Si vous préférez utiliser des fichiers `.properties` au lieu de `.yaml` pour la configuration, voici comment adapter le tutoriel précédent. Les fichiers `.properties` sont une alternative aux fichiers `.yaml` et sont couramment utilisés dans les projets Spring Boot.

Étape 1: Configuration du dépôt Git pour le service de configuration

1. **Créer un dépôt Git pour le service de configuration**:
   - Créez un nouveau dépôt Git (par exemple, `conf-repo`).
   - Ce dépôt contiendra les fichiers de configuration pour tous les microservices.

2. **Structure du dépôt `conf-repo`**:
   - Dans le dépôt `conf-repo`, créez un fichier de configuration pour chaque microservice. Par exemple:
     ```
     conf-repo/
     ├── service1.properties
     ├── service2.properties
     ├── service3.properties
     ├── eureka-server.properties
     └── gateway.properties
     ```

3. **Contenu des fichiers de configuration**:
   - **service1.properties**:
     ```properties
     server.port=8081
     spring.application.name=service1
     ```
   - **service2.properties**:
     ```properties
     server.port=8082
     spring.application.name=service2
     ```
   - **service3.properties**:
     ```properties
     server.port=8083
     spring.application.name=service3
     ```
   - **eureka-server.properties**:
     ```properties
     server.port=8761
     eureka.client.register-with-eureka=false
     eureka.client.fetch-registry=false
     ```
   - **gateway.properties**:
     ```properties
     server.port=8080
     spring.application.name=gateway
     spring.cloud.gateway.routes[0].id=service1
     spring.cloud.gateway.routes[0].uri=http://localhost:8081
     ```

```
spring.cloud.gateway.routes[0].predicates[0]=Path=/service1/**
spring.cloud.gateway.routes[1].id=service2
spring.cloud.gateway.routes[1].uri=http://localhost:8082
spring.cloud.gateway.routes[1].predicates[0]=Path=/service2/**
spring.cloud.gateway.routes[2].id=service3
spring.cloud.gateway.routes[2].uri=http://localhost:8083
spring.cloud.gateway.routes[2].predicates[0]=Path=/service3/**
```

Étape 2: Création du service de configuration (Spring Cloud Config Server)

1. **Créer un nouveau projet Spring Boot**:
   - Utilisez Spring Initializr pour créer un projet avec les dépendances suivantes:
     - `Config Server`
     - `Spring Web`

2. **Configurer le service de configuration**:
   - Dans `application.properties` du service de configuration, ajoutez:
     ```properties
     server.port=8888
     spring.application.name=config-server
     spring.cloud.config.server.git.uri=https://github.com/votre-utilisateur/conf-repo.git
     spring.cloud.config.server.git.clone-on-start=true
     ```

3. **Activer le serveur de configuration**:
   - Dans la classe principale, ajoutez l'annotation `@EnableConfigServer`:
     ```java
     @SpringBootApplication
     @EnableConfigServer
     public class ConfigServerApplication {
         public static void main(String[] args) {
             SpringApplication.run(ConfigServerApplication.class, args);
         }
     }
     ```

Étape 3: Création du service de découverte (Eureka Server)

1. **Créer un nouveau projet Spring Boot**:
   - Utilisez Spring Initializr pour créer un projet avec les dépendances suivantes:
     - `Eureka Server`
     - `Spring Web`

2. **Configurer le service de découverte**:
   - Dans `application.properties` du service de découverte, ajoutez:
     ```properties
     server.port=8761
     ```

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

3. **Activer le serveur Eureka**:
   - Dans la classe principale, ajoutez l'annotation `@EnableEurekaServer`:
   ```java
   @SpringBootApplication
   @EnableEurekaServer
   public class EurekaServerApplication {
       public static void main(String[] args) {
           SpringApplication.run(EurekaServerApplication.class, args);
       }
   }
   ```

Étape 4: Création des microservices simples

1. **Créer trois nouveaux projets Spring Boot**:
   - Utilisez Spring Initializr pour créer trois projets avec les dépendances suivantes:
     - `Eureka Discovery Client`
     - `Spring Web`
     - `Config Client`

2. **Configurer chaque microservice**:
   - Dans `bootstrap.properties` de chaque microservice, ajoutez:
   ```properties
   spring.application.name=service1 # ou service2, service3
   spring.cloud.config.uri=http://localhost:8888
   ```

3. **Créer un contrôleur simple pour chaque microservice**:
   - Par exemple, pour `service1`:
   ```java
   @RestController
   @RequestMapping("/service1")
   public class Service1Controller {
       @GetMapping("/hello")
       public String hello() {
           return "Hello from Service 1!";
       }
   }
   ```

Étape 5: Création de la gateway (Spring Cloud Gateway)

1. **Créer un nouveau projet Spring Boot**:
   - Utilisez Spring Initializr pour créer un projet avec les dépendances suivantes:

    - `Gateway`
    - `Eureka Discovery Client`
    - `Config Client`

2. **Configurer la gateway**:
   - Dans `bootstrap.properties` de la gateway, ajoutez:
     ```properties
     spring.application.name=gateway
     spring.cloud.config.uri=http://localhost:8888
     ```

3. **Activer la découverte Eureka dans la gateway**:
   - Dans la classe principale, ajoutez l'annotation `@EnableDiscoveryClient`:
     ```java
     @SpringBootApplication
     @EnableDiscoveryClient
     public class GatewayApplication {
         public static void main(String[] args) {
             SpringApplication.run(GatewayApplication.class, args);
         }
     }
     ```

Étape 6: Tester l'ensemble du système

1. **Démarrer les services dans l'ordre suivant**:
   - Config Server
   - Eureka Server
   - Les trois microservices (service1, service2, service3)
   - Gateway

2. **Accéder aux services via la gateway**:
   - Par exemple, pour accéder à `service1`, utilisez l'URL `http://localhost:8080/service1/hello`.

Différences entre `.yaml` et `.properties`

- **Syntaxe**:
  - `.yaml` utilise une syntaxe basée sur l'indentation.
  - `.properties` utilise une syntaxe clé-valeur séparée par `=`.

- **Avantages de `.properties`**:
  - Plus simple pour les configurations simples.
  - Plus facile à lire pour les développeurs habitués à cette syntaxe.

- **Avantages de `.yaml`**:
  - Plus lisible pour les configurations complexes.
  - Supporte les structures de données imbriquées.

Avec cette adaptation, vous pouvez maintenant utiliser des fichiers `.properties` pour configurer vos microservices tout en conservant la même architecture et les mêmes fonctionnalités.