Configuration Management

Centralizing configuration every service has kept in its own application.yml so far, using Spring Cloud Config: setting up a separate config-server application with @EnableConfigServer, a Config Repository deciding what's genuinely worth centralizing, becoming a Config Client with spring.config.import, environment-specific overrides through profiles, refreshing configuration without a restart via @RefreshScope, and why secrets should never be stored in plain text.

Intermediate 24 min

Last updated: August 23, 2026
Read online: https://www.learnforgex.com/en/topics/configuration-management

---

Configuration Management

Every service in this course has kept its own configuration in its own `application.yml` -- order-service's port and datasource settings (see the Spring Boot Microservice Basics lesson's "Its Own `application.yml`: Port, Application Name, and Database" section), its Eureka client settings (see the Service Discovery & Eureka lesson), and now its Resilience4j instances (see the Resilience4j lesson). That's fine for a handful of services -- but imagine twenty services all needing the SAME datasource pool-size tuning, or a value that needs to change ACROSS every one of them at once. Copy-pasting the same block into twenty `application.yml` files, and editing all twenty when it changes, doesn't scale. This lesson introduces the piece that centralizes configuration instead: Spring Cloud Config.

What Is Configuration Management?

Configuration management, in the microservices sense, is keeping configuration OUTSIDE the applications that use it, in one central place, instead of duplicating it inside every service's own packaged code. A Config Server serves configuration OVER THE NETWORK to any service that asks for it by name, on startup (and, as this lesson covers, sometimes without even a restart).

Why Does It Exist?

Configuration duplicated across many services creates two real problems: first, a value that's genuinely SHARED (a connection pool size, a feature flag, a third-party API's base URL) has to be updated in every single service's own file when it changes -- easy to miss

one, and each service needs its own redeploy just to pick up a value that never touched its actual code. Second, some configuration needs to be DIFFERENT per environment (a database URL in staging vs. production) without duplicating the entire file for each one. Centralizing configuration in one server, with support for per-environment overrides (see "Profiles: Different Configuration for Different Environments"), solves both.

History

Spring Cloud Config was one of the original Spring Cloud projects, released alongside Eureka and Zuul around 2015 (see the Service Discovery & Eureka and API Gateway lessons' "History" sections) as part of the same wave of Netflix-OSS-inspired tooling brought into the Spring ecosystem. Unlike Eureka, it isn't built on any Netflix library -- it's a Spring-native answer to a problem every distributed system eventually runs into, and it remains actively maintained within Spring Cloud today.

Setting Up a Config Server

config-server, like eureka-server, is its OWN independent Spring Boot application -- a fifth one in this course, alongside order-service, inventory-service, eureka-server, and api-gateway.

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

// A FIFTH Spring Boot application in this course, next to order-service,
// inventory-service, eureka-server, and api-gateway -- config-server, like
// eureka-server, is pure infrastructure: no business logic, no domain database.
// Its only job is serving CONFIGURATION files to every other service in the
// system (see "Setting Up a Config Server").
//
// @EnableConfigServer is the ONE annotation that turns a plain Spring Boot app
// into a Config Server -- exactly the same shape as @EnableEurekaServer on
// EurekaServerApplication (see the Service Discovery & Eureka lesson's "Eureka
// Server: A Central Registry" section).
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

```
# config-server's own application.yml -- runs as its own independent process,
# on its own port, exactly like eureka-server (see the Service Discovery &
# Eureka lesson) and api-gateway (see the API Gateway lesson).

server:
  port: 8888                          # Config Server's traditional default
                                       # port, by the same convention Eureka
                                       # follows with 8761

spring:
  application:
    name: config-server
  profiles:
    active: native                    # "native" backend: reads configuration
                                       # files from the local filesystem/
                                       # classpath, NOT from a Git repository --
                                       # the simplest backend to reason about
                                       # while learning (see "The Config
                                       # Repository: Where Configuration
                                       # Actually Lives" for the real-world
                                       # Git-backed alternative)
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/config-repo/   # where the actual
                                                       # order-service.yml,
                                                       # inventory-service.yml
                                                       # etc. files live
```

> This lesson uses Config Server's "native" backend (a local filesystem/classpath directory) instead of a Git repository, to keep the example self-contained -- in production, `spring.cloud.config.server.git.uri` pointing at a real Git repo is far more common, since it gives configuration changes the same version history and review process as code.

The Config Repository: Where Configuration Actually Lives

The actual configuration values -- one YAML file per service, named to match that service's `spring.application.name` -- live in the Config Repository, completely separate from any service's own packaged code.

```
# This file lives INSIDE config-server's own repository (at
# classpath:/config-repo/order-service.yml, matching ConfigServerConfig.yml's
# search-locations) -- NOT inside order-service itself. Config Server serves
# this content to order-service over HTTP when it asks for configuration
# named "order-service" (see "Making order-service a Config Client").
#
# Notice this is a SUBSET of what used to live entirely inside order-service's
# own application.yml (see the Spring Boot Microservice Basics lesson's "Its
# Own `application.yml`: Port, Application Name, and Database" section) --
# server.port and spring.application.name stay LOCAL to order-service (a
# service needs to know its own port before it can even ask Config Server for
# anything else), only the parts worth centralizing move here.

greeting:
  message: "Welcome to the Order Service"   # the property RefreshableGreetingController
                                             # exposes -- changing THIS file and
                                             # refreshing (see "Refreshing
                                             # Configuration Without Restarting")
                                             # changes the running service's
                                             # behavior with no restart at all

datasource:
  max-pool-size: 10                         # exactly the kind of tuning value that's
                                             # genuinely worth sharing/centralizing
                                             # across every service using the same
                                             # database technology
```

Notice this file only contains what's genuinely worth CENTRALIZING -- `server.port` and `spring.application.name` stay in order-service's own local `application.yml`, since a service needs to know its own identity and port before it can even ASK Config Server for anything else.

Making order-service a Config Client

A single property, added to order-service's EXISTING `application.yml`, is what makes it fetch and merge in configuration from config-server on startup.

```
# Added to order-service's own application.yml -- on top of everything from
# the Spring Boot Microservice Basics lesson, plus the eureka.client block
# from Service Discovery & Eureka. Nothing already there is REMOVED; this is
# what makes order-service ask config-server for MORE configuration on
# startup, in addition to what it already knows locally.

spring:
  config:
    import: "configserver:http://localhost:8888"   # on startup, order-service
                                                     # fetches config named after
                                                     # its own spring.application.name
                                                     # ("order-service") from this
                                                     # URL -- Spring Cloud Config
                                                     # matches the request to
                                                     # config-repo/order-service.yml
                                                     # automatically
```

Profiles: Different Configuration for Different Environments

A Config Repository file can be split further by PROFILE -- `order-service-staging.yml` and `order-service-production.yml`, alongside the base `order-service.yml` above, override or add to it depending on which profile order-service is started with (`spring.profiles.active=staging`, for instance). This is the SAME profile mechanism Spring Boot already uses locally (`application-dev.yml` overriding `application.yml`, if this project's own `application-dev.yml`/`application-prod.yml` pattern looks familiar) -- Config Server just applies it to centrally-hosted files instead of local ones.

Refreshing Configuration Without Restarting: @RefreshScope

By default, a `@Value`-injected property is read exactly once, when Spring creates the bean -- editing a Config Repository file afterward has NO effect on an already-running service. `@RefreshScope` changes that: it makes a bean eligible to be thrown away and recreated, re-reading its `@Value`s, whenever a refresh is triggered (a `POST` to that service's own `/actuator/refresh` endpoint).

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

// Without @RefreshScope, a @Value-injected property is read EXACTLY ONCE, when
// this bean is first created -- editing config-repo/order-service.yml's
// greeting.message afterward would have NO effect on an already-running
// order-service; a full restart would be the only way to pick it up.
// @RefreshScope makes Spring throw this bean away and recreate it (re-reading
// every @Value) whenever a refresh is triggered (see "Refreshing Configuration
// Without Restarting: @RefreshScope") -- the endpoint below always reflects
// whatever config-server is currently serving, without order-service ever
// stopping.
@RestController
@RefreshScope
class RefreshableGreetingController {

    @Value("${greeting.message}")
    private String greetingMessage;

    // GET /greeting -- returns whatever greeting.message currently is. Change
    // OrderServiceExternalConfig.yml, POST to order-service's own
    // /actuator/refresh, and call this again: the response changes, with
    // order-service never having restarted.
    @GetMapping("/greeting")
    String greeting() {
        return greetingMessage;
    }
}
```

> A single `/actuator/refresh` call only refreshes the ONE service you called it on -- with many services, triggering every one of them by hand doesn't scale much better than editing files by hand did. Spring Cloud Bus (broadcasting a refresh event to every service at once over a shared message broker) solves this, but is out of scope for this lesson -- worth knowing it exists once a system has more than a couple of services.

Secrets: What Config Server Should NOT Store in Plain Text

Not everything belongs in a Config Repository file as plain text -- a database password is configuration in the sense that it varies by environment, but storing it unencrypted in a file (even a private Git repo) is a real security risk. Spring Cloud Config supports encrypting individual values, and dedicated secrets tools (HashiCorp Vault is the most common pairing) exist specifically for this -- this lesson keeps using `${ORDERS_DB_PASSWORD}` as an environment variable (see the Spring Boot Microservice Basics lesson) for the genuinely sensitive value, and only centralizes configuration that ISN'T a secret.

Best Practices

- **Centralize values that are genuinely SHARED or that change independently of code** (pool sizes, feature flags, third-party URLs) -- leave a service's own identity (port, application name) in its local `application.yml`.
- **Use profiles for environment-specific overrides**, instead of maintaining separate near-duplicate files by hand.
- **Mark beans that need live updates with `@RefreshScope` deliberately, not everywhere** -- refreshing has a real cost (bean recreation), and most beans never need it.
- **Never put secrets in a Config Repository file as plain text** -- use encryption or a dedicated secrets tool, exactly as this lesson keeps `ORDERS_DB_PASSWORD` as an environment variable rather than centralizing it.

Common Mistakes

- **Centralizing `server.port` or `spring.application.name`.** A service needs both LOCALLY, before it can even contact Config Server to ask for anything else.
- **Editing a Config Repository file and expecting a running service to pick it up immediately.** Without `@RefreshScope` and an explicit refresh call (or Spring Cloud Bus), nothing changes until the next restart.
- **Storing a database password or API key in a Config Repository file as plain text.** Even in a private repository, this is a real credential leak waiting to happen -- see "Secrets" above.
- **Applying `@RefreshScope` to every bean "just in case."** It adds real overhead to bean creation and complicates reasoning about a bean's lifecycle, for beans that in practice never change at runtime.

Summary, Cheat Sheet, and Glossary

Configuration management centralizes configuration OUTSIDE the services that use it. Spring Cloud Config's Config Server (`@EnableConfigServer`) serves per-service YAML files from a Config Repository (a Git repo in production, a local directory here); a service becomes a Config Client with a single `spring.config.import` property. Profiles let a Config Repository file be overridden per environment, the same way this project's own `application-dev.yml` / `application-prod.yml` already work locally. `@RefreshScope` lets a bean pick up new configuration without a restart, triggered by `/actuator/refresh`. Secrets never belong in a Config Repository file as plain text.

Quick reference:

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication { ... }   // its own Spring Boot app,
                                                // serves configuration only

// order-service's application.yml
// spring.config.import: "configserver:http://localhost:8888"

@RestController
@RefreshScope                                  // re-reads @Value on refresh,
class SomeController {                         // instead of only at startup
    @Value("${greeting.message}")
    private String greetingMessage;
}
```

Glossary

Config Server — A Spring Boot application, enabled with `@EnableConfigServer`, that serves per-service configuration over the network.

Config Repository — Where the actual configuration files live -- a Git repository in production, a local directory in this lesson's example.

Config Client — A service that fetches and merges in configuration from a Config Server on startup, via `spring.config.import`.

Profile — A named variant of configuration (e.g. staging, production) that overrides or extends a base configuration file.

`@RefreshScope` — An annotation that makes a bean's `@Value`-injected properties re-read whenever a refresh is triggered, instead of only once at startup.