 Database** : Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- **Rest Repositories** : Exposing Spring Data repositories over REST via Spring Data REST.
- **Lombok** : Java annotation library which helps to reduce boilerplate code.
- **Spring Boot DevTools** : Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- **Eureka Discovery Client** : a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
- **OpenFeign** : Declarative REST Client. OpenFeign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations.
- **Spring HATEOAS** : Eases the creation of RESTful APIs that follow HATEOAS principle when working with Spring / Spring MVC.



# Mise en œuvre avec springboot : cas de communication entre micro services

```
spring.application.name=billing-service
server.port=8083
spring.cloud.discovery.enabled=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

```
@Data
public class Customer {
    private Long id; private String name; private String email;
}
```

```
import org.springframework.data.jpa.repository.JpaRepository;
@RepositoryRestResource
public interface ProductItemRepository extends JpaRepository<ProductItem, Long> {
}
```

```
@Data
public class Product {
    private Long id; private String name; private double price;
}
```

```
@Projection(name="fullBill", types = Bill.class)
public interface BillProjection {
    public Long getId();
    public Date getBillingDate();
    public Long getCustomerID();
    public Collection<ProductItem> getProductItems();
}
```

```
import org.springframework.data.jpa.repository.JpaRepository;
@RepositoryRestResource
public interface BillRepository extends JpaRepository<Bill, Long>{
}
```

# Mise en œuvre avec springboot : cas de communication entre micro services

```
@SpringBootApplication
@EnableFeignClients
public class Application {
    @Autowired
    private RepositoryRestConfiguration restConfiguration;
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CommandLineRunner start (BillRepository billRepository,
        ProductItemRepository productItemRepository,
        CustomerService customerService,
        InventoryService inventoryService)
    {
        restConfiguration.exposeIdsFor(Bill.class, ProductItem.class);
        return args -> {
            Customer c1 = customerService.findCustomerById(1L);
            System.out.println("*****");
            System.out.println("ID="+c1.getId());
            System.out.println("Name="+c1.getName());
            System.out.println("Email="+c1.getEmail());
            System.out.println("*****");
        }
    }
}
```

```
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class Bill {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date billingDate;
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Long customerID;
    @Transient
    private Customer customer;
    @OneToMany(mappedBy = "bill")
    private Collection<ProductItem> productItems;
}
```

```
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class ProductItem {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Long productId;
    @Transient
    private Product product;
    private double price;
    private int quantity;
    @ManyToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Bill bill;
}
```

# Mise en œuvre avec springboot : cas de communication entre micro services

```
@RestController
public class BillRestController {
    // accès repository local
    @Autowired
    private BillRepository billRepository;
    // accès repository local
    @Autowired
    private ProductItemRepository productItemRepository;
    // accès service distant
    @Autowired
    private CustomerService customerService;
    // accès service distant
    @Autowired
    private InventoryService inventoryService;
    @GetMapping("/fullBill/{id}")
    public Bill getBill(@PathVariable(name="id")Long id) {
        Bill bill=billRepository.findById(id).get();
        bill.setCustomer(customerService.findCustomerById(bill.getCustomerID()));
        bill.getProductItems().forEach(pi->{
            pi.setProduct(inventoryService.findProductById(pi.getProductId()));
        });
        return bill;
    }
}
```

```
@FeignClient(name="CUSTOMER-SERVICE")
public interface CustomerService {
    @GetMapping("/customers/{id}")
    public Customer findCustomerById(@PathVariable(name="id")Long id);
}
```

```
@FeignClient(name="INVENTORY-SERVICE")
public interface InventoryService {
    @GetMapping("/products/{id}")
    public Product findProductById(@PathVariable(name="id")Long id);
    @GetMapping("/products")
    public PagedModel<Product> findAllProducts();
}
```



# Mise en œuvre avec springboot : intégration service distant (dans le service gateway)

```
@Bean
RouteLocator staticRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(r -> r
            .path("/publicCountries/**")
            .filters(f->f
                .addRequestHeader("x-rapidapi-host", "nanosdk-countries-v1.p.rapidapi.com")
                .addRequestHeader("x-rapidapi-key", "b2753199f7mshf443b8c55977e27p190bafjsn903e058e9245")
                .rewritePath("/publicCountries/(?<segment>.*)", "/${segment}")
                .hystrix(h->h.setName("countries").setFallbackUri("forward:/defaultCountries"))
            )
            .uri("https://nanosdk-countries-v1.p.rapidapi.com/countries").id("r1"))
        .build();
}
```

# Mise en œuvre avec springboot : monitoring

```
spring.application.name=gateway-service
server.port=8888
spring.cloud.discovery.enabled=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
management.endpoints.web.exposure.include=hystrix.stream
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=1000
```

```
@SpringBootApplication
@EnableHystrix
public class GatewayServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(GatewayServiceApplication.class, args);
    }
}
```

localhost:8888/actuator/hystrix.stream

data:{"type":"ping"}

data:

```
{"type":"HystrixCommand","name":"countries","group":"HystrixGatewayFilterFactory","currentTime":1592201460668,"isCircuitBreakerOpen":true,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmitted":0,"rollingCountExceptionsThrown":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountOpen":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentRequests":0,"latencyExecute_mean":0,"latencyExecute":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"latencyTotal_mean":0,"latencyTotal":{"0":0,"25":0,"50":0,"75":0,"90":0,"95":0,"99":0,"99.5":0,"100":0},"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_executionIsolationStrategy":"SEMAPHORE","propertyValue_executionIsolationThreadTimeoutInMilliseconds":2000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"HystrixGatewayFilterFactory"}
```

data:{"type":"ping"}



# Mise en œuvre avec springboot : monitoring



## Hystrix Dashboard

<http://localhost:8888/actuator/hystrix.stream>

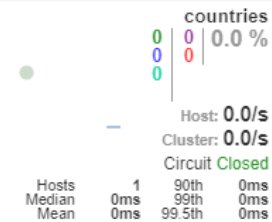
Cluster via Turbine (default cluster): <https://turbine-hostname:port/turbine.stream>  
Cluster via Turbine (custom cluster): [https://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](https://turbine-hostname:port/turbine.stream?cluster=[clusterName])  
Single Hystrix App: <https://hystrix-app:port/actuator/hystrix.stream>

Delay:  ms Title:

[Monitor Stream](#)

## Hystrix Stream: <http://localhost:8888/actuator/hystrix.stream>

**Circuit** Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)



**Thread Pools** Sort: [Alphabetical](#) | [Volume](#) |

# CENTRALISATION DE LA CONFIGURATION

- Ce micro service aura pour rôle de centraliser la configuration
- Il devra donc avoir pour dépendance maven « `spring.cloud.config.server` » et l'annotation `@EnableDiscoveryServer`
- Tous les autres micro-services viendront chercher leur configuration dans ce micro service
- Chaque micro service aura pour fichier de configuration `bootstrap.properties`
- Le micro service de configuration va se référer à un répertoire (ici `my-config`)
- Dans ce repertoire on mettra un fichier par micro service et chaque fichier aura pour nom `nom-du-microservice.properties`
- C'est donc le micro service qui est démarré en premier lorsqu'il existe









# CENTRALISATION DE LA CONFIGURATION

```
server.port=8888
spring.cloud.config.server.git.uri = file://${user.home}/my-config
```

```
<!-- permet de rendre ce microservice serveur de configuration -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
</dependencies>
```

	admin-service.properties	21/06/2019 15:46	Fichier PROPERTIES	1 Ko
	application.properties	20/06/2019 17:25	Fichier PROPERTIES	1 Ko
	commande-service.properties	21/06/2019 17:39	Fichier PROPERTIES	1 Ko
	eureka-service.properties	20/06/2019 18:48	Fichier PROPERTIES	1 Ko
	paiement-service.properties	21/06/2019 17:39	Fichier PROPERTIES	1 Ko
	produit-service.properties	21/06/2019 16:24	Fichier PROPERTIES	1 Ko

```
@EnableConfigServer
@SpringBootApplication
public class MicroserviceConfigurationApplication {

    public static void main(String[] args) {
        SpringApplication.run(MicroserviceConfigurationApplication.class, args);
    }
}
```

# CENTRALISATION DE LA CONFIGURATION

- Chaque micro service devra avoir la dépendance `spring.cloud.config.client` et l'annotation `@EnableDiscoveryClient`

```
@EnableDiscoveryClient
@EnableEurekaClient
@SpringBootApplication
public class MicroservicePaieementApplication {

    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(MicroservicePaieementApplication.class, args);

        PaieementRepository paieementRepository = ctx.getBean(PaieementRepository.class);
        /*paieementRepository.save(new Paieement(1,500,578941123));
        paieementRepository.save(new Paieement(2,354,578941123));
        paieementRepository.save(new Paieement(3,8974,47894521));
        paieementRepository.save(new Paieement(4,150,487556666));*/

        List<Paieement> paiements = paieementRepository.findAll();
        for (Paieement paieement : paiements) {
            System.out.println(paieement);
        }
    }
}
```



# CENTRALISATION DE LA CONFIGURATION

- Dans les autres micro services on enlève application.properties et on mets un fichier bootstrap.properties
- Spring.cloud.config.url est le chemin vers le micro service de configuration (ici il est dans le port 8888)

```
spring.application.name=paielement-service
spring.cloud.config.uri = http://localhost:8888

#ajout de l'administration
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
eureka.instance.health-check-url-path= /actuator/health
spring.boot.admin.client.url=http://localhost:9105

info.app.version=1.0-Beta
```

# Conclusion

- MERCI POUR VOTRE ATTENTION