Config Server Guide

Profiles Concepts

Spring Profiles are a feature in the Spring Framework that allows you to define different configurations for different environments (such as development, testing, and production) and switch between these environments easily. By using profiles, you can manage application behavior or configuration values based on where the application is running.

Here's a breakdown of the key concepts of Spring Profiles:

1. What Are Spring Profiles?

Spring Profiles are a mechanism that allows developers to create and maintain separate configurations for different environments, such as:

- **dev**: For development.
- **test**: For testing.
- **prod**: For production.

Each environment can have its own set of beans, properties, and configurations, and Spring will load the appropriate configuration based on the active profile.

2. How to Define Profiles in Spring?

Profiles are typically defined in two main ways:

- **Annotation-based**: Using `@Profile` annotation to specify which beans or components should be loaded for which profiles.
- **Property-based**: Using different properties files or YAML configurations for different environments.

Example of `@Profile` Annotation:

```
@Configuration
public class DataSourceConfig {
```

```
    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        // Configuration for dev
        return new H2DataSource();
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        // Configuration for production
        return new OracleDataSource();
    }
}
```

In this example, two `DataSource` beans are defined for different profiles. The `devDataSource` will be used when the "dev" profile is active, and `prodDataSource` will be used when the "prod" profile is active.

3. How to Activate Profiles?

You can activate Spring profiles in several ways:

- **In application.properties or application.yml**:
  You can specify active profiles in your configuration file.

  ```
  spring.profiles.active=dev
  ```

  Or in `application.yml`:

  ```
  spring:
    profiles:
      active: dev
  ```

- **Through command-line arguments**:
  When starting the application, you can pass a profile as an argument:

  ```
  java -jar myapp.jar --spring.profiles.active=prod
  ```

- **Environment variables**:
  You can also set profiles using environment variables, such as in a containerized environment or cloud service.

4. Profile-Specific Configuration Files

Spring allows you to create profile-specific configuration files. For example, you can have different `application.properties` or `application.yml` files for each environment:

- `application-dev.properties`
- `application-prod.properties`

Spring automatically loads the configuration file corresponding to the active profile.

5. Default Profile

Spring allows the use of a **default profile** if no profile is explicitly activated. This can be specified using:

```
spring.profiles.default=dev
```

6. Multiple Profiles

You can activate multiple profiles at the same time, which is useful when you want to layer configurations. For example:

```
spring.profiles.active=dev,debug
```

In this case, Spring will combine the beans and properties of both `dev` and `debug` profiles.

7. Conditional Beans Based on Profiles

You can define beans that should only be loaded when certain profiles are active. This is useful for loading environment-specific beans like different databases, services, or logging configurations.

Example:

```
@Profile("test")
@Bean
public MyService testMyService() {
    return new TestMyService();
}
```

8. Testing with Profiles

Spring also supports profiles for testing environments. You can activate profiles within test classes by using the `@ActiveProfiles` annotation.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@ActiveProfiles("test")
public class MyServiceTest {
    // Test cases go here
}
```

9. Priority in Profiles

When multiple profiles are active or multiple configuration files exist, Spring loads the properties in a specific order:

1. `application.properties` or `application.yml`
2. Profile-specific files like `application-dev.properties` or `application-prod.yml`

3. Command-line arguments or environment variables have the highest precedence.

10. Use Cases for Spring Profiles

- **Development vs. Production**: Different database configurations (e.g., H2 in-memory for development, PostgreSQL for production).
- **Testing**: Use mock services or in-memory databases for unit testing with specific profiles.
- **Cloud Configurations**: Enable different cloud-specific properties or beans when running the application in a cloud environment.

Spring Profiles simplify environment-specific configuration management, making your application flexible and easier to maintain across different stages of development and deployment.

Would you like a more detailed example or additional information on any specific part?

Externalizing the configurations using command line argument

```
java -jar app.java --spring.profiles.active=prod
```

Using JVM system properties

```
java -jar app.java -Dspring.profiledactive=prod
```

Using Environment Variables

1. Write properties in capital letters with separated by underscore

```
java -jar app.java SPRING_PROFILES_ACTIVE=prod
```

Drawback of externalizing the configurations using spring boot framework alone

1. Manually setting variables. For thousands of services is very challenging.
2. In thousands of services we will have more configurations files. and all file can contain configurations
3. Environment variables can be seen by server admin.
4. Multiple instances of same mciroservices so for each doing manual configurations is challenging.
5. Security: passing argument as plain text is security issue
6. Read latest property value dynamic.

Spring Cloud Config

1. Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system
2. With the Config Server, you have a central place to manage external properties for applications across all environments
3. Store configuration in any location github, database, local files etc.

📌 Complete Guide to Implementing Spring Cloud Config Server

🔷 Step 1: Set Up a Spring Cloud Config Server

1️⃣ Create a Spring Boot Application for Config Server

Use **Spring Initializr** or manually create a new Spring Boot project.

Add Dependencies in `pom.xml`

```
<dependencies>
    <!-- Spring Boot Config Server -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>

    <!-- Spring Boot Actuator for Monitoring -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

Enable Config Server (`ConfigServerApplication.java`)

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

📌 Step 2: Storing Configurations in Classpath

- Store configurations inside `src/main/resources/config/` in the Config Server.

🔷 Modify `application.yml` in Config Server

```
server:
  port: 8888

spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/config
```

🔷 Create Configuration Files Inside `src/main/resources/config/`

Example:

```
src/main/resources/config/
    ├── application.yml
    ├── food-service.yml
    ├── food-service-dev.yml
```

🔷 Example `food-service.yml`

```
server:
  port: 8081
spring:
  application:
    name: food-service
message: "Hello from Classpath Config"
```

✅ Start Config Server

```
mvn spring-boot:run
```

✅ Test Configuration Fetch

```
curl http://localhost:8888/food-service/default
```

📌 Step 3: Storing Configurations in Local File System

To store configurations **outside the JAR**, use a local folder.

🔷 Modify `application.yml` in Config Server

```
spring:
  cloud:
    config:
      server:
        native:
          search-locations: file:/path/to/local/configs
```

🔷 Create Configurations in `/path/to/local/configs/`

```
mkdir -p /path/to/local/configs
cd /path/to/local/configs
```

Create a file `food-service.yml`:

```
server:
  port: 8081
spring:
  application:
    name: food-service
message: "Hello from Local Config"
```

✅ **Restart Config Server and Test**

```
mvn spring-boot:run
```

```
curl http://localhost:8888/food-service/default
```

📌 Step 4: Storing Configurations in GitHub

Using **GitHub** ensures centralized versioning.

🔷 Modify `application.yml` in Config Server

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/yourusername/config-repo
          clone-on-start: true
```

🔷 Create a Public or Private GitHub Repository

1. Create a **GitHub repo** named `config-repo`
2. Inside the repo, add config files:

```
config-repo/
    ├── application.yml
    ├── food-service.yml
    ├── food-service-dev.yml
```

✅ Restart Config Server and Test

```
curl http://localhost:8888/food-service/default
```

📌 Step 5: Encrypting Sensitive Properties

For security, encrypt sensitive data (e.g., database credentials).

🔷 Add Spring Cloud Encryption Dependency

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

🔷 Enable Encryption in Config Server (`application.yml`)

```
encrypt:
```

```
  key: my-secret-key
```

🔷 Encrypt a Property

Use:

```
curl -X POST http://localhost:8888/encrypt -d "my-secret-password"
```

Response:

```
{cipher}AQASDF12345HGG==
```

🔷 Store the Encrypted Property in Config File

```
db:
  password: "{cipher}AQASDF12345HGG=="
```

✅ **Restart Config Server and Test**

📌 Step 6: Automatic Refresh of Configurations

By default, microservices **DO NOT automatically reload configurations** when updates occur in the Config Server.

🔷 Enable Spring Cloud Bus with RabbitMQ

1️⃣ Add Dependencies in Microservices (`food-service` `pom.xml`)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

2️⃣ Enable Actuator for Refreshing

Modify `application.yml` of `food-service`:

```
management:
  endpoints:
    web:
      exposure:
        include: refresh, bus-refresh
```

3️⃣ Trigger Refresh Using Spring Cloud Bus

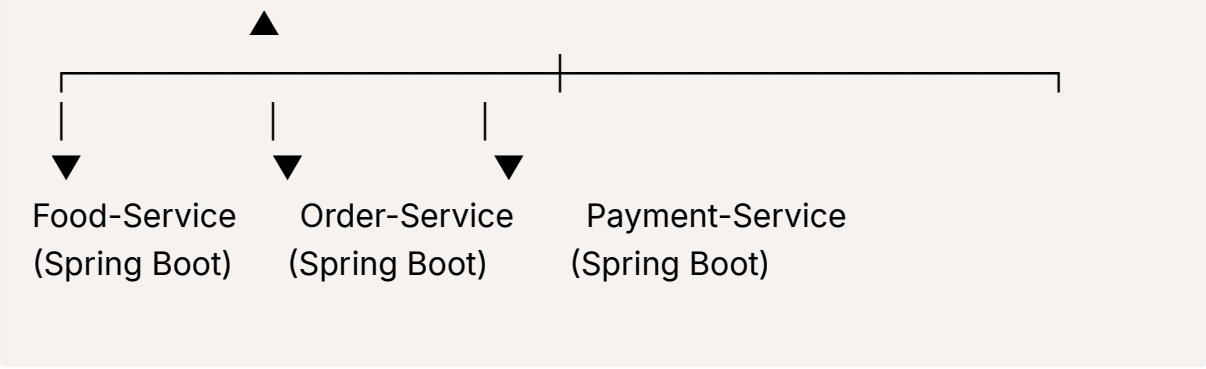
Run:

```
sh
CopyEdit
curl -X POST http://localhost:8888/actuator/bus-refresh
```

✅ Now, all microservices **automatically update configurations** when changed in the Config Server!

📌 Final Architecture

```
┌──────────────┐
│ Config Server │
└──────────────┘
```



📌 Summary of All Steps

Step	Action
1	Set up Config Server
2	Store configurations in `classpath:/config`
3	Store configurations in local file system (`file:/path/to/configs`)
4	Store configurations in GitHub (`git.uri`)
5	Encrypt properties using `{cipher}`
6	Enable Spring Cloud Bus for automatic refresh

🔷 Do We Need to Install RabbitMQ for Spring Cloud Config?

Yes, but only if you want to enable automatic refresh using Spring Cloud Bus.

If you are using **Spring Cloud Config without** automatic refresh, **RabbitMQ is NOT required.**

However, if you want **Spring Cloud Bus** to automatically notify services when configuration changes, **RabbitMQ is required.**

📌 When is RabbitMQ Required?

Feature	Requires RabbitMQ?	Alternative
Basic Config Server (Fetching Configs)	❌ No	Manual refresh (`/actuator/refresh`)
Dynamic Refresh (`/actuator/refresh`)	❌ No	Manual HTTP call

Automatic Refresh with Spring Cloud Bus	✅ **Yes**	Kafka (Alternative to RabbitMQ)

📌 Installing RabbitMQ for Spring Cloud Bus

If you want **automatic refresh**, install **RabbitMQ**.

1️⃣ Install RabbitMQ Locally

🔷 Option 1: Install via Docker (Recommended)

```
docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management
```

- **5672** → RabbitMQ default port for communication
- **15672** → RabbitMQ management UI

✅ **Check RabbitMQ is Running**:

- Open `http://localhost:15672/`
- Login: `guest` / `guest`

🔷 Option 2: Install RabbitMQ Manually

For Windows/macOS/Linux:

- Download RabbitMQ from: https://www.rabbitmq.com/download.html
- Start RabbitMQ:

```
rabbitmq-server
```

✅ **Verify RabbitMQ is running**:

```
rabbitmqctl status
```

📌 Step-by-Step Integration of Spring Cloud Bus with RabbitMQ

2️⃣ Add Dependencies (`food-service/pom.xml`)

```
xml
CopyEdit
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

3️⃣ Configure RabbitMQ in `application.yml`

Add this to **all microservices (** `food-service` **,** `order-service` **, etc.)** and **Config Server**:

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
  cloud:
    bus:
      enabled: true
```

4️⃣ Enable Actuator Refresh in Microservices

In **all microservices** (`food-service`, etc.), enable refresh endpoints:

```
management:
  endpoints:
    web:
      exposure:
        include: refresh, bus-refresh
```

5️⃣ Trigger Automatic Refresh

After updating a config file in **GitHub/Local**, send a refresh event:

```
curl -X POST http://localhost:8888/actuator/bus-refresh
```

✅ **This will notify all connected microservices automatically!** 🎯

📌 What Happens Internally?

1. Config is updated in **GitHub** or **Local Files**.
2. You trigger `/actuator/bus-refresh` on **Config Server**.
3. Config Server sends a message to **RabbitMQ**.
4. **RabbitMQ** broadcasts the message to **all microservices**.
5. **Microservices refresh their configurations dynamically** without restarting.

📌 Alternative to RabbitMQ

If you don't want RabbitMQ, use **Apache Kafka** instead:

```
xml
CopyEdit
<dependency>
```

```
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>
```

Then configure:

```
spring:
  cloud:
    bus:
      enabled: true
  kafka:
    bootstrap-servers: localhost:9092
```

✅ **But Kafka requires setting up Zookeeper + Kafka, which is more complex than RabbitMQ.**

📌 Summary

Feature	Requires RabbitMQ?	Alternative
Fetching Configurations	❌ No	Direct HTTP call
`/actuator/refresh` (Manual Refresh)	❌ No	Manual refresh
Automatic Refresh (`/bus-refresh`)	✅ **Yes**	Kafka

✅ **If you need automatic updates in microservices, install RabbitMQ.**

❌ **If you don't need auto-refresh, just manually call `/actuator/refresh` when needed.**

Refresh Configurations using git-hub web-hook

1. Add new dependency in Cloud config server
    a. Spring Cloud Config Monitor(in config server)
    b. it expose /monitor → now we can monitor webhook

2. Expose actuator api

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

3. Mention rabbitmql configuration in config server

```
spring:
   rabbitmq:
      host: "localhost"
      port: 5672
      username: "guest"
      password: "guest"
```

4. Create web hook

   Settings>Webhooks>AddWebhooks>

   1. Provide monitor url
   2. use public url[]
   3. use hookdeck.com [Configure]