

# TP3- Application: Architecture micro service avec Spring Cloud

## TP3 - Application: Architecture micro service avec Spring Cloud

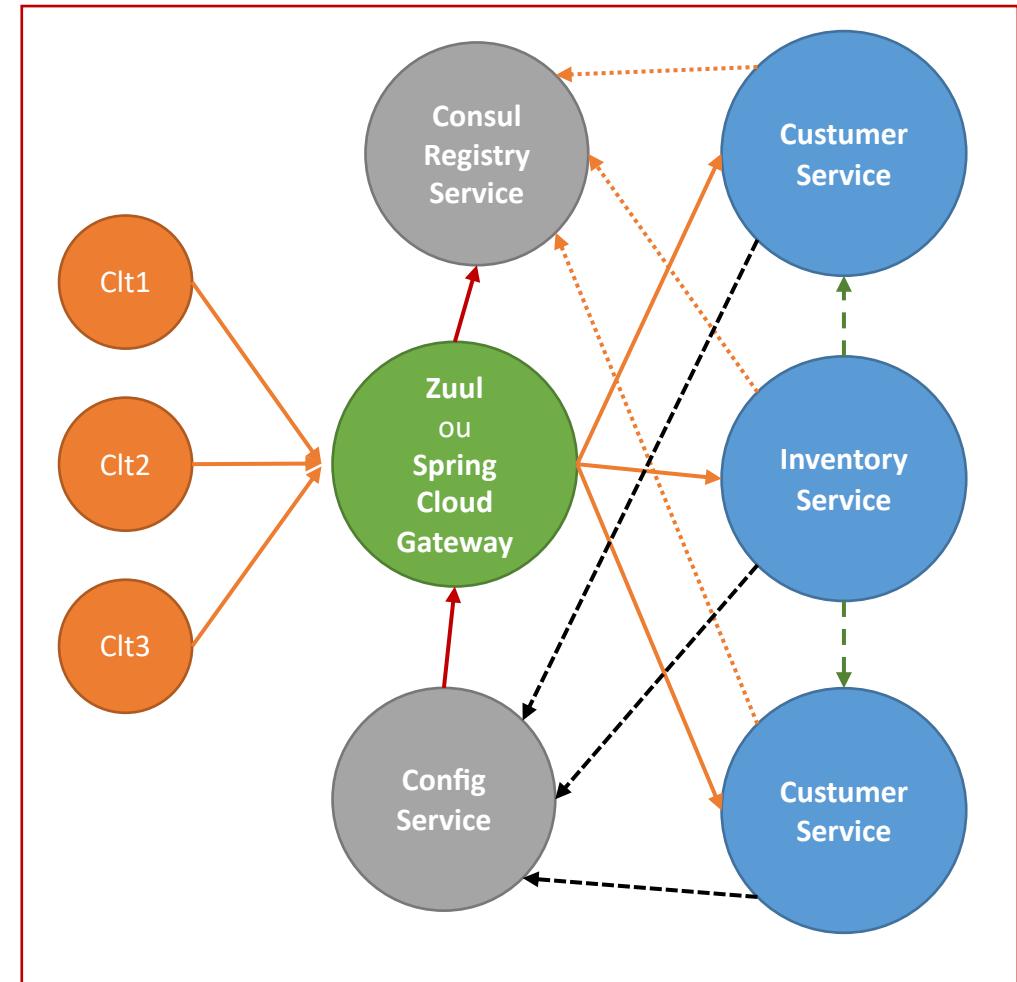
Créer une application basée sur 3 services métiers:

- Service de gestion des clients
- Service de gestion de d'inventaire
- Service de gersion des commandes

L'orchestration des services se fait via les services

techniques de Spring Cloud :

- Spring Cloud Gateway Service comme service proxy
- Consul Registry Service
- Resilience4j : Circuit Breaker; ..



## Service Discovery and Registration (Eureka, Consul)

---

La découverte de services et l'enregistrement sont des composantes essentielles dans une architecture de microservices pour permettre une communication dynamique et efficace entre les services. La découverte de services permet aux services de se localiser et de communiquer entre eux sans dépendances codées en dur. Deux solutions populaires de découverte de services et d'enregistrement sont **Eureka** et **Consul**.

# Service Discovery and Registration (Eureka, Consul)

---

## Eureka

Eureka, développé par Netflix, est un outil de découverte de services qui permet aux services de s'enregistrer et de découvrir d'autres services au sein d'un écosystème de microservices. Il suit une architecture client-serveur avec les composants clés suivants :

- **Eureka Server:** Il fonctionne comme un registre central où les services peuvent s'enregistrer et fournir leurs métadonnées. Le serveur Eureka maintient un registre de tous les services enregistrés et de leur état actuel.
- **Eureka Client :** Chaque microservice agit en tant que client qui s'enregistre auprès du serveur Eureka au démarrage et envoie périodiquement des signaux de disponibilité (heartbeat). Les bibliothèques client Eureka sont disponibles pour différents langages de programmation.
- **Service Registration :** Les microservices s'enregistrent auprès du serveur Eureka en fournissant des informations telles que leur nom d'hôte, leur port et leurs URL de vérification de santé. Le serveur Eureka stocke ces informations dans son registre.
- **Service Discovery :** D'autres microservices qui ont besoin de communiquer avec un service spécifique peuvent interroger le serveur Eureka pour obtenir l'emplacement du service et d'autres métadonnées.

# Service Discovery and Registration (Eureka, Consul)

---

## Consul

Consul, développé par HashiCorp, est un outil complet de découverte de services et de gestion de configuration. Il offre des fonctionnalités de découverte de services, de vérification de la santé, de stockage de clés-valeurs et de coordination distribuée. Les principaux composants de Consul comprennent :

- **Consul Server:** Les serveurs Consul forment un cluster distribué qui maintient une vue cohérente des services enregistrés, des vérifications de la santé et des données clés-valeurs.
- **Consul Client:** Chaque microservice exécute un agent client Consul, qui s'enregistre auprès des serveurs Consul et fournit des informations sur les services et des vérifications de la santé. Les agents clients Consul communiquent avec les serveurs pour l'enregistrement des services, leur découverte et la surveillance de la santé.
- **Service Registration:** Les microservices s'enregistrent auprès des agents Consul en fournissant les détails de leur service. Les agents Consul transmettent ces informations aux serveurs Consul pour le stockage et la distribution.
- **Service Discovery:** Les microservices peuvent découvrir d'autres services en interrogeant les serveurs Consul ou en utilisant l'interface DNS fournie par Consul. Consul prend en charge différentes méthodes de découverte de services, notamment basées sur DNS, HTTP et gRPC.

# **Service Discovery and Registration (Eureka, Consul)**

---

**Eureka et Consul** sont deux solutions de découverte de services qui sont couramment utilisées dans les architectures de microservices. Voici les principales différences entre Eureka et Consul :

## **Développement et Support :**

- Eureka a été développé par Netflix et est souvent utilisé en combinaison avec d'autres outils de Netflix OSS.
- Consul est développé par HashiCorp, la même société derrière des outils populaires comme Vagrant, Terraform, et Vault.

## **Langage de Programmation :**

- Eureka est principalement utilisé avec des applications Java.
- Consul est agnostique en termes de langage, ce qui signifie qu'il peut être utilisé avec des applications écrites dans divers langages.

## **Stockage de Données :**

- Eureka stocke les informations de service dans la mémoire, ce qui le rend généralement plus rapide pour les opérations de recherche de services.
- Consul utilise un stockage distribué, ce qui lui permet de maintenir une cohérence plus stricte des données, mais peut être plus lent pour les opérations de recherche.

## **Prise en Charge de la Sécurité :**

- Consul offre un ensemble plus complet de fonctionnalités de sécurité, y compris l'authentification, l'autorisation et le chiffrement des données.
- Eureka peut nécessiter des ajouts ou des composants complémentaires pour atteindre le même niveau de sécurité.

# Service Discovery and Registration (Eureka, Consul)

---

## Architecture :

- Eureka suit une architecture client-serveur, où les services s'enregistrent auprès du serveur Eureka et les clients interrogent le serveur pour découvrir d'autres services.
- Consul suit une architecture peer-to-peer, où chaque nœud Consul est à la fois client et serveur, ce qui permet une plus grande redondance.

## Écosystème et Adoption:

- Eureka est plus couramment utilisé dans les environnements Java et les entreprises qui utilisent des outils Netflix OSS.
- Consul a gagné en popularité en raison de sa polyvalence et est utilisé dans un éventail plus large d'environnements.

Le choix entre Eureka et Consul dépend des besoins spécifiques de votre architecture de microservices, de vos compétences en matière d'outils et de votre tolérance aux compromis entre cohérence, disponibilité et partition-tolérance.

# Spring Boot Actuator

---

**Spring Boot Actuator** ajoute des fonctionnalités supplémentaires de monitoring à vos applications. Ces fonctionnalités sont disponibles via des endpoints HTTP.

Pour activer Spring Boot Actuator dans votre application, il suffit d'ajouter le starter correspondant aux dépendances dans le pom.xml de l'application :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Il faut aussi activer l'exposition des endpoints http dans votre fichier application.properties

```
management.endpoints.web.exposure.include=*
```

Ou bien dans votre fichier application.yml, en ajoutant la configuration suivante  
(Attention à respecter l'indentation):

Le endpoint de l'Actuator Spring Boot devient alors accessible sur l'url suivante :  
**http://localhost:8080/actuator**

Le résultat donne la liste de tous les endpoints de l'Actuator.

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

# Spring Boot Actuator

```
localhost:8081/actuator/
```

```
{"_links": {"self": {"href": "http://localhost:8081/actuator", "templated": false}, "beans": {"href": "http://localhost:8081/actuator/beans", "templated": false}, "caches-cache": {"href": "http://localhost:8081/actuator/caches/{cache}", "templated": true}, "caches": {"href": "http://localhost:8081/actuator/caches", "templated": false}, "health": {"href": "http://localhost:8081/actuator/health", "templated": false}, "health-path": {"href": "http://localhost:8081/actuator/health/{*path}", "templated": true}, "info": {"href": "http://localhost:8081/actuator/info", "templated": false}, "conditions": {"href": "http://localhost:8081/actuator/conditions", "templated": false}, "configprops-prefix": {"href": "http://localhost:8081/actuator/configprops/{prefix}", "templated": true}, "configprops": {"href": "http://localhost:8081/actuator/configprops", "templated": false}, "env": {"href": "http://localhost:8081/actuator/env", "templated": false}, "env-toMatch": {"href": "http://localhost:8081/actuator/env/{toMatch}", "templated": true}, "loggers": {"href": "http://localhost:8081/actuator/loggers", "templated": false}, "loggers-name": {"href": "http://localhost:8081/actuator/loggers/{name}", "templated": true}, "heapdump": {"href": "http://localhost:8081/actuator/heapdump", "templated": false}, "threaddump": {"href": "http://localhost:8081/actuator/threaddump", "templated": false}, "metrics-requiredMetricName": {"href": "http://localhost:8081/actuator/metrics/{requiredMetricName}", "templated": true}, "metrics": {"href": "http://localhost:8081/actuator/metrics", "templated": false}, "scheduledtasks": {"href": "http://localhost:8081/actuator/scheduledtasks", "templated": false}, "mappings": {"href": "http://localhost:8081/actuator/mappings", "templated": false}, "refresh": {"href": "http://localhost:8081/actuator/refresh", "templated": false}, "features": {"href": "http://localhost:8081/actuator/features", "templated": false}, "serviceregistry": {"href": "http://localhost:8081/actuator/serviceregistry", "templated": false}}}
```

The screenshot shows the Postman application interface. At the top, there's a header bar with 'API Network' and 'Explore' buttons, a search bar, and various user icons. Below the header, the main workspace shows a 'Report' section with an 'Overview' tab selected. A 'GET' request is listed under 'Overview' with the URL 'http://localhost:8081/actuator/'. The 'Body' tab is selected, displaying a JSON response. The response is a large object with many nested properties and href values. The 'Pretty' tab is selected, showing the JSON with line numbers and indentation. The status bar at the bottom indicates 'Status: 200 OK Time: 6 ms Size: 2.07 KB'. The right side of the screen has a sidebar with sections for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. There are also tabs for 'Cookies', 'Body', 'Cookies', 'Headers (5)', and 'Test Results'.

```
http://localhost:8081/actuator/
```

```
GET http://localhost:8081/actuator/
```

```
Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies
```

```
Query Params
```

Key	Value	Description	Bulk Edit
Key	Value	Description	

```
Body Cookies Headers (5) Test Results
```

```
Pretty Raw Preview Visualize JSON
```

```
1 {  
2   "_links": {  
3     "self": {  
4       "href": "http://localhost:8081/actuator",  
5       "templated": false  
7     },  
8     "beans": {  
9       "href": "http://localhost:8081/actuator/beans",  
10      "templated": false  
11    },  
12    "caches-cache": {  
13      "href": "http://localhost:8081/actuator/caches/{cache}",  
14      "templated": true  
15    },  
16    "caches": {  
17      "href": "http://localhost:8081/actuator/caches",  
18      "templated": false  
19    },  
20    "health": {  
21      "href": "http://localhost:8081/actuator/health",  
22      "templated": false  
23    },  
24    "health-path": {  
25      "href": "http://localhost:8081/actuator/health/{*path}",  
26      "templated": true  
27    },  
28    "info": {  
29      "href": "http://localhost:8081/actuator/info"  
30    }  
31  }  
32}
```

```
>Status: 200 OK Time: 6 ms Size: 2.07 KB Save as example
```

# Spring Boot Actuator

---

## Exemple avec le service « Refresh »

```
management.endpoints.web.exposure.include=refresh
```

localhost:9001/actuator/refresh

- Appel de l'API via Postman : Request method 'GET' not supported : il faut utiliser la méthode POST au lieu de GET
- Résultat : présenter les propriétés qui ont été modifiées depuis le dernier commit

## Exemple du service «health»

```
management.endpoints.web.exposure.include=health
```

localhost:9001/actuator/health

- Permet de tester si le microservice est opérationnel.
- Utilisé dans la surveillance en temps réel des instances en cours d'exécution des microservices via un check régulier de cet endpoint
- Cet endpoint fonctionne en analysant le retour de toutes les classes qui héritent de HealthIndicator,
- Chacune de ces classes doit effectuer ses tests, et dire si le service est UP ou DOWN.
- Il est possible de créer son propre indicateur afin de tester si le microservice fonctionne correctement en fonction des critères contextuels du microservice

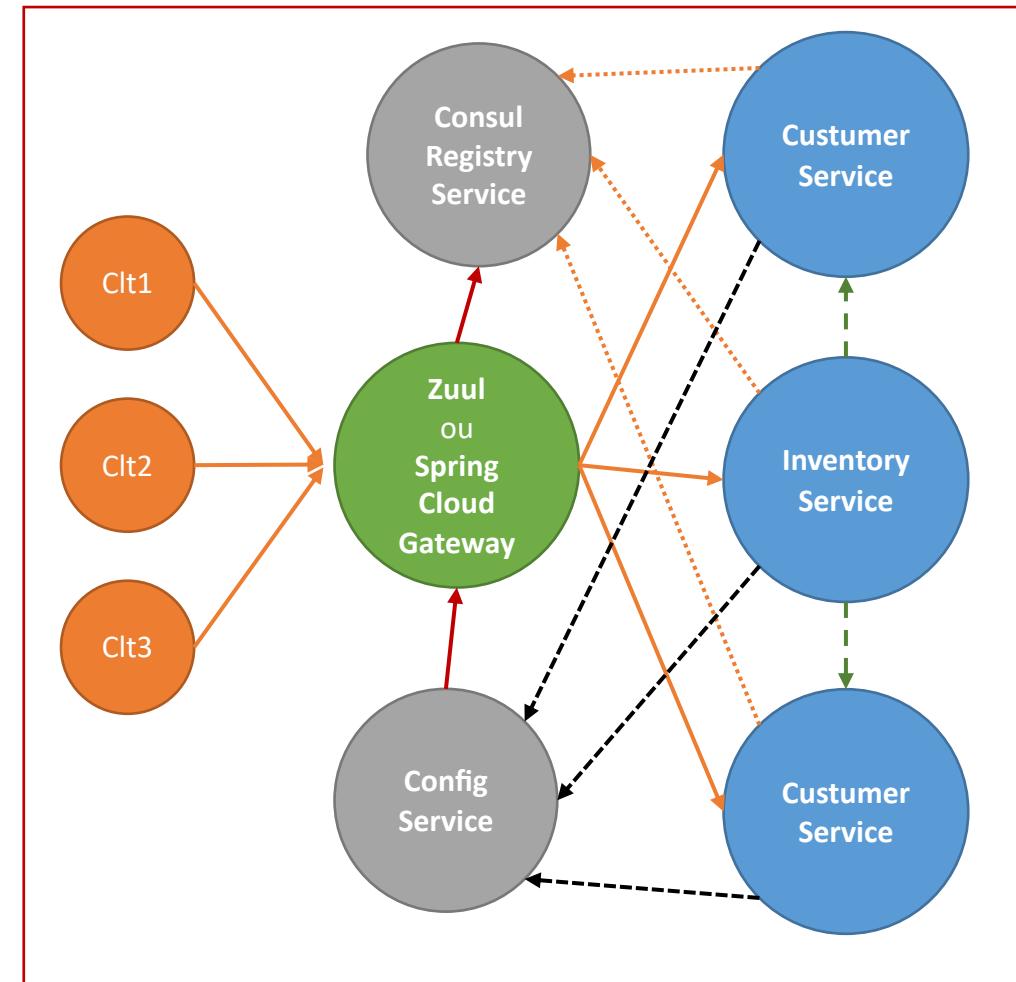
## Application: Architecture micro service avec Spring Cloud

Créer une application basée sur 3 services métiers:

- Service de gestion des clients
- Service de gestion de d'inventaire
- Service de gersion des commandes

L'orchestration des services se fait via les services techniques de Spring Cloud :

- Spring Cloud Gateway Service comme service proxy
- Consul Registry Service
- Resilience4j : Circuit Breaker; ..



## Application: Configuration Service (CONFIGURATION EXTERNALISÉE)



**Project**  
 Gradle - Groovy    Gradle - Kotlin  
 Maven    Java    Kotlin    Groovy

**Spring Boot**  
 3.2.0 (SNAPSHOT)    3.2.0 (RC1)    3.1.6 (SNAPSHOT)    3.1.5  
 3.0.13 (SNAPSHOT)    3.0.12    2.7.18 (SNAPSHOT)    2.7.17

**Project Metadata**

Group	org.sid
Artifact	config-service
Name	config-service
Description	Demo project for Spring Boot
Package name	org.sid.config-service
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 21 <input checked="" type="radio"/> 17 <input type="radio"/> 11 <input type="radio"/> 8

### Selected dependencies:

- **Config Server**: Central management for configuration via Git, SVN, or HashiCorp Vault.
- **Spring Boot Actuator**: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- **Consul Discovery**: Service discovery with Hashicorp Consul.

## **Application:** Configuration Service

---

**Spring Cloud Config** est un composant de la **Spring Cloud** qui **facilite la gestion des configurations de manière centralisée** pour les applications distribuées(micro-services). Il permet de stocker les configurations de différentes applications et environnements dans un emplacement centralisé, appelé "configuration store". Ces configurations peuvent inclure des paramètres, des valeurs de propriétés et d'autres données de configuration nécessaires au bon fonctionnement des applications.

Quelques points clés à retenir sur Spring Cloud Config :

**Centralisation des configurations** : Spring Cloud Config permet de centraliser les configurations de plusieurs applications au sein d'un seul emplacement, ce qui facilite la gestion et la coordination des configurations.

**Utilisation de Git** : Le configuration store est généralement versionné à l'aide de Git, un système de contrôle de version populaire. Cela signifie que les configurations sont stockées dans un référentiel Git, ce qui permet de suivre les modifications, de gérer les versions et d'appliquer des bonnes pratiques de gestion de configuration.

**Changement à la demande** : Les applications clientes peuvent charger leurs configurations à partir du serveur de configuration à la demande. Cela signifie que les configurations peuvent être modifiées sans avoir à redémarrer les applications, ce qui est particulièrement utile pour la gestion dynamique des configurations.

**Support pour différents environnements** : Spring Cloud Config permet de gérer les configurations spécifiques à différents environnements, tels que le développement, la production, la pré-production, etc.

**Sécurité** : Spring Cloud Config intègre des fonctionnalités de sécurité pour s'assurer que seules les applications autorisées ont accès aux configurations.

En utilisant Spring Cloud Config, les équipes de développement peuvent centraliser la gestion des configurations, améliorer la cohérence des configurations entre les environnements, et faciliter les mises à jour de configuration sans nécessiter de redémarrage des applications. Cela contribue à rendre le déploiement et la maintenance des applications distribuées plus efficaces.

## Application: customer-service

### Selected dependencies:

- **Spring Web**: Build web, including RESTful, applications using Spring MC. 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.
- **Spring Boot Actuator**: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- **Consul Discovery** : Service discovery with Hashicorp Consul.
- **Config Client**: Client that connects to a Spring Cloud Config Server to fetch the application's configuration.



Project	Language
<input type="radio"/> Gradle - Groovy	<input type="radio"/> Gradle - Kotlin
<input checked="" type="radio"/> Maven	<input type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy
Spring Boot	
<input type="radio"/> 3.2.0 (SNAPSHOT)	<input type="radio"/> 3.2.0 (RC1)
<input type="radio"/> 3.0.13 (SNAPSHOT)	<input type="radio"/> 3.0.12
	<input type="radio"/> 2.7.18 (SNAPSHOT)
	<input checked="" type="radio"/> 2.7.17
Project Metadata	
Group	org.sid
Artifact	customer-service
Name	customer-service
Description	Demo project for Spring Boot
Package name	org.sid.customer-service
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 21 <input checked="" type="radio"/> 17 <input type="radio"/> 11 <input type="radio"/> 8

# Application: inventory-service

## Selected dependencies:

- **Spring Web**: Build web, including RESTful, applications using Spring MC. 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.
- **Spring Boot Actuator**: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- **Consul Discovery** : Service discovery with Hashicorp Consul.
- **Config Client**: Client that connects to a Spring Cloud Config Server to fetch the application's configuration.



<b>Project</b>	<b>Language</b>		
<input type="radio"/> Gradle - Groovy	<input type="radio"/> Gradle - Kotlin	<input checked="" type="radio"/> Java	<input type="radio"/> Kotlin
<input checked="" type="radio"/> Maven	<input type="radio"/>	<input type="radio"/>	<input type="radio"/> Groovy
<b>Spring Boot</b>			
<input type="radio"/> 3.2.0 (SNAPSHOT)	<input type="radio"/> 3.2.0 (RC1)	<input type="radio"/> 3.1.6 (SNAPSHOT)	<input type="radio"/> 3.1.5
<input type="radio"/> 3.0.13 (SNAPSHOT)	<input type="radio"/> 3.0.12	<input type="radio"/> 2.7.18 (SNAPSHOT)	<input checked="" type="radio"/> 2.7.17
<b>Project Metadata</b>			
Group	org.sid		
Artifact	inventory-service		
Name	inventory-service		
Description	Demo project for Spring Boot		
Package name	org.sid.inventory-service		
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War	
Java	<input type="radio"/> 21	<input checked="" type="radio"/> 17	<input type="radio"/> 11
			<input type="radio"/> 8

## Application: order-service

### Selected dependencies:

- **Spring Web**: Build web, including RESTful, applications using Spring MC. 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.
- **Spring Boot Actuator**: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- **Consul Discovery** : Service discovery with Hashicorp Consul.
- **Config Client**: Client that connects to a Spring Cloud Config Server to fetch the application's configuration.



<b>Project</b>	<b>Language</b>		
<input type="radio"/> Gradle - Groovy	<input type="radio"/> Gradle - Kotlin	<input checked="" type="radio"/> Java	<input type="radio"/> Kotlin
<input checked="" type="radio"/> Maven	<input type="radio"/>	<input type="radio"/>	<input type="radio"/> Groovy
<b>Spring Boot</b>			
<input type="radio"/> 3.2.0 (SNAPSHOT)	<input type="radio"/> 3.2.0 (RC1)	<input type="radio"/> 3.1.6 (SNAPSHOT)	<input type="radio"/> 3.1.5
<input type="radio"/> 3.0.13 (SNAPSHOT)	<input type="radio"/> 3.0.12	<input type="radio"/> 2.7.18 (SNAPSHOT)	<input checked="" type="radio"/> 2.7.17
<b>Project Metadata</b>			
Group	org.sid		
Artifact	order-service		
Name	order-service		
Description	Demo project for Spring Boot		
Package name	org.sid.order-service		
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War	
Java	<input type="radio"/> 21	<input checked="" type="radio"/> 17	<input type="radio"/> 11
			<input type="radio"/> 8

## Application: Gateway-service

### 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.
- **Eureka Discovery Client** : a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
- **Config Client**: Client that connects to a Spring Cloud Config Server to fetch the application's configuration

The screenshot shows the Spring Initializr interface at [start.spring.io](https://start.spring.io). The page title is "spring initializr". The "Project" section is set to Maven. The "Language" section is set to Java. The "Spring Boot" section is set to version 2.7.17. The "Project Metadata" section includes fields for Group (org.sid), Artifact (Gateway-service), Name (Gateway-service), Description (Demo project for Spring Boot), Package name (org.sid.Gateway-service), and Packaging (Jar). Below the packaging field, there are statistics: Java 21, Jar 17, War 11, and Other 8.

Project	Language			
<input type="radio"/> Gradle - Groovy	<input type="radio"/> Gradle - Kotlin	<input checked="" type="radio"/> Java	<input type="radio"/> Kotlin	<input type="radio"/> Groovy
<input checked="" type="radio"/> Maven				

Spring Boot			
<input type="radio"/> 3.2.0 (SNAPSHOT)	<input type="radio"/> 3.2.0 (RC1)	<input type="radio"/> 3.1.6 (SNAPSHOT)	<input type="radio"/> 3.1.5
<input type="radio"/> 3.0.13 (SNAPSHOT)	<input type="radio"/> 3.0.12	<input type="radio"/> 2.7.18 (SNAPSHOT)	<input checked="" type="radio"/> 2.7.17

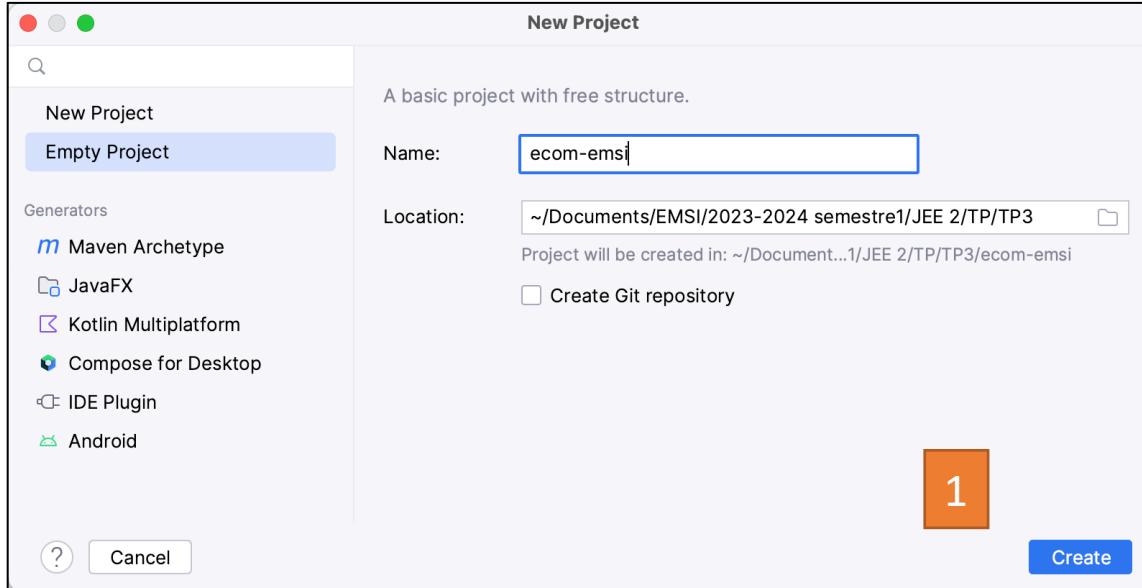
Project Metadata	
Group	org.sid
Artifact	Gateway-service
Name	Gateway-service
Description	Demo project for Spring Boot
Package name	org.sid.Gateway-service
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War

Java 21    17    11    8

# Application:

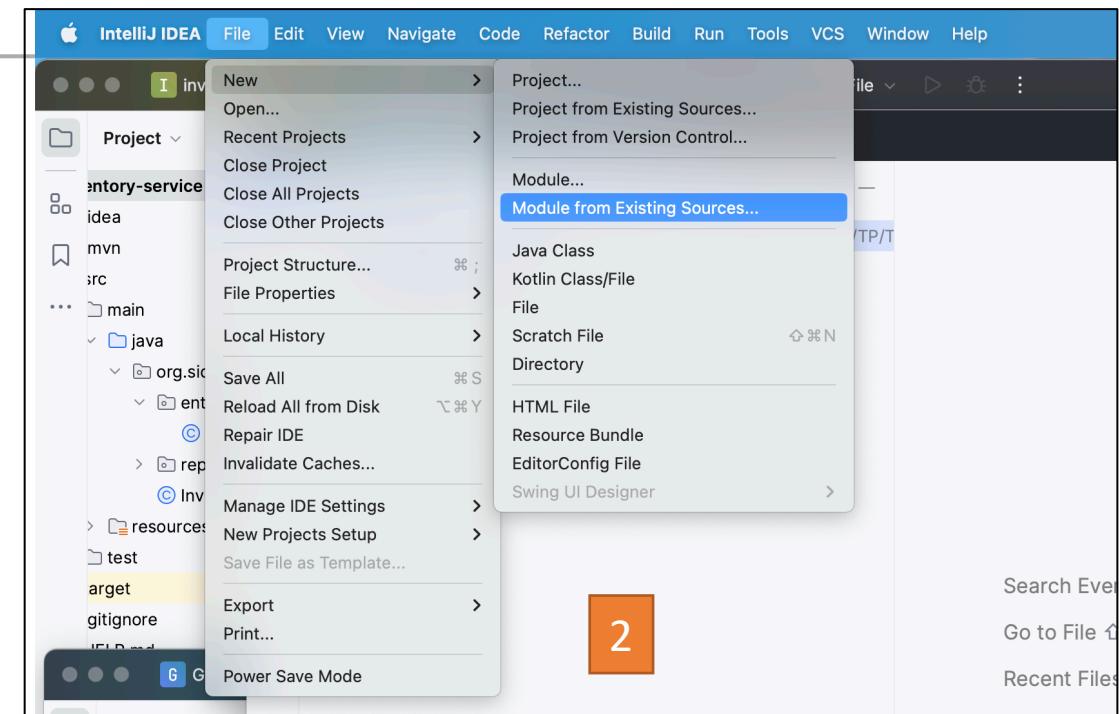
## Grouper tous les micro-services dans un seul workspace

### 1. Cree un projet vide (Empty Project)

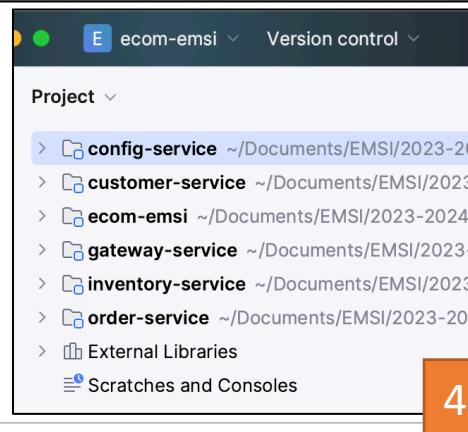


1

### 2. Ajouter les autres projets comme modules

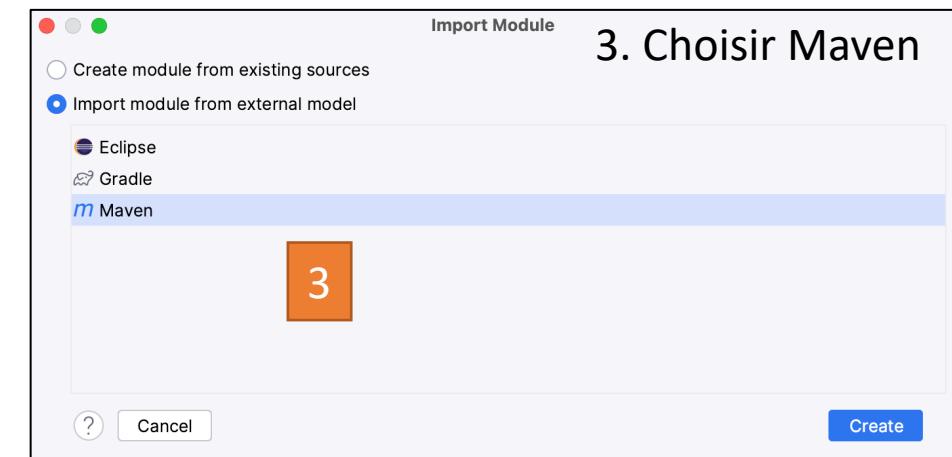


2



4

### 3. Choisir Maven



3

# Application: registration service

## Télécharger Consul

The screenshot shows a web browser window for the HashiCorp Developer AI website at developer.hashicorp.com/consul/downloads?product\_intent=consul. The page title is "Install Consul". The main content area features a "Consul" icon and the heading "Install Consul". Below it, a sub-section titled "Operating System" shows "macOS" selected from a dropdown menu, with other options like Windows, Linux, FreeBSD, and Solaris. Under "Package manager for macOS", there is a code block with two commands:  
\$ brew tap hashicorp/tap  
\$ brew install hashicorp/tap/consul

Below this, under "Binary download for macOS", there are two sections: "AMD64" (Version: 1.17.0) and "ARM64" (Version: 1.17.0), each with a "Download" button.

The left sidebar contains navigation links such as "Consul Home", "Install Consul" (which is highlighted in pink), "Get Started", "What is Consul?", "Consul with Kubernetes", "Consul with Virtual Machines", "HCP Consul", "Install Consul K8s CLI", "Resources", "Tutorial Library", "Certifications", "Community Forum", "Support", and "GitHub".

The right sidebar includes sections for "About Consul", "Featured docs" (links to "What is Consul?", "API Introduction", and "Commands (CLI)"), and a "HCP Consul" section with a "Try HCP Consul for free" button.

## **Application:** registration service

# Démarrer Consul

## Récupère l'adresse ip

Carte inconnue Connexion au réseau local :

2

Statut du média . . . . . : Média déconnecté  
Suffixe DNS propre à la connexion . . . :

### Carte réseau sans fil Connexion au réseau local\* 1 :

Statut du média. . . . . : Média déconnecté  
Suffixe DNS propre à la connexion. . . :

Carte réseau sans fil Connexion au réseau local\* 2 :

Statut du média. . . . . : Média déconnecté  
Suffixe DNS propre à la connexion. . . :

#### Carte réseau sans fil Wi-Fi :

Carte Ethernet Connexion réseau Bluetooth :

Statut du média . . . . . : Média déconnecté  
Suffixe DNS propre à la connexion . . . :

## L'ajouter dans la commande

3

```
C:\Tools\consul>consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui -bind=192.168.95
```

# Application: registration service

## Démarrer Consul

The screenshot shows the Consul UI interface. The left sidebar has a dark theme with white text and includes the following navigation items:

- Overview
- Services** (selected)
- Nodes
- Key/Value
- Intentions
- ACCESS CONTROLS (red dot)
- Tokens
- Policies
- Roles
- Auth Methods
- ORGANIZATION
- Peers

The main content area is titled "Services 1 total". It features a search bar with "Search" and "Search Across" dropdowns, and filters for "Health Status", "Service Type", and "Unhealthy to Healthy". A single service entry is listed:

- consul**

Below the service name, it says "1 instance". At the bottom left of the main area, it says "Consul v1.17.0".

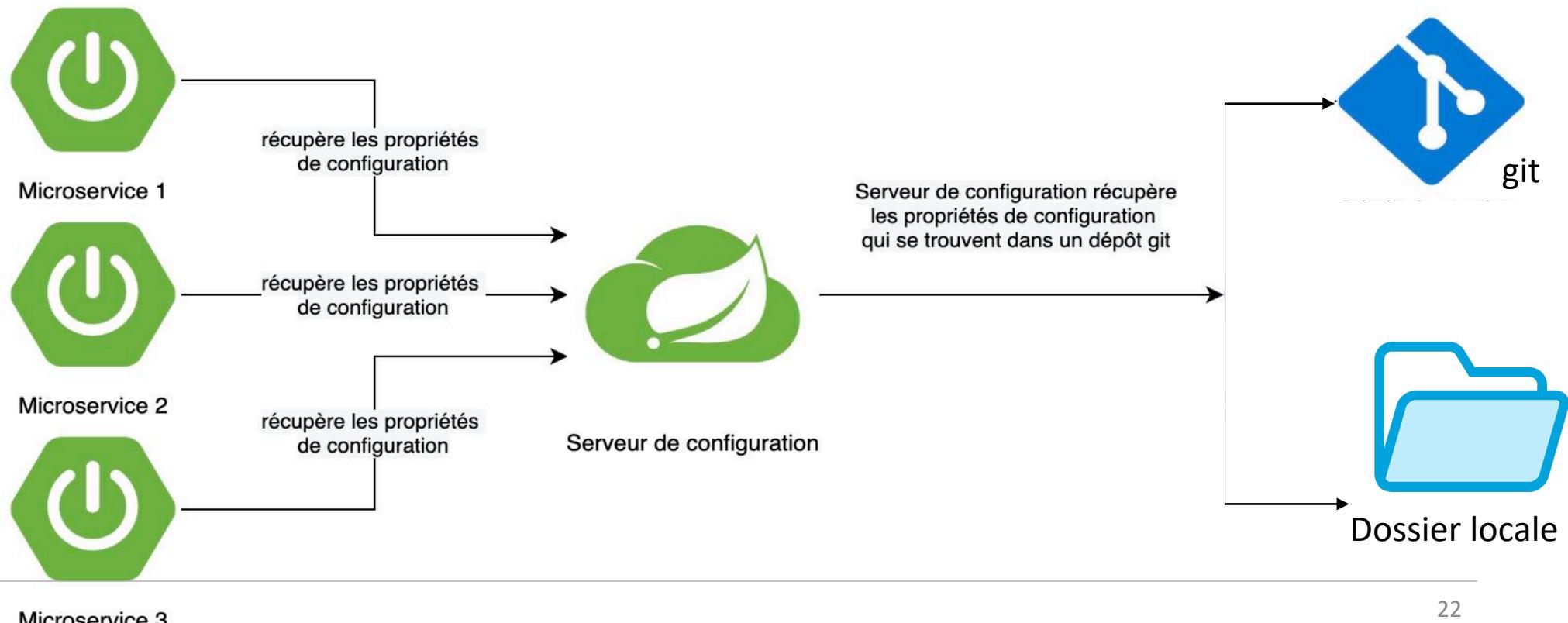
4

Visualiser l'interface

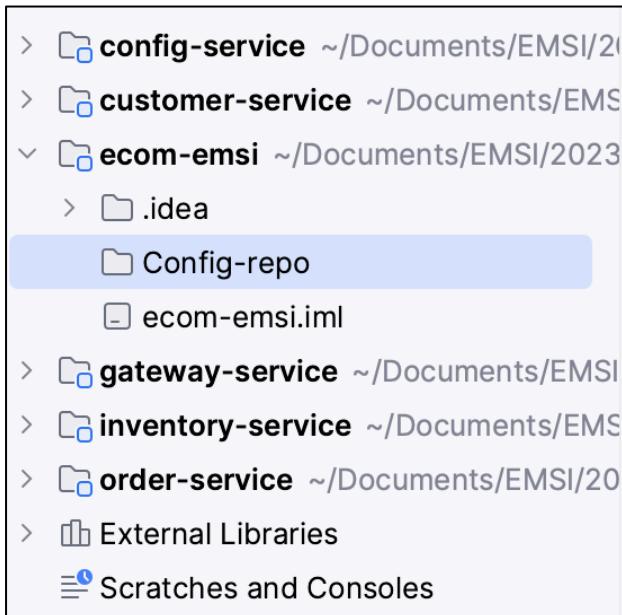
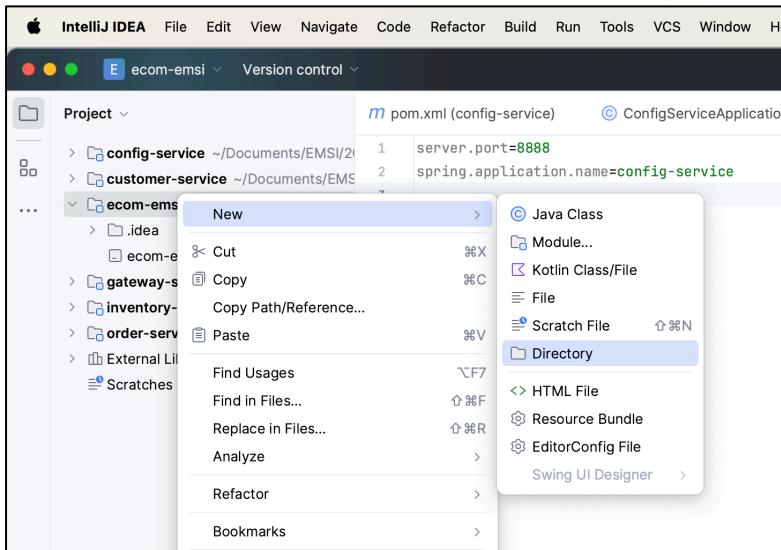
## Application: Config-service (CONFIGURATION EXTERNALISÉE)

Pour externaliser les fichiers de configuration, nous utiliserons Spring Cloud Config en tant que serveur de distribution de fichiers de configuration. L'idée est de stocker tous les fichiers de configuration dans un référentiel Git. En utilisant des conventions de nommage de fichiers spécifiques, Spring Cloud Config peut déterminer quel fichier de configuration est destiné à quel microservice, en se basant sur le nom du microservice.

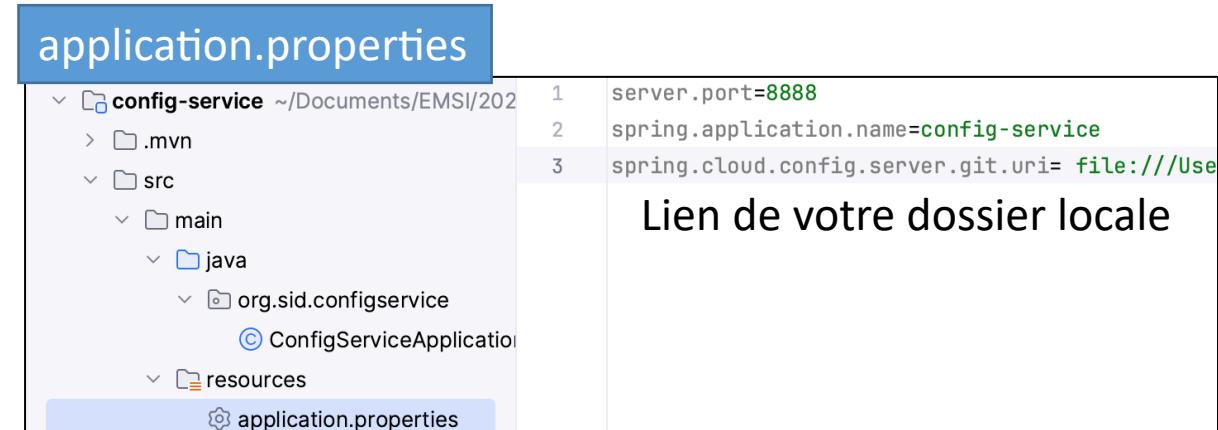
Pour effectuer des modifications ultérieures dans la configuration d'un microservice, il suffira de pousser les changements dans le référentiel Git. Spring Cloud Config détectera automatiquement la nouvelle version et la mettra à disposition, le tout sans nécessiter l'arrêt des microservices !



# Application: Config-service (CONFIGURATION EXTERNALISÉE)



Cree un dossier local où on va mettre la configuration. C'est un dossier externe au projet (on peut le mettre n'importe où)



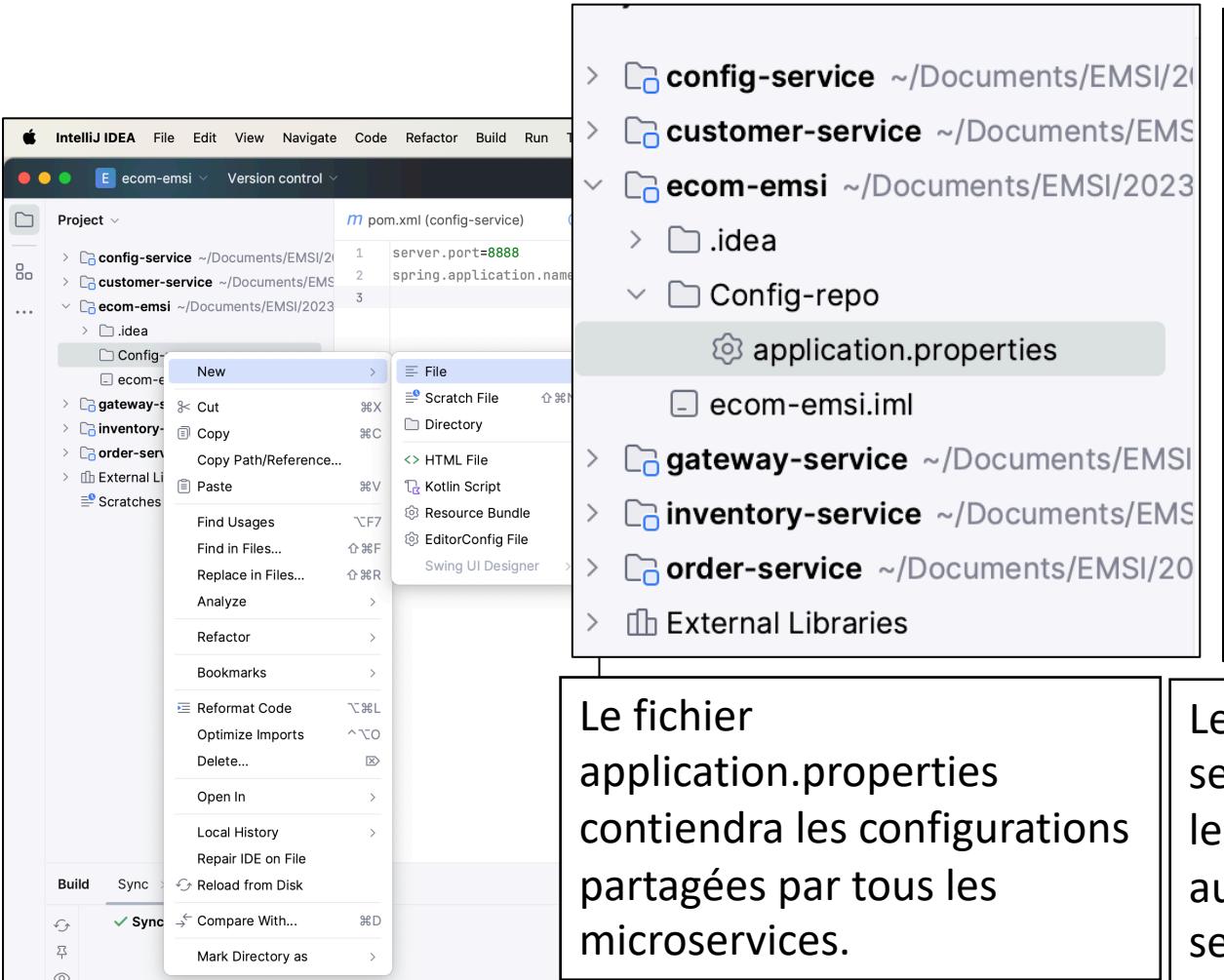
```
ConfigServiceApplication

@SpringBootApplication
@EnableConfigServer ←
@EnableDiscoveryClient //activer explicitement discovery cote consul normalement doit etre activer par default
public class ConfigServiceApplication {

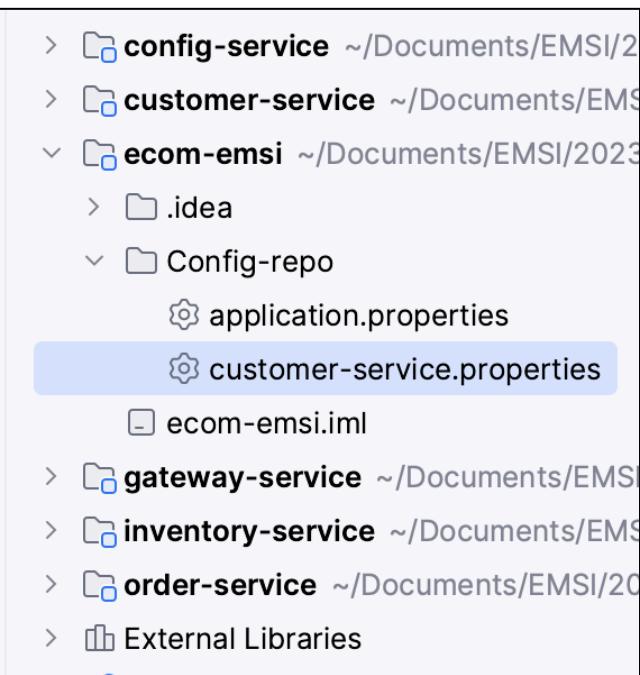
    public static void main(String[] args) { SpringApplication.run(ConfigServiceApplication.class, args); }
}
```

# Application: Config-service (CONFIGURATION EXTERNALISÉE)

Cree les fichier de configuration a l'intérieur de ce dossier externe



Le fichier application.properties contiendra les configurations partagées par tous les microservices.



Le fichier customer-service.properties contiendra les configurations spécifique au microservice customer-service.

On peut spécifier les configuration de divers environnements.

Exemple:

- customer-service-dev.properties va definir la configuration de l'environnement de developpement.
- customer-service-prod.properties va definir la configuration de l'environnement de production.

Grâce à cette correspondance de noms que le serveur de configuration fera le lien entre ce fichier et le microservice correspondant.

## Application: Config-service (CONFIGURATION EXTERNALISÉE)

---

Cree les fichier de configuration a l'intérieur de ce dossier externe

Pour tester ajouter `customer.params.x=6555`  
Dans **application.properties**

1	<code>customer.params.p1=6555</code>
---	--------------------------------------

Et ajouter

`global.params.x= 555`  
`global.params.y= 267`

1	<code>global.params.x= 555</code>
2	<code>global.params.y= 267</code>

Dans **customer-service.properties**

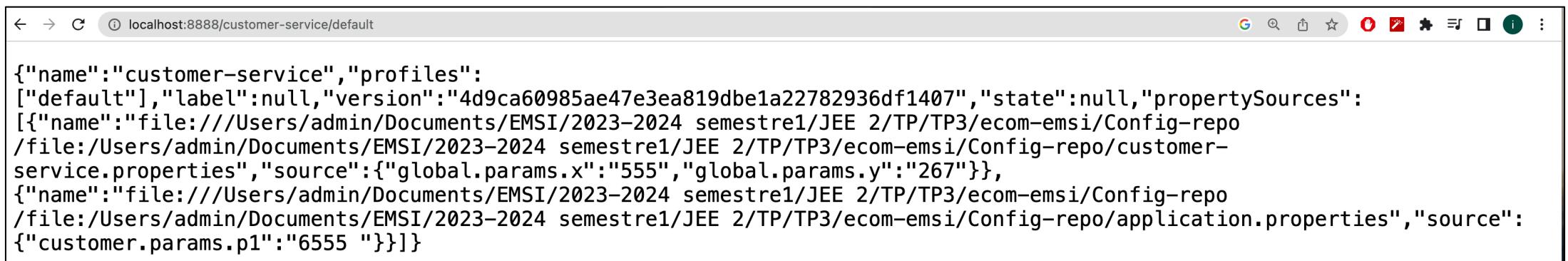
## Application: Config-service (CONFIGURATION EXTERNALISÉE)

Spring Cloud Config Server a besoin d'un repository et on a besoin d'un system de gestion de versionning

- > Ouvrir le dossier config-repo dans le terminale
- > exécuter les commandes suivantes

```
admin@Imanes-MacBook-Pro Config-repo % git init
Initialized empty Git repository in /Users/admin/Documents/EMSI/2023-2024 semestre1/JEE 2/TP/TP3/ecom-emsi/Config-repo/.git/
admin@Imanes-MacBook-Pro Config-repo % git add .
admin@Imanes-MacBook-Pro Config-repo % git commit -m "V1"
```

Test: visiter <http://localhost:8888/default>



## Application: Config-service (CONFIGURATION EXTERNALISÉE)

Revenant à **customer-service**: ajouter les configurations suivantes dans **application.properties**

The screenshot shows a file explorer window with the following structure:

- pom.xml** (blue icon)
- customer-service** (~/Documents/EMS)
  - .mvn** (grey folder)
  - src** (grey folder)
    - main** (grey folder)
      - java** (blue folder)
      - resources** (orange folder)
        - static** (grey folder)
        - templates** (grey folder)
    - application.properties** (grey file)

On the right, the **application.properties** file is open with the following content:

```
1 server.port=8081
2 spring.application.name=customer-service
3 spring.config.import=optional:configserver:http://localhost:8888/
4
```

A callout box with an arrow points from the line `spring.config.import=optional:configserver:http://localhost:8888/` to the text: "Montre ou micro-service ou il doit aller pour récupérer sa configuration".

## Application: Config-service (CONFIGURATION EXTERNALISÉE)

Toujours dans **customer-service**: ajouter un package **web**. Dans Web crée la classe **CustomerConfigTestController**

```
package org.sid.customerservice.web;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Map;

no usages

@RestController
public class CustomerConfigTestController {

    1 usage
    @Value("${customer.params.p1}")//inject la valeur d'un parametre de configuration depuis le fichier de configuration
    private String p1;
    1 usage
    @Value("${global.params.x}")
    private String x;
    1 usage
    @Value("${global.params.y}")
    private String y;

    no usages
    @GetMapping("/params")
    public Map<String, String> params() {
        return Map.of( k1: "p1",p1, k2: "x",x, k3: "y",y);
    }
}
```

Le but est de tester si le microservice récupère sa configuration depuis le service de configuration.

## Application: Config-service (CONFIGURATION EXTERNALISÉE)

Lors du démarrage de customer-service il apparait sur consul.

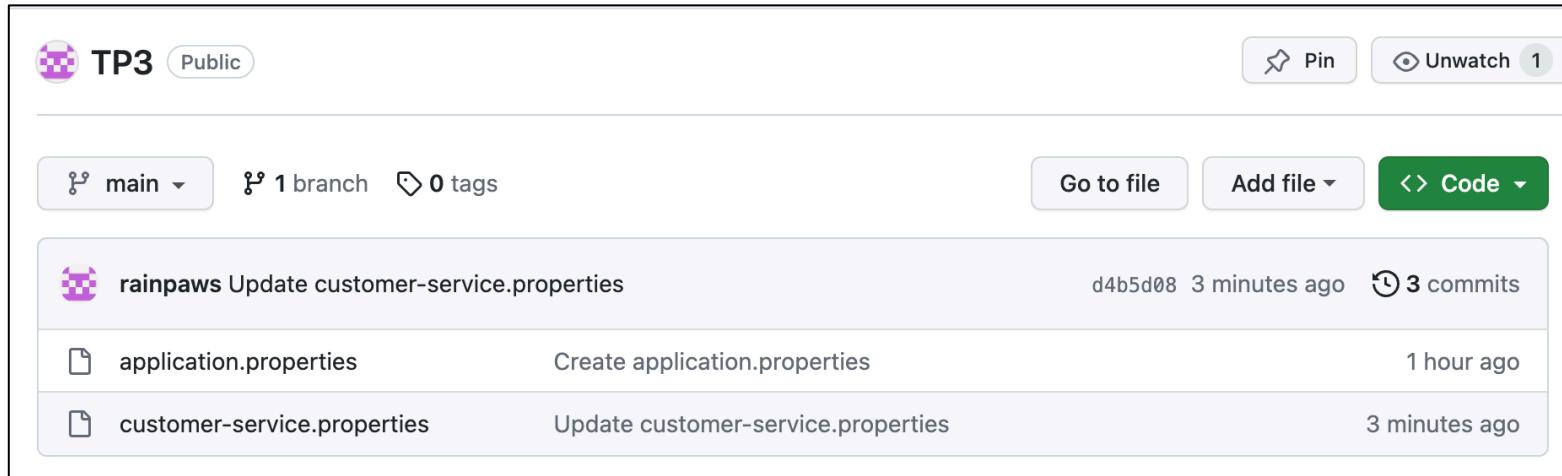
The screenshot shows the Consul UI interface. The left sidebar has a dark theme with white text and includes links for Overview, Services (which is selected), Nodes, Key/Value, Intentions, ACCESS CONTROLS (with a red dot), Tokens, Policies, Roles, Auth Methods, ORGANIZATION, and Peers. The main content area is titled "Services 3 total". It features a search bar with "Search" and "Search Across" dropdowns, and filters for "Health Status" (set to "consul"), "Service Type" (set to "any"), and "Unhealthy to Healthy" (set to "Unhealthy to Healthy"). Below these are three service entries: "consul" (1 instance, healthy), "config-service" (1 instance, healthy), and "customer-service" (1 instance, healthy).

Customer service arrive bien à récupérer les paramètres

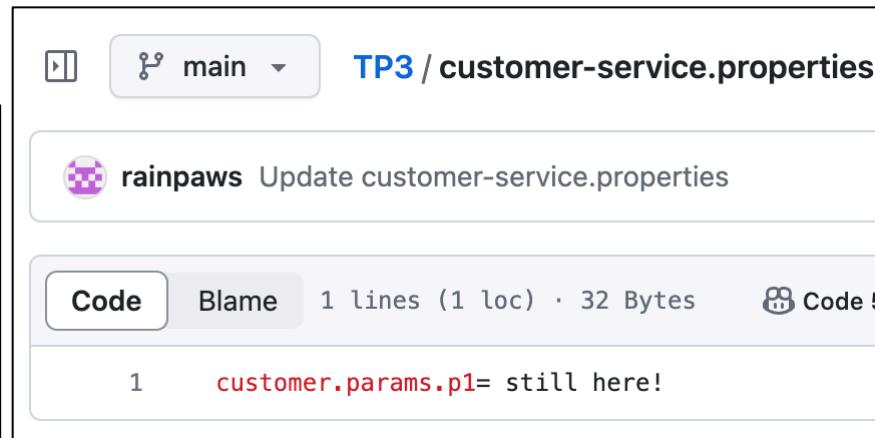
The screenshot shows a browser window with the URL "localhost:8081/params". The page displays a JSON object: {"y": "267", "x": "555", "p1": "6555 "}. An annotation box points from the text above to this JSON response.

## Application: Config-service (CONFIGURATION EXTERNALISÉE)

Au lieu d'utiliser un dossier local on peut utiliser Github

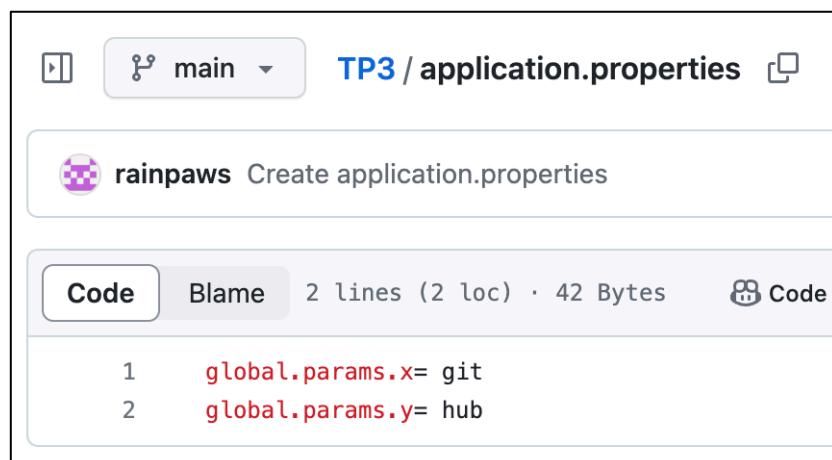


TP3 (Public)  
main 1 branch 0 tags  
rainpaws Update customer-service.properties  
application.properties Create application.properties  
customer-service.properties Update customer-service.properties



main TP3 / customer-service.properties  
rainpaws Update customer-service.properties  
Code Blame 1 lines (1 loc) · 32 Bytes Code  
1 customer.params.p1= still here!

Le fichier application.properties de config-service



main TP3 / application.properties  
rainpaws Create application.properties  
Code Blame 2 lines (2 loc) · 42 Bytes Code  
1 global.params.x= git  
2 global.params.y= hub

```
server.port=8888
spring.application.name=config-service
#spring.cloud.config.server.git.uri= file:///Users/admin/Documents/EMSI/2
spring.cloud.config.server.git.uri=https://github.com/username/TP3.git
github.token= Votre token
```

Testez pour voir si vous parvenez à récupérer la configuration

