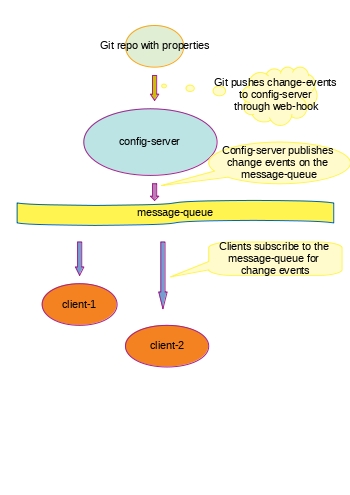
Spring Cloud Config Server and Client

# What are we trying to solve?

With the advent of Microservices based architecture, a lot of stress is given on externalisation of properties. Gone are the days of old when the property files were bundled within the jar or war files. With the advent of 12 Factor App paradigm, we try to store our config files external to our application. How can we do this in a Spring Boot application? Well, the spring-cloud provides support to both config-server (a server storing all our configs, for all environment) and the config-client (client applications that use the configs from the server). After a lot of reading and gleaning a lot of resources, I present to you the perfect, complete guide to implementing a production-ready cloud-config-server and its client.

# Architectural Overview



Well, there are a couple of moving parts to this. Let me list them below:

1. The Git Repo: The git repo is used as the actual storage for the config files or property files. You can store sentitive properties like passwords, in encrypted form. By the way, in this particular example, this is a Git repo, but this can also be any other store. Spring supports Vault, File System, and many other forms of stores.

2. The config-server-proxy: The config server acts as a proxy for the Git Repo storing the actual properties. The clients will always point to the config-server for fetching the properties. The clients are completely insulated from the backed store, the Git Repo in our case. The config server clones the Git Repo locally, decrypts the encrypted properties and then serves the properties in plain-text to the clients

3. In case of any changes to the properties, the Git Repo notifies the config-server through WebHooks. The config-server then publishes the change event onto the default topic called **springCloudBus**.

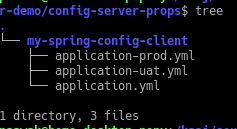
4. The clients are also subscribed to the topic **springCloudBus**. So, they get notified as soon as any config property changes. Then, the changed properties are re-loaded without having to re-start.

# Implementation Details

## Git Repository

## Structure

The Git repo stores the .yml or properties files inside folders for specific projects. The git repo in this case is called **config-server-props**. I have a single project identified by **my-spring-config-client**. So, all properties for different environments will go inside this.



## Configuring web-hook

We can configure web-hooks in any of the Git Repos (GitHub, BitBucket, GitLab etc.) to post an event to the config server (http://localhost:8888/config-server-proxy/monitor) when any push happens. This is how it is done on the GitHub:

## Message Queue

Even before we get started, we would need to have a message-queue. This is needed to decouple of the client and the server. The server publishes change events to the bus, and the clients subscribe to the bus and get notified. In this example we are using a dockerised RabbitMQ. This is how we start it:

docker run -it -d -p 5672:5672 -p 15672:15672 -e RABBITMQ\_DEFAULT\_USER=my\_mq\_user -e RABBITMQ\_DEFAULT\_PASS=mq\_secret\_57\*key rabbitmq:3.7.2-management-alpine

## Server Configuration

This is a Spring Boot project. The name of the project is **config-server-proxy**.

### Dependencies

We would need the below dependencies in the pom.xml:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-monitor</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

### Creating a self-signed certificate

Since we would be doing encryption and decryption of passwords, we would need to create a key-pair. This is the command:

keytool -genkey -noprompt -trustcacerts -keyalg RSA -alias config-server.palashray.com -dname "CN=Palash Ray, OU=Demo, O=Swayam, L=Bangalore, ST=Karnataka, C=IN" -keypass **Red123** -storepass **Yellow98** -keystore certs/config-server-keystore.jks

### Configuration files

We would have to have 2 property files:

bootstrap.yml

spring:

application:

name: config-server-proxy # This is the name/identifier of the project

encrypt: # Details used for encryption and decryption

keyStore:

location: [file:certs/config-server-keystore.jks](file:///certs/config-server-keystore.jks)

alias: config-server.palashray.com

password: **Yellow98** # Storepass

secret: **Red123** # KeyPass

application.yml

server:

contextPath: /config-server-proxy

port: 8888

security:

user:

name: my\_user # custom user name

password: MySecret&23 # custom password: if not set, spring provides a random password on start-up

spring:

rabbitmq:

host: localhost

port: 5672

username: my\_mq\_user

password: mq\_secret\_57\*key

cloud:

config:

server:

git:

uri: https://github.com/paawak/spring-boot-demo.git # path to the Git repo

addLabelLocations: **false**

searchPaths: /spring-config-server-demo/config-server-props/{application} # Path relative to the root of the Git repo, including the name of the client project

basedir: target/props-git-checkout # location where the Git Repo is cloned by the proxy server

cloneOnStart: **true**

management:

security:

enabled: **false**

Note that the searchPaths is appended with /{application}. It is a placeholder for the client-project-name.

### Main Class

You need to decorate the main class with the annotation @EnableConfigServer:

@SpringBootApplication

@EnableConfigServer

**public** **class** ConfigServerProxyApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ConfigServerProxyApplication.**class**, args);

}

}

### Fetching config properties

Fetching config properties from the proxy server is a GET request:

http://localhost:8888/config-server-proxy/clientName-profileName.format

For example, the below url will fetch my default profile in yml format:

curl -u "my\_user:MySecret&23" -s -X GET "http://localhost:8888/config-server-proxy/my-spring-config-client-default.yml"

The below url will fetch my prod profile in properties format:

curl -u "my\_user:MySecret&23" -s -X GET "http://localhost:8888/config-server-proxy/my-spring-config-client-prod.properties"

### Trigerring re-load of properties by the clients

Though it is always recommened to configure web-hook on the Git Repo and not notify the clients directly, we can as well do that. The below command tells the clients to reload all their properties without re-start:

curl -s -X POST --header 'Content-Type: application/x-www-form-urlencoded' --header 'Accept: application/json' -d 'path=\*' "http://localhost:8888/config-server-proxy/monitor"

### Encrypting a password

We can encrypt a password by doing a simple POST:

curl -u "my\_user:MySecret&23" -s -X POST "http://localhost:8888/config-server-proxy/encrypt" --data-urlencode my\_passwd

We can then use this encrypted property inside of our .yml files in the Git Repo.

### Decrypting a password

We can decrypt a password by doing a simple POST:

curl -u "my\_user:MySecret&23" -s -X POST "http://localhost:8888/config-server-proxy/decrypt" --data-urlencode my\_encoded\_passwd

Note that when the proxy-server serves the properties to the client, it decrypts the encrypted passwords and serve them as plain-texts.

## Client Configuration

This is again a Spring Boot project. The name of the project is **my-spring-config-client**.

### Dependencies

We would need the below dependencies in the pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.retry</groupId>

<artifactId>spring-retry</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

### Configuration files

Just a single **bootstrap.yml** would be enough.

spring:

application:

name: my-spring-config-client # This is the client name that would be replaced in the {application} placeholder while resolving the searchPaths for Git Repo

cloud:

config:

uri: http://localhost:8888/config-server-proxy/

username: my\_user

password: MySecret&23

fail-fast: **true**

retry:

initialInterval: 2000

maxAttempts: 3

maxInterval: 3000

multiplier: 1.5

Note that the property spring.application.name is particularly important. This is the property which the config-server uses to identify the searchPaths, and is represented by the placeholder {application}. This is also the name of the folder in the Git Repo where all the project-specific property files reside.

It is always a good practice to have a retry mechanism and fail-fast if the config-server is not available. All the properties under retry are self-explanatory.

### Testing the client

The below curl command will fetch 3 properties that are currently active:

curl  -X GET "http://localhost:8080/my-spring-config-client/"

After you configure the web-hook, change some property in the Git Repo. Wait for some time and give the above command. You would be able to see the changed properties.

## Sources

The sources for all the 3 components are available here: <https://github.com/paawak/spring-boot-demo/tree/master/spring-config-server-demo>

# References

<http://cloud.spring.io/spring-cloud-static/spring-cloud-config/1.4.2.RELEASE/single/spring-cloud-config.html>