

Atelier 4

Externaliser la configuration des Microservices

1. Ajoutez vos propres configurations

Quand vous développez un Microservice, vous avez souvent besoin de **définir certaines propriétés ou valeurs fixes** qui conditionnent le fonctionnement de votre application. Par exemple : le port d'écoute d'un Microservice, une clé secrète d'accès à des ressources protégées, ou encore des valeurs liées aux langues et dictionnaires utilisés.

Pourquoi ne pas mettre tout simplement des constantes dans le code ?

Cette solution est à éviter, car quand vous avez un micro commerce avec énormément de valeurs fixes à renseigner, il devient complexe de les retrouver dans les dizaines de classes de votre Microservice.

Pire encore, quand vous souhaitez changer une constante, il faut arrêter le Microservice et le mettre à jour avant de le redéployer.

La solution est de **regrouper toutes vos constantes de configuration** dans un fichier que nous avons depuis le début : ***application.properties***.

Pour prendre un exemple, nous allons essayer de créer une valeur limite du nombre de produits à retourner quand on fait appel à l'URI */Produits* (méthode *listeDesProduits*).

Rendez-vous dans ***application.properties*** de Microservice-produits et ajoutez ceci :

```
spring.application.name=microservice-produits

server.port 9001

#Configurations H2
spring.jpa.show-sql=true
spring.h2.console.enabled=true

#défini l'encodage pour data.sql
spring.datasource.sql-script-encoding=UTF-8

#Nos configurations
mes-configs.limitDeProduits= 4
```

Nous avons donc ajouté `mes-configs.limitDeProduits= 4`. Il faut veiller à ce que tous les noms de vos propriétés soient précédés d'un préfixe afin qu'elles soient identifiables plus tard. Dans notre cas, **le préfixe est *mes-configs***.

Afin de récupérer les valeurs que nous avons indiquées dans *application.configuration*, nous allons utiliser l'annotation `@ConfigurationProperties`. Pour ce faire, nous allons créer une classe de configuration.

Créez une classe appelée ***ApplicationPropertiesConfiguration*** dans un package nommé ***configurations***.

ApplicationPropertiesConfiguration.java

```
package com.mproduits.configurations;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties("mes-configs")
public class ApplicationPropertiesConfiguration {

    private int limitDeProduits;

    public int getLimitDeProduits() {
        return limitDeProduits;
    }

    public void setLimitDeProduits(int limitDeProduits) {
        this.limitDeProduits = limitDeProduits;
    }

}
```

Il ne faut pas oublier d'ajouter la dépendance nécessaire au fichier pom.xml.

Explications :

- `@Component` : demande à Spring de scanner cette classe à la recherche de configurations.
- `@ConfigurationProperties("mes-configs")` : précise que cette classe de configuration va récupérer des propriétés dans *application.properties* dont le préfixe est : *mes-configs*.
- Il suffit ensuite de déclarer des propriétés avec les mêmes noms que celles du fichier de configuration. Dans notre cas, il s'agit de `limitDeProduits`. Il ne faut pas oublier de générer les Getters et Setters.

Il nous suffit à présent de retourner dans notre contrôleur pour accéder aux valeurs très simplement.

Extrait de *ProductController.java* :

```
@Autowired
ApplicationPropertiesConfiguration appProperties;

// Affiche la liste de tous les produits disponibles
@GetMapping(value = "/Produits")
public List<Product> listeDesProduits(){

    List<Product> products = productDao.findAll();

    if(products.isEmpty()) throw new ProductNotFoundException("Aucun produit n'est disponible à la vente");

    List<Product> listeLimitee = products.subList(0, appProperties.getLimitDeProduits());

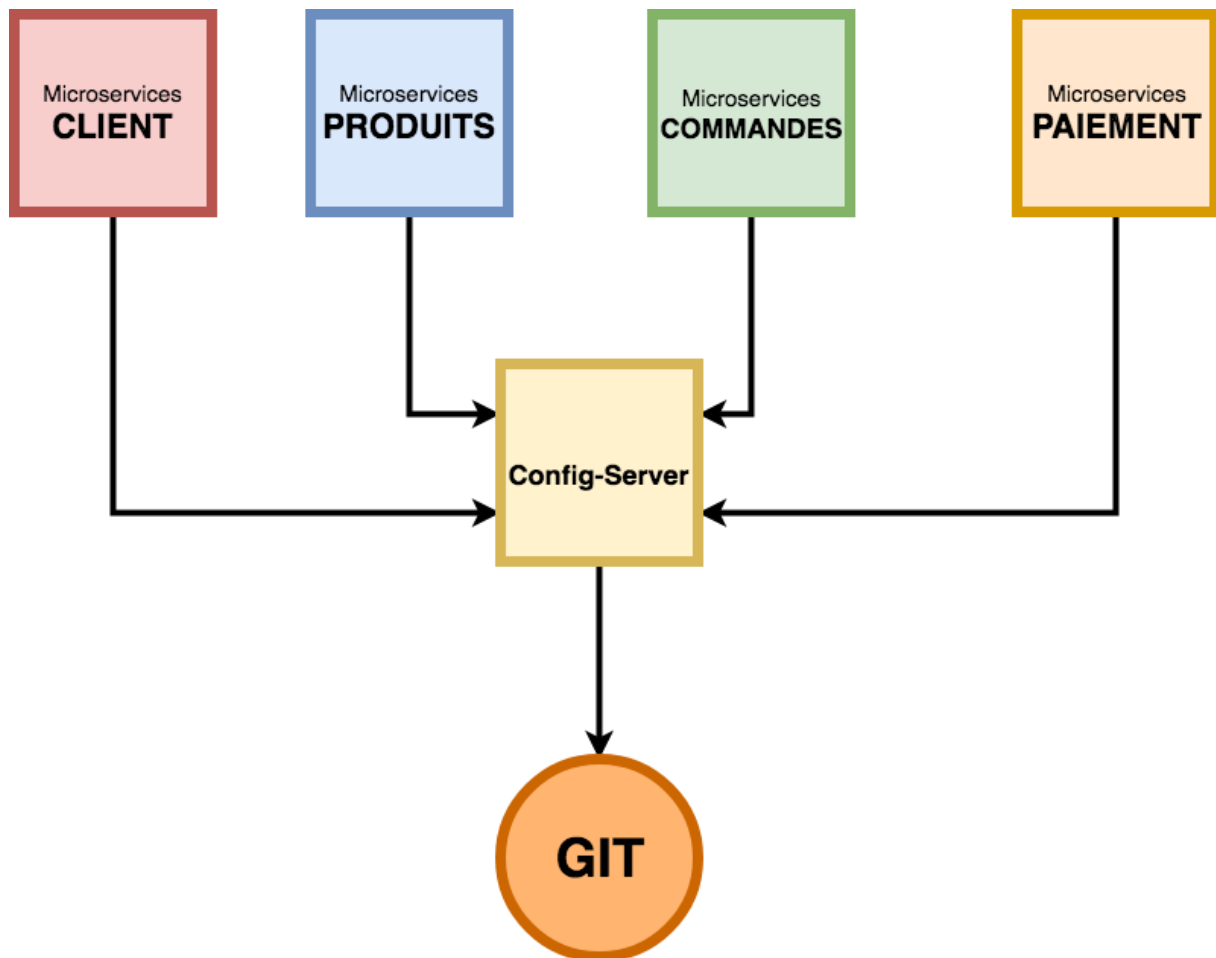
    return listeLimitee;
}
```

Explications :

- On crée une variable de type `ApplicationPropertiesConfiguration` qu'on "Autowire". Spring nous fournit donc une instance correspondante.
- `appProperties.getLimitDeProduits()` va retourner le chiffre 4 que nous avons défini dans le fichier de configuration. On le passe donc à `subList` qui coupe une liste donnée à la limite donnée en 2^{ème} argument.

Voilà, le tour est joué ! Vous pouvez définir autant de valeurs que vous le souhaitez dans le fichier de configuration et y accéder très simplement dans votre code.

2. Fonctionnement de l'externalisation



Afin d'externaliser les fichiers de configuration, nous allons utiliser un Edge Microservice appelé **Spring Cloud Config**. Il se positionne comme serveur de **distribution des fichiers de configuration** (figure ci-dessus).

Il suffit de placer tous les fichiers de configuration dans un **dépôt GIT**. Grâce à des conventions de nommage de fichiers, Spring Cloud Config sera capable de savoir quel fichier va servir à quel Microservice en se basant sur le nom du Microservice.

Pour modifier plus tard la configuration d'un Microservice, il suffira de pousser les changements dans le GIT. Spring Cloud Config se mettra alors à servir la nouvelle version ! Pas besoin d'arrêter le moindre Microservice !

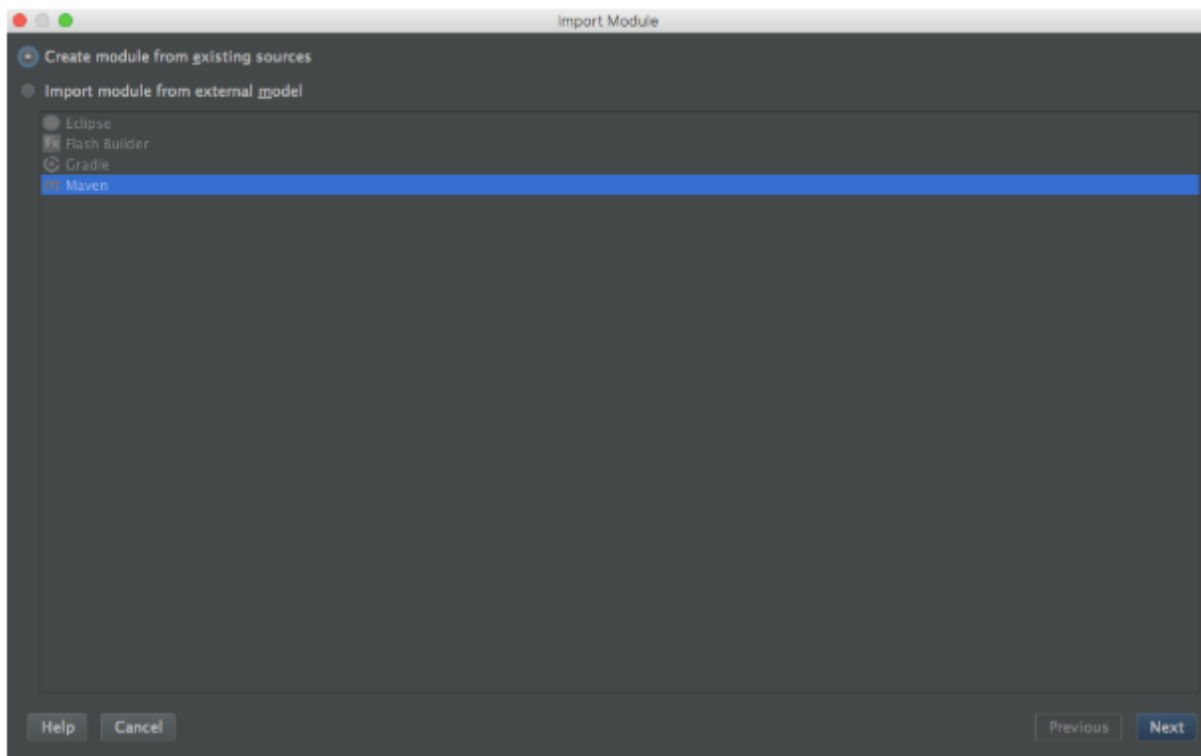
3. Création du dépôt GIT

Commençons par le bas du diagramme et créons un dépôt GIT dans lequel iront nos fichiers de configuration.

Créez un **compte sur Github** (si vous n'en avez pas déjà un), puis créez un dépôt et appelez-le par exemple : *mcommerce-config-repo*.

Revenez dans le dossier du projet Mcommerce sur votre ordinateur et créez un **nouveau dossier appelé *config-server-repo***. Nous allons mettre dans ce dossier les fichiers de configuration des Microservices, puis nous le connecterons au dépôt distant que vous avez créé.

Importez ce dossier en tant que nouveau module, comme vous l'avez fait pour les autres Microservices. La différence est que, cette fois, vous choisirez "**Create module from existing source**" :

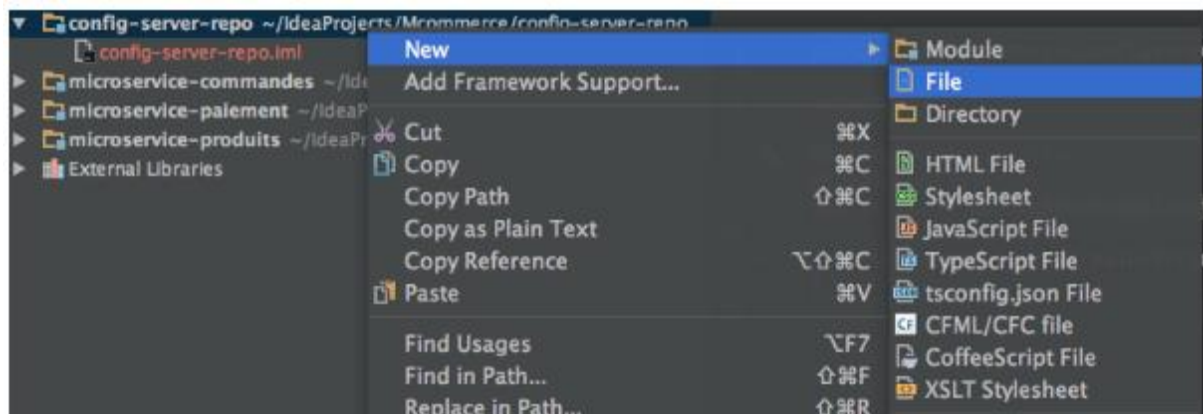


Continuez l'importation par défaut. Il n'y a pas besoin de choisir de dossier "source" ou "resources".

Le dossier apparaîtra alors à gauche dans votre projet à côté des autres Microservices.

L'utilité d'importer le dossier comme nouveau module dans notre projet IntelliJ est d'avoir le dossier sous les yeux et de pouvoir le manipuler depuis le projet. Vous pouvez tout à fait le laisser à l'extérieur du projet. Ce qui importe est le contenu du dépôt distant.

Créez maintenant un **nouveau fichier** dans ce dossier :



Appelez-le *microservice-produit.properties*.

Le nom doit correspondre **exactement** au nom que vous avez donné à votre Microservice-produits dans *application.properties*.

```
spring.application.name=microservice-produits
```

C'est grâce à cette correspondance de noms que notre serveur de configuration fera le lien entre ce fichier et le Microservice correspondant.

Coupez le contenu *d'application.properties* du Microservice-produits, excepté le nom de celui-ci, et collez le tout dans le nouveau fichier créé :

microservice-produits.properties

```
server.port 9001

#Configurations H2
spring.jpa.show-sql=true
spring.h2.console.enabled=true

#défini l'encodage pour data.sql
spring.datasource.sql-script-encoding=UTF-8

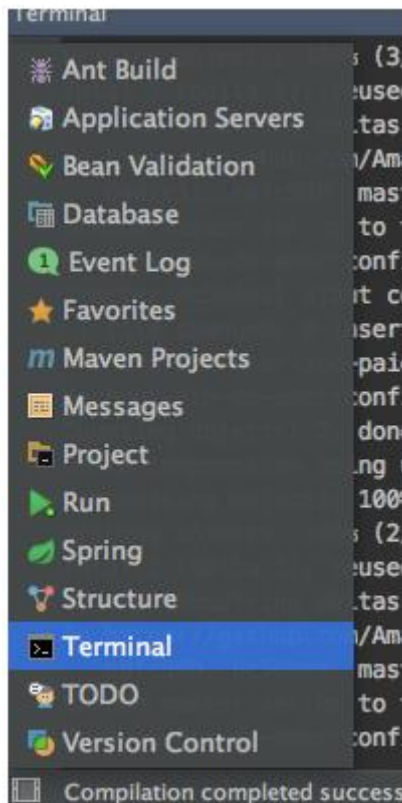
#Nos configurations

mes-configs.limitDeProduits= 4
```

Dans *application.properties* du Microservice-produits, il vous restera donc :

```
spring.application.name=microservice-produits
```

Rendez-vous au **terminal d'IntelliJ** :



terminal d'IntelliJ

Pointez vers le dossier créé :

```
cd config-server-repo/
```

Initialisez un nouveau dépôt GIT local que nous allons pousser plus tard vers le distant :

```
git init
```

Ajoutez les fichiers du dossier au GIT :

```
git add .
```

Ajoutez l'URL du dépôt distant à celui-ci :

```
git remote add origin https://NOM-UTILISATEUR:MOT-DE-PASSE@github.com/AmarMicroDev/mcommerce-config-repo.git
```

Veillez à faire les remplacements de nom d'utilisateur et de mot de passe, ainsi qu'à mettre votre propre URL à la place de la mienne.

Faites un commit du contenu :

```
git commit -m "Premier commit"
```

Poussez enfin le tout vers le dépôt distant :

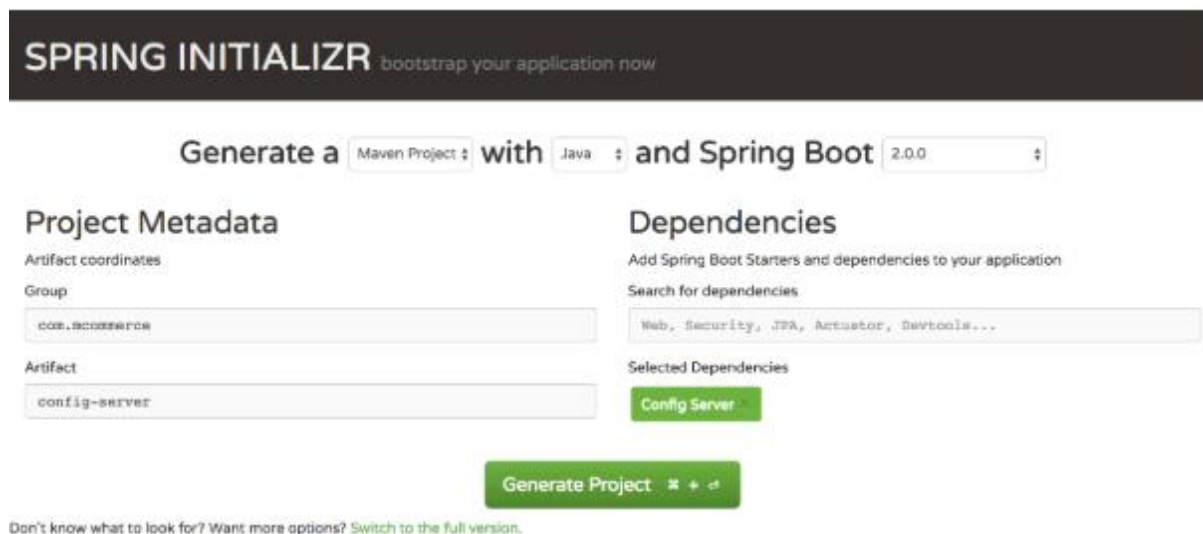
```
git push -u origin master
```

Vous avez maintenant toutes les configurations du Microservice-produits disponibles dans le dépôt distant. Maintenant, mettons en place le serveur de configuration.

4. Spring Cloud Config

Nous allons utiliser un **starter** afin de créer notre serveur de configuration.

Rendez-vous dans Spring Initializr, choisissez **Spring Cloud Config**, puis renseignez les champs comme suit :



The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below this is a header "Generate a" followed by a dropdown menu set to "Maven Project", then "with" followed by a dropdown menu set to "Java", and finally "and Spring Boot" followed by a dropdown menu set to "2.0.0".

Below the header, there are two main sections: "Project Metadata" and "Dependencies".

Project Metadata

Artifact coordinates

Group

com.example

Artifact

config-server

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

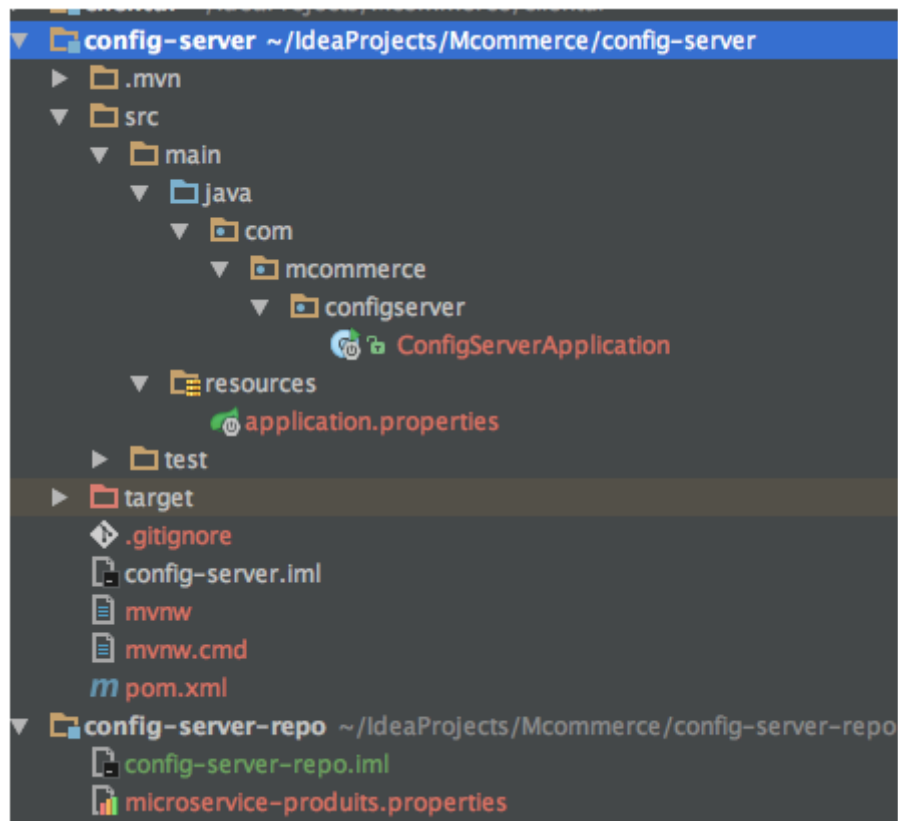
Config Server

Generate Project

Don't know what to look for? Want more options? [Switch to the full version.](#)

Téléchargez et extrayez le dossier dans celui de notre projet. Importez-le ensuite comme nouveau module, exactement comme vous l'avez fait pour les autres Microservices dans le Lab 1.

Vous obtenez alors cette arborescence :



Pour que notre serveur fonctionne, nous avons besoin de lui indiquer où aller chercher les fichiers de configuration pour les servir ensuite à nos Microservices.

Rendez-vous donc à *application.properties* et renseignez l'**URL de dépôt distant** :

```
spring.application.name=config-server
server.port:9101

spring.cloud.config.server.git.username=yourusername
spring.cloud.config.server.git.password=Yourpassword

spring.cloud.config.server.git.uri=https://github.com/AmarMicroDev/mcommerce-config-repo.git
```

Tous nos Edge Microservices seront sur des ports commençant par 91. Celui de *config-server* est donc 9101. N'oubliez pas de renseigner également son nom (*config-server*).

Il suffit maintenant de déclarer ce Microservice comme étant un serveur de configuration grâce à `@EnableConfigServer`.

ConfigServerApplication

```
package com.mcommerce.configserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
```



```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

Démarrez votre serveur Spring Cloud Config et rendez-vous à l'URL : <http://localhost:9101/microservice-produits/default>.

Vous obtenez alors :

```
{
  "name": "microservice-produits",
  "profiles": [
    "default"
  ],
  "label": null,
  "version": "6c360ea596e6d23a71eaa53788643c40ac22acdd",
  "state": null,
  "propertySources": [
    {
      "name": "https://github.com/AmarMicroDev/mcommerce-config-repo.git/microservice-produits.properties",
      "source": {
        "server.port": "9001",
        "spring.jpa.show-sql": "true",
        "spring.h2.console.enabled": "true",
        "spring.datasource.sql-script-encoding": "UTF-8",
        "mes-configs.limitDeProduits": "4"
      }
    }
  ]
}
```

Le serveur est donc allé chercher le fichier de configuration dans le GIT et expose une API qui répond à l'URL `"/nom-du-microservice/default"`. Il fournit ensuite sous format JSON toutes les configurations présentes dans le fichier.

5. Lier un Microservice à Spring Cloud Config

Nous avons un dépôt distant relié à notre serveur Spring Cloud Config. Il reste à demander au Microservice-produits de récupérer le contenu de ces fichiers de configuration depuis le serveur de configuration.

Pour ce faire, il suffit de **modifier *pom.xml* de Microservice-produits afin d'ajouter le starter *spring-cloud-starter-config*** :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mproduits</groupId>
```

```
<artifactId>mproduits</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>mproduits</name>
<description>Microservice de gestion des produits</description>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
  <spring-cloud.version>Finchley.M8</spring-cloud.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>

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

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
```

```

    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

  <repositories>
    <repository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

</project>

```

Ajouter au fichier `application.properties` la ligne :

```
spring.cloud.config.uri=http://localhost:9101
```

Renommez ensuite ***application.properties*** en ***bootstrap.properties***.

Lancez *config-server* et *Microservice-produits*. Vous verrez dans la console en première ligne :

```
Fetching config from server at: http://localhost:9101
```

Cette ligne indique que la première chose que fait votre Microservice est de récupérer la configuration via *server-config*.

Testez votre Microservice "produits" en récupérant la liste des produits. Vous devriez n'en recevoir que 4, comme configuré dans le dépôt Github.

6. Actualisez la configuration de votre Microservice à la volée

Nous avons un dernier problème à résoudre. Si vous changez la valeur de `mes-configs.limitDeProduits` et que vous réinvoquez */Produits*, vous remarquerez que rien ne change.

En effet, la configuration est chargée une seule fois au démarrage du Microservice, pour des raisons évidentes de performance.

Vous avez néanmoins la possibilité de demander à votre Microservice de s'actualiser et de recharger le fichier de configuration.

Pour ce faire, il faut lui envoyer un signal de "Refresh". Pour cela, nous devons **ajouter Spring Actuator** à notre Microservice qui nous exposera un URI */refresh* permettant de forcer la réactualisation des valeurs de configuration.

Ajoutez donc Spring Actuator dans le *pom.xml* de Microservice-produits :

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

Actualisez Maven.

Rendez-vous dans *bootstrap.properties* (ou dans le fichier correspondant dans le GIT) et ajoutez ceci :

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

Cela aura pour effet d'exposer tous les endpoints d'Actuator. Nous y reviendrons dans un prochain chapitre.

Nous allons maintenant indiquer à un de nos beans qui accède aux propriétés de se rafraîchir à chaque fois qu'un événement Refresh est lancé. Nous allons donc ajouter l'annotation `@RefreshScope` à *ApplicationPropertiesConfiguration.java* :

```
@Component
@ConfigurationProperties("mes-configs")
@RefreshScope
public class ApplicationPropertiesConfiguration {

    private int limitDeProduits;

    public int getLimitDeProduits() {
        return limitDeProduits;
    }

    public void setLimitDeProduits(int limitDeProduits) {
        this.limitDeProduits = limitDeProduits;
    }
}
```

Lancez tous les Microservices, puis appelez Microservice-produits : `localhost:9001/Produits`.

Changez la valeur de `mes-configs.limitDeProduits` dans le GIT, puis appelez de nouveau `localhost:9001/Produits`. Comme vous pouvez le constater, vous avez le même nombre de produits retournés. Cela indique que votre Microservice n'a pas pris en compte ce changement.

Déclenchez alors un événement *Refresh* en envoyant une requête POST à <http://localhost:9001/actuator/refresh> via Postman. Vous obtenez alors ce résultat :

```
[
  "config.client.version",
  "mes-configs.limitDeProduits"
]
```

Ce retour vous indique clairement que `mes-configs.limitDeProduits` a changé. Cela déclenche la réactualisation dans tous les beans annotés par `@RefreshScope`. Maintenant, si vous retentez `localhost:9001/Produits`, vous avez le bon nombre de produits en retour.

Le bean *ApplicationPropertiesConfiguration* s'est actualisé et a récupéré les nouvelles valeurs mises à jour dans le GIT. **Comme on accède à ces propriétés dans notre Microservice exclusivement via ce bean, cette valeur est mise à jour partout.**

Challenge : Essayez d'externaliser tous les fichiers de configuration des autres Microservices !

La branche pour retrouver tout le code de ce chapitre est **ExternalisationConfig**.

En résumé

- `application.properties` peut être utilisé pour stocker des constantes auxquelles on peut accéder grâce à un bean annoté avec `@ConfigurationProperties`.
- *Config-Server* permet de récupérer les fichiers de configuration dans un dépôt et de les servir aux Microservices en se basant sur leurs noms.
- Pour récupérer automatiquement ces configurations, il suffit d'ajouter le starter *spring-cloud-starter-config* et de renommer *application.properties* en *bootstrap.properties*.
- Pour actualiser à la volée la configuration d'un Microservice en l'obligeant à récupérer une version fraîche du fichier *.properties*, il suffit d'envoyer un POST vers l'endpoint */refresh* exposé par Spring Actuator.