UDC 004.428.4

REVIEW OF SPRING CLOUD CONFIG

RAMAN KHRAPAVITSKI, YAUHEN SUKHAREU
Polotsk State University, Belarus

This article focuses on Spring Cloud Config, which is a library of Spring Framework. A short overview of functionality and capabilities of this library is presented.

Introduction. Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the ConfigServer you have a central place to manage external properties for applications across all environments. The concepts on both client and server map are identical to Spring Environment and PropertySource abstractions, so they fit Spring applications very well, but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git so it easily supports labeled versions of configuration environments, as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

Changing configurations on running services can be cumbersome, especially if the application does not have a way to be configured remotely. Spring Cloud Config provides a way to make this easier by introducing a Config Server from which clients fetch their configuration. The figure below illustrates how Spring Cloud Config works.

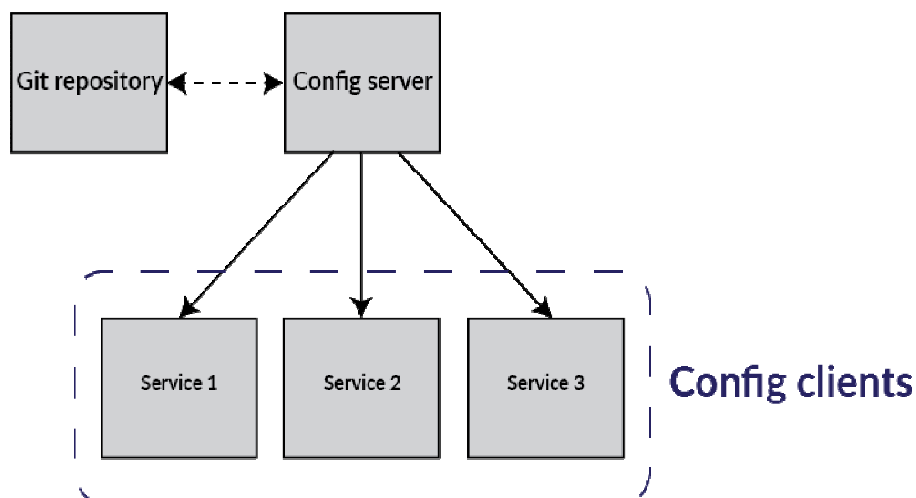

Fig. 1 – Common architecture diagram of how Spring cloud config works

Spring Cloud Config Server provides the following features:
1. HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content)
2. Encrypt and decrypt property values (symmetric or asymmetric)
3. Embeddable easily in a Spring Boot application using @EnableConfigServer.

Config Client provides the following features (for Spring applications):
1. Bind to the Config Server and initialize Spring Environment with remote property sources
2. Encrypt and decrypt property values (symmetric or asymmetric)
3. Fail fast and retry

The server is a Spring Boot application so it is very easy to create stand-alone, production-grade Spring based applications that can be "just run".

The default strategy for locating property sources is to clone a git repository (at spring.cloud.config.server.git.uri) and use it to initialize a mini SpringApplication. The Spring boot applications environment is used to enumerate property sources and publish them via a JSON endpoint.

The HTTP service has the following resources forms:

1. /{application}/{profile}[/{label}]
2. /{application}-{profile}.yml
3. /{label}/{application}-{profile}.yml
4./{application}-{profile}.properties
5./{label}/{application}-{profile}.properties

"Application" is injected as the spring.config.name in the SpringApplication (i.e. what is normally "application" in a regular Spring Boot app), "profile" is an active profile (or comma-separated list of properties), and "label" is an optional git label (defaults to "master".)

Also you should create bootstrap.properteies file and add it to classpath of your spring boot application. Then you just need to add one property to this file: spring.cloud.config.server.git.uri: {pathToGitRepository}.

That is all, server side of Spring Cloud Config is ready to be used.

Now let's look at client side of Spring Cloud Config. To use Spring Cloud Config features in an application, just build it as a Spring Boot application that depends on spring-cloud-config-client (e.g. see the test cases for the config-client, or the sample app). The most convenient way to add the dependency is via a Spring Boot starter org.springframework.cloud:spring-cloud-starter-config. Then you just need to add one property to your client of Spring Cloud Config: spring.cloud.config.uri: http://myconfigserver.com.

That is all. When Cloud Config client will start it automatically and fetch properties from Cloud Config server, all other work for injecting properties to your beans Spring does by itself.

You are free to secure your Config Server in any way that makes sense to you (from physical network security to OAuth2 bearer tokens), and Spring Security and Spring Boot make it easy to do a lot of things.

If the remote property sources contain encrypted content, (values starting with {cipher}) they will be decrypted before sending to clients over HTTP. The main advantage of this set up is that the property values do not have to be in plain text when they are "at rest" (e.g. in a git repository). If a value cannot be decrypted, it is removed from the property source and an additional property is added with the same key, but prefixed with "invalid." and a value that means "not applicable" (usually "<n/a>"). This is largely to prevent cipher text being used as a password and leaking accidentally.

To use the default Spring Boot configured HTTP Basic security, just include Spring Security on the classpath (e.g. through spring-boot-starter-security). The default is a username of "user" and a randomly generated password, which isn't very useful in practice, so we recommend you configure the password (via security.user.password) and encrypt it (see below for instructions on how to do that).

In some cases, it may be desirable to fail startup of a service if it cannot connect to the Config Server. If this is the desired behaviour, set the bootstrap configuration property spring.cloud.config.failFast=true and the client will halt with an Exception.

If you expect that the Config server may occasionally be unavailable when your app starts, you can ask it to keep trying after a failure. First, you need to set spring.cloud.config.failFast=true, and then you need to add spring-retry and spring-boot-starter-aop to your classpath. The default behaviour is to retry 6 times with an initial backoff interval of 1000ms and an exponential multiplier of 1.1 for subsequent backoffs. You can configure these properties (and others) using spring.cloud.config.retry.* configuration properties.

Conclusion. Spring Cloud Config allows managing external properties for applications across all environments. Now we are able to create a configuration server to provide a set of configuration files from a Git repository to client applications. As you can see, spring cloud is a very easy and friendly library to use.

REFERENCES

1. Repository of spring cloud config – Github [Electronic resource] / Github Inc., 2017. – Mode of access: https://github.com/spring-cloud/spring-cloud-config. – Date of access: 28.01.2017.
2. Spring framework – Wikipedia [Electronic resource] / Wikimedia Foundation., 2017. – Mode of access: https://en.wikipedia.org/wiki/Spring_Framework. – Date of access: 28.01.2017.
3. Microservice architecture with Spring cloud config – Habrahabr [Electronic resource]. – Mode of access: https://habrahabr.ru/post/280786/. – Date of access: 28.01.2017.