# **Quick Intro to Spring**
# Cloud Configuration

Last modified: July 19, 2017

by baeldung (http://www.baeldung.com/author/baeldung/)



## 1. Overview

***Spring Cloud Config*** is Spring's client/server approach for storing and serving distributed configurations across multiple applications and environments.

This configuration store is ideally versioned under *Git* version control and can be modified at application runtime. While it fits very well in Spring applications using all the supported configuration file formats together with constructs like *Environment*, *PropertySource or @Value (/2012/02/06/properties-with-spring/)*, it can be used in any environment running any programming language.

In this write-up, we'll focus on an example of how to setup a *Git*-backed config server, use it in a simple *REST* application server and setup a secure environment including encrypted property values.

## 2. Project Setup and Dependencies

To get ready for writing some code, we create two new *Maven* projects first. The server project is relying on the *spring-cloud-config-server (http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-*

*cloud-config-server%22)* module, as well as the *spring-boot-starter-security (http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-security%22)* and *spring-boot-starter-web (http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-web%22)* starter bundles:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
    <version>1.1.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>
```

However for the client project we're going to only need the *spring-cloud-starter-config (http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-config%22)* and the *spring-boot-starter-web modules (http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-web%22)*:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>1.1.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>
```

# 3. A Config Server Implementation

The main part of the application is a config class – more specifically a *@SpringBootApplication* (/spring-boot-application-configuration) – which pulls in all the required setup through the *auto-configure* annotation *@EnableConfigServer:*

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {

    public static void main(String[] arguments) {
        SpringApplication.run(ConfigServer.class, arguments);
    }
}
```

Now we need to configure the server *port* on which our server is listening and a *Git*-url which provides our version-controlled configuration content. The latter can be used with protocols like *http*, *ssh* or a simple *file* on a local filesystem.

> **Tip:** If you are planning to use multiple config server instances pointing to the same config repository, you can configure the server to clone your repo into a local temporary folder. But be aware of private repositories with two-factor authentication, they are difficult to handle! In such a case, it is easier to clone them on your local filesystem and work with the copy.

There are also some *placeholder variables and search patterns* for configuring the *repository-url* available; but this is beyond the scope of our article. If you are interested, the official documentation is a good place to start.

We also need to set a username and a password for the *Basic-Authentication* in our *application.properties* to avoid an auto-generated password on every application restart:

```
server.port=8888
spring.cloud.config.server.git.uri=ssh://localhost/config-repo (ssh://localhost/config-repo)
spring.cloud.config.server.git.clone-on-start=true
security.user.name=root
security.user.password=s3cr3t
```

# 4. A Git Repository as Configuration Storage

To complete our server, we have to initialize a *Git* repository under the configured url, create some new properties files and popularize them with some values.

The name of the configuration file is composed like a normal Spring *application.properties*, but instead of the word 'application' a configured name, e.g. the value of the property *'spring.application.name'* of the client is used, followed by a dash and the active profile. For example:

```
$> git init
$> echo 'user.role=Developer' > config-client-development.properties
$> echo 'user.role=User'      > config-client-production.properties
$> git add .
$> git commit -m 'Initial config-client properties'
```

**Troubleshooting:** If you run into *ssh*-related authentication issues, double check *~/.ssh/known_hosts* and *~/.ssh/authorized_keys* on your ssh server!

# 5. Querying the Configuration

Now we're able to start our server. The *Git*-backed configuration API provided by our server can be queried using the following paths:

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

In which the *{label}* placeholder refers to a Git branch, *{application}* to the client's application name and the *{profile}* to the client's current active application profile.

So we can retrieve the configuration for our planned config client running under development profile in branch *master* via:

```
$> curl http://root:s3cr3t@localhost:8888/config-client/development/master
```

# 6. The Client Implementation

Next, let's take care of the client. This will be a very simple client application, consisting of a *REST* controller with one *GET* method.

The configuration, to fetch our server, must be placed in a resource file named *bootstrap.application*, because this file (like the name implies) will be loaded very early while the application starts:

```
@SpringBootApplication
```

```
@RestController
public class ConfigClient {

    @Value("${user.role}")
    private String role;

    public static void main(String[] args) {
        SpringApplication.run(ConfigClient.class, args);
    }

    @RequestMapping(
      value = "/whoami/{username}",
      method = RequestMethod.GET,
      produces = MediaType.TEXT_PLAIN_VALUE)
    public String whoami(@PathVariable("username") String username) {
        return String.format("Hello!
          You're %s and you'll become a(n) %s...\n", username, role);
    }
}
```

In addition to the application name, we also put the active profile and the connection-details in our b*ootstrap.properties*:

```
spring.application.name=config-client
spring.profiles.active=development
spring.cloud.config.uri=http://localhost:8888 (http://localhost:8888)
spring.cloud.config.username=root
spring.cloud.config.password=s3cr3t
```

To test, if the configuration is properly received from our server and the *role value* gets injected in our controller method, we simply curl it after booting the client:

```
$> curl http://localhost:8080/whoami/Mr_Pink
```

If the response is as follows, our *Spring Cloud Config Server* and its client are working fine for now:

```
Hello! You're Mr_Pink and you'll become a(n) Developer...
```

# 7. Encryption and Decryption

**Requirement**: To use cryptographically strong keys together with Spring encryption and decryption features you need the *'Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files'* installed in your *JVM.* These can be downloaded for example from Oracle (http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html). To install follow the instructions included in the download. Some Linux distributions also provide an installable package through their package managers.

Since the config server is supporting encryption and decryption of property values, you can use public repositories as storage for sensitive data like usernames and passwords. Encrypted values are prefixed with the string *{cipher}* and can be generated by an REST-call to the path *'/encrypt'*, if the server is configured to use a symmetric key or a key pair.

An endpoint to decrypt is also available. Both endpoints accept a path containing placeholders for the name of the application and its current profile: *'/*/{name}/{profile}'*. This is especially useful for controlling cryptography per client. However, before they become useful, you have to configure a cryptographic key which we will do in the next section.

> **Tip:** If you use curl to call the en-/decryption API, it's better to use the *–data-urlencode* option (instead of *–data/-d*), or set the 'Content-Type' header explicit to *'text/plain'*. This ensures a correct handling of special characters like '+' in the encrypted values.

If a value can't be decrypted automatically while fetching through the client, its *key* is renamed with the name itself, prefixed by the word 'invalid'. This should prevent, for example the usage of an encrypted value as password.

> **Tip:** When setting-up a repository containing YAML files, you have to surround your encrypted and prefixed values with single-quotes! With Properties this is not the case.

## 7.1. Key Management

The config server is per default enabled to encrypt property values in a symmetric or asymmetric way.

**To use symmetric cryptography**, you simply have to set the property *'encrypt.key'* in your *application.properties* to a secret of your choice. Alternatively you can pass-in the environment variable *ENCRYPT_KEY*.

**For asymmetric cryptography**, you can set *'encrypt.key'* to a *PEM*-encoded string value or configure a *keystore* to use.

Because we need a highly secured environment for our demo server, we chose the latter option and generating a new keystore, including a *RSA* key-pair, with the Java *keytool* first:

```
$> keytool -genkeypair -alias config-server-key \
       -keyalg RSA -keysize 4096 -sigalg SHA512withRSA \
       -dname 'CN=Config Server,OU=Spring Cloud,O=Baeldung' \
       -keypass my-k34-s3cr3t -keystore config-server.jks \
       -storepass my-s70r3-s3cr3t
```

After that, we're adding the created keystore to our server's *application.properties* and re-run it:

```
encrypt.key-store.location=classpath:/config-server.jks
encrypt.key-store.password=my-s70r3-s3cr3t
encrypt.key-store.alias=config-server-key
encrypt.key-store.secret=my-k34-s3cr3t
```

As next step we can query the encryption-endpoint and add the response as value to a configuration in our repository:

```
$> export PASSWORD=$(curl -X POST --data-urlencode d3v3L \
       http://root:s3cr3t@localhost:8888/encrypt)
$> echo "user.password=$PASSWORD" >> config-client-development.properties
$> git commit -am 'Added encrypted password'
$> curl -X POST http://root:s3cr3t@localhost:8888/refresh
```

To test, if our setup works correctly, we're modifying the *ConfigClient* class and restart our client:

```
@SpringBootApplication
@RestController
public class ConfigClient {

    ...

    @Value("${user.password}")
    private String password;

    ...
    public String whoami(@PathVariable("username") String username) {
        return String.format("Hello!
          You're %s and you'll become a(n) %s, " +
          "but only if your password is '%s'!\n",
          username, role, password);
    }
}
```

A final query against our client will show us, if our configuration value is being correct decrypted:

```
$> curl http://localhost:8080/whoami/Mr_Pink
Hello! You're Mr_Pink and you'll become a(n) Developer, \
  but only if your password is 'd3v3L'!
```

## 7.2. Using Multiple Keys

If you want to use multiple keys for encryption and decryption, for example: a dedicated one for each served application, you can add another prefix in the form of *{name:value}* between the *{cipher}* prefix and the *BASE64*-encoded property value.

The config server understands prefixes like *{secret:my-crypto-secret}* or *{key:my-key-alias}* nearly out-of-the-box. The latter option needs a configured keystore in your *application.properties*. This keystore is searched for a matching key alias. For example:

```
user.password={cipher}{secret:my-499-s3cr3t}AgAMirj1DkQC0WjRv...
user.password={cipher}{key:config-client-key}AgAMirj1DkQC0WjRv...
```

For scenarios without keystore you have to implement a *@Bean* of type *TextEncryptorLocator* which handles the lookup and returns a *TextEncryptor*-Object for each key.

### 7.3. Serving Encrypted Properties

If you want to disable server-side cryptography and handle decryption of property-values locally, you can put the following in your server's *application.properties*:

```
spring.cloud.config.server.encrypt.enabled=false
```

Furthermore you can delete all the other 'encrypt.*' properties to disable the *REST* endpoints.

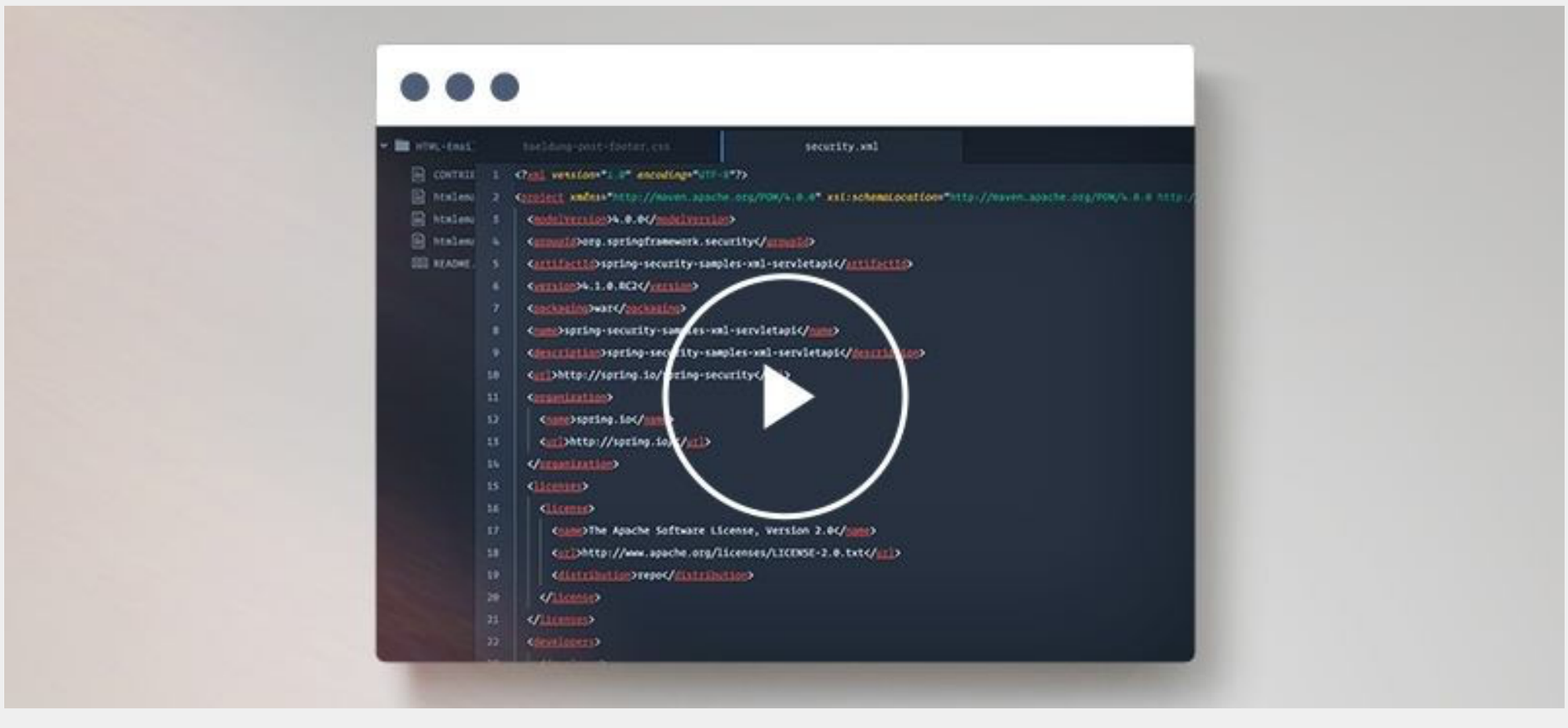
## 8. Conclusion

Now we are able to create a configuration server to provide a set of configuration files from a *Git* repository to client applications. There are a few other things you can do with such a server.

For example:

- Serve configuration in *YAML* or *Properties* format instead of *JSON* – also with placeholders resolved. Which can be useful, when using it in non-Spring environments, where the configuration is not directly mapped to a *PropertySource*.
- Serve plain text configuration files – in turn optionally with resolved placeholders. This can be useful for example to provide an environment-dependent logging-configuration.
- Embed the config server into an application, where it configures itself from a *Git* repository, instead of running as standalone application serving clients. Therefore some bootstrap properties must be set and/or the *@EnableConfigServer* annotation must be removed, which depends on the use case.
- Make the config server available at Spring Netflix Eureka service discovery and enable automatic server discovery in config clients. This becomes important if the server has no fixed location or it moves in its location.

And to wrap up, you'll find the source code to this article *on Github (https://github.com/eugenp/tutorials/tree/master/spring-cloud/spring-cloud-config)*.




**Learn the basics of securing a REST API with Spring**

Get access to the video lesson!

Email Address

Access >>

Sort by: newest|oldest|most voted

Fernando Antonio Barbeiro Camp

Guest

Nice article, congrats!

By the way, I've written about some microservices and I have a post about Spring Cloud as well – the approach is a little bit different from yours, but maybe it can be useful for someone.

https://fernandoabcampos.wordpress.com/2016/03/16/config-server/ (https://fernandoabcampos.wordpress.com/2016/03/16/config-server/)

Regards

2

1 year 2 months ago

Eugen Paraschiv (http://www.baeldung.com/)

Guest

Looks interesting Fernando, it's on my reading list 🙂

Cheers,
Eugen.

0

1 year 2 months ago

Guest

## Mark

Seems using @EnableWebSecurity is not enough. need to use a config class to use basic authentication @Configuration @EnableWebSecurity public class SecurityConfiguration extends WebSecurityConfigurerAdapter { @Value("${security.user.name}") private String authUser; @Value("${security.user.password}") private String authPassword; @Autowired protected void configure(AuthenticationManagerBuilder auth) throws Exception { auth.inMemoryAuthentication().withUser(authUser).password(authPassword).roles("CLIENT"); } @Override protected void configure(HttpSecurity http) throws Exception { http.authorizeRequests().anyRequest().fullyAuthenticated(); http.httpBasic(); http.csrf().disable(); } }

0

1 year 1 month ago

Guest

## Grzegorz Piwowarek

Hmm, we'll give it a closer look and update the article if necessary. Thanks!

0

1 year 1 month ago

Guest

## Grzegorz Piwowarek

Ok, @EnableWebSecurity behaves unexpectedly in the context of Spring Cloud. We managed to get this working only by removing this annotation but yes, if we wanted to keep it, we would need to do something like you suggested 🙂 thanks!

0

1 year 1 month ago

Guest

## Hans Jacob Melby

Hi Eugen and thanks for a great intro to Spring config server 🙂

Do you know if it is possible to have several property files for each application ?

Lets say you have a "large" application where you want to splitt the config into several property files :

Myapp-Datasource.properties

Myapp-Security.properties

Myapp-featureToggles.properties

etc

etc

All examples I can find only show one property file Myapp.properties. This does sond a bit limiting if you ask me..This file can become quite large..

How would you solve this issue ?

Regards

Hans-Jacob Melby

0

11 months 22 days ago

Guest

## Grzegorz Piwowarek

You can definitely split the property file into smaller ones. It's all a matter of telling Spring where to search for the remaining files. Google should give an answer quickly 🙂

0

11 months 13 days ago

Guest

## Hans Jacob Melby

Actually, after comunicating with Josh Long, he confirmed by findings that spring cloud config does not support several property files pr app. (Not that this is a limitation only when you want to use spring cloud ) Well.. you have two.. The "default "one (application.property/yaml) and your spesific (applicationname.property/yaml.. You can also have both yaml and .properties and thus have 3 files. (yaml will have presidence is you have two properties with same name) If you want more files you can support it by using profiles. the "problem" then is that you for example need to have several profiles active... Read more »

0

10 months 15 days ago

Guest

### Grzegorz Piwowarek

Hmm, that's very interesting. The solution with using multiple profiles sounds like a hack so probably should be considered only in "emergency" situations. Thanks for clarifying this up!

0

10 months 15 days ago

Guest

### annettemccull (http://annettemccullough.com)

You can have multiple yaml files by adding spring.cloud.config.name to your config-clients bootstrap.properties file.

e.g. spring.cloud.config.name: Myapp, Myapp-Datasource, Myapp-Security, Myapp-featureToggles

N.B. your .properties file will not be used at all.

0

8 months 26 days ago

Guest

### HP

i always encountered this error.
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'configclientApplication': Injection of autowired dependencies failed; nested exception is org.springframework.beans.factory.BeanCreationException: Could not autowire field: private java.lang.String com.test.ConfigclientApplication.role; nested exception is java.lang.IllegalArgumentException: Could not resolve placeholder 'user.role' in string value "${user.role}"

i can't map my properties

0

11 months 10 days ago

Guest

### Grzegorz Piwowarek

Are you working on our code samples or trying to build your own app?

0

11 months 10 days ago

Guest

### Ana

Is there a way to configure spring-cloud-config client to throw an exception when the config server is down and it cannot fetch the properties, instead of using the local property file?

0

9 months 27 days ago

Guest

### Timothy Schimandle

This is possible by setting this property: `spring.cloud.config.failFast=true` in your bootstrap.properties file. This will stop the application from starting altogether.

0

9 months 25 days ago

C

S
(HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
R
(HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
J
(HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
S
(HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-
2/)
P
(HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
J
(HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
H
(HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

A

A
B
(HTTP://WWW.BAELDUNG.COM/ABOUT/)
T
C
(HTTP://COURSES.BAELDUNG.COM)
N
B
(HTTP://META.BAELDUNG.COM/)
T
F
A
(HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
W
F
B
(HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-
GUIDELINES)
F
P
(HTTP://WWW.BAELDUNG.COM/PRIVACY-
POLICY)
T
O
S
(HTTP://WWW.BAELDUNG.COM/TERMS-
OF-
SERVICE)
C
(HTTP://WWW.BAELDUNG.COM/CONTACT)
C
I
(HTTP://WWW.BAELDUNG.COM/BAELDUNG-
COMPANY-
INFO)
A
C
T
J
V
(HTTP://WWW.BAELDUNG.COM/JAVA-

WEEKLY-
SPONSORSHIP)
C
V
(HTTP://WWW.BAELDUNG.COM/CONSULTING)