Configuration Management in Microservices

# Intro

As we know from the 15 factors of cloud-native applications, we need to externalized the configuration and we have to follow the dev/prod parity principle.

We need to first externalized the configurations and inject them into the app at run time.

We also need to maintain these configurations and deal with the injection of sensitive information like credentials.

We will discuss the options available in configuring spring boot applications and finally using a config server we will subscribe to the principles of cloud-native applications.

# Configuration in Spring boot Applications

You can use a variety of external configuration sources including Java properties files, YAML files, environment variables, and command-line arguments.

Property values can be injected directly into your beans by using the @Value annotation, accessed through Spring’s **Environment abstraction**, or be [bound to structured objects](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.typesafe-configuration-properties) through @ConfigurationProperties.

Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. **Later property sources can override the values defined in earlier ones.** Sources are considered in the following order:

1. Default properties (specified by setting SpringApplication.setDefaultProperties).
2. [@PropertySource](https://docs.spring.io/spring-framework/docs/6.1.x/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotations on your @Configuration classes. Please note that such property sources are not added to the Environment until the application context is being refreshed. This is too late to configure certain properties such as logging.\* and spring.main.\* which are read before refresh begins.
3. **Config data (such as application.properties files).**
4. A RandomValuePropertySource that has properties only in random.\*.
5. **OS environment variables.**
6. **Java System properties (System.getProperties(). They are per-process in JVM not the entire JVM).**
7. **JNDI attributes from java:comp/env.**
8. **ServletContext init parameters.**
9. **ServletConfig init parameters.**
10. **Properties from SPRING\_APPLICATION\_JSON (inline JSON embedded in an environment variable or system property).**
11. **Command line arguments.**
12. **properties attribute on your tests. Available on**[**@SpringBootTest**](https://docs.spring.io/spring-boot/api/java/org/springframework/boot/test/context/SpringBootTest.html)**and the**[**test annotations for testing a particular slice of your application**](https://docs.spring.io/spring-boot/reference/testing/spring-boot-applications.html#testing.spring-boot-applications.autoconfigured-tests)**.**
13. [**@DynamicPropertySource**](https://docs.spring.io/spring-framework/docs/6.1.x/javadoc-api/org/springframework/test/context/DynamicPropertySource.html)**annotations in your tests.**
14. [**@TestPropertySource**](https://docs.spring.io/spring-framework/docs/6.1.x/javadoc-api/org/springframework/test/context/TestPropertySource.html)**annotations on your tests.**
15. [Devtools global settings properties](https://docs.spring.io/spring-boot/reference/using/devtools.html#using.devtools.globalsettings) in the $HOME/.config/spring-boot directory when devtools is active.

Config data files are considered in the following order:

1. [Application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files) packaged inside your jar (application.properties and YAML variants).
2. [Profile-specific application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files.profile-specific) packaged inside your jar (**application-{profile}.properties** and YAML variants).
3. [Application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files) outside of your packaged jar (application.properties and YAML variants).
4. [Profile-specific application properties](https://docs.spring.io/spring-boot/reference/features/external-config.html#features.external-config.files.profile-specific) outside of your packaged jar (application-{profile}.properties and YAML variants).

## How to inject properties into your business code

### Using @Value

@Component

public class MyBean {

@Value("${name}")

private String name;

*// ...*

}

Spring boot is going to look for this property in the above property sources and if there is multiple sources providing this property, it will be overridden according to the specified order. For example:

On your application classpath (for example, inside your jar) you can have an application.properties file that provides a sensible default property value for name. When running in a new environment, an application.properties file can be provided outside of your jar that overrides the name. For one-off testing, you can launch with a specific command line switch (for example**, java -jar app.jar --name="Spring").**

### Using Environment

There is a Bean of type Environment that I think Spring boot will put all the properties inside it and you can inject this bean and access the properties through it. We skip this part for now because we’re more interested in cloud config approach

### Using @ConfigurationProperties

We skip this part for now because we’re more interested in cloud config approach

### There could be More Options and more to discuss about this subject but we’ll skip them for now.

# Draw Backs of Externalized Configuration Using Spring Boot Alone