Spring Cloud Config

Table of Contents

2.0.0.BUILD-SNAPSHOT

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the Spring `Environment` and `PropertySource` abstractions, so they fit very well with Spring applications, 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 labelled 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.

1. Quick Start

Start the server:

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

The server is a Spring Boot application so you can run it from your IDE instead if you prefer (the main class is `ConfigServerApplication`). Then try out a client:

```
$ curl localhost:8888/foo/development
{"name":"foo","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-development.properties","source":{"bar":"spam"}},
  {"name":"https://github.com/scratches/config-repo/foo.properties","source":{"foo":"bar"}}
```

```
]}
```

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 mini-application's `Environment` is used to enumerate property sources and publish them via a JSON endpoint.

The HTTP service has resources in the form:

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

where the "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".)

Spring Cloud Config Server pulls configuration for remote clients from a git repository (which must be provided):

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
```

1.1 Client Side Usage

To use these 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`. There is also a parent pom and BOM (`spring-cloud-starter-parent`) for Maven users and a Spring IO version management properties file for Gradle and Spring CLI users. Example Maven configuration:

pom.xml.

```
   <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.3.5.RELEASE</version>
       <relativePath /> <!-- lookup parent from repository -->
   </parent>

<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-dependencies</artifactId>
                        <version>Brixton.RELEASE</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>

<dependencies>
        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
```

```
                <scope>test</scope>
        </dependency>
</dependencies>

<build>
        <plugins>
           <plugin>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-maven-plugin</artifactId>
           </plugin>
        </plugins>
</build>

   <!-- repositories also needed for snapshots and milestones -->
```

Then you can create a standard Spring Boot application, like this simple HTTP server:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

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

}
```

When it runs it will pick up the external configuration from the default local config server on port 8888 if it is running. To modify the startup behaviour you can change the location of the config server using `bootstrap.properties` (like `application.properties` but for the bootstrap phase of an application context), e.g.

```
spring.cloud.config.uri: http://myconfigserver.com
```

The bootstrap properties will show up in the `/env` endpoint as a high-priority property source, e.g.

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

(a property source called "configService:<URL of remote repository>/<file name>" contains the property "foo" with value "bar" and is highest priority).

the URL in the property source name is the git repository not the config server URL.

2. Spring Cloud Config Server

The Server provides an HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content). The server is easily embeddable in a Spring Boot application using the `@EnableConfigServer` annotation. So this app is a config server:

ConfigServer.java.

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServer.class, args);
  }
}
```

Like all Spring Boot apps it runs on port 8080 by default, but you can switch it to the conventional port 8888 in various ways. The easiest, which also sets a default configuration repository, is by launching it with `spring.config.name=configserver` (there is a `configserver.yml` in the Config Server jar). Another is to use your own `application.properties`, e.g.

application.properties.

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

where `${user.home}/config-repo` is a git repository containing YAML and properties files.

in Windows you need an extra "/" in the file URL if it is absolute with a drive prefix, e.g. `file:///${user.home}/config-repo`.

Here's a recipe for creating the git repository in the example above:

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

using the local filesystem for your git repository is intended for testing only. Use a server to host your configuration repositories in production.

the initial clone of your configuration repository will be quick and efficient if you only keep text files in it. If you start to store binary files, especially large ones, you may experience delays on the first request for configuration and/or out of memory errors in the server.

2.1 Environment Repository

Where do you want to store the configuration data for the Config Server? The strategy that governs this behaviour is the `EnvironmentRepository`, serving `Environment` objects. This `Environment` is a shallow copy of the domain from the Spring `Environment` (including `propertySources` as the main feature). The `Environment` resources are parametrized by three variables:

- `{application}` maps to "spring.application.name" on the client side;
- `{profile}` maps to "spring.profiles.active" on the client (comma separated list); and
- `{label}` which is a server side feature labelling a "versioned" set of config files.

Repository implementations generally behave just like a Spring Boot application loading configuration files from a "spring.config.name" equal to the `{application}` parameter, and "spring.profiles.active" equal to the `{profiles}` parameter. Precedence rules for profiles are also the same as in a regular Boot application: active profiles take precedence over defaults, and if there are multiple profiles the last one wins (like adding entries to a `Map`).

Example: a client application has this bootstrap configuration:

bootstrap.yml.

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

(as usual with a Spring Boot application, these properties could also be set as environment variables or command line arguments).

If the repository is file-based, the server will create an `Environment` from `application.yml` (shared between all clients), and `foo.yml` (with `foo.yml` taking precedence). If the YAML files have documents inside them that point to Spring profiles, those are applied with higher precedence (in order of the profiles listed), and if there are profile-specific YAML (or properties) files these are also applied with higher precedence than the defaults. Higher precedence translates to a `PropertySource` listed earlier in the `Environment`. (These are the same rules as apply in a standalone Spring Boot application.)

2.1.1 Git Backend

The default implementation of `EnvironmentRepository` uses a Git backend, which is very convenient for managing upgrades and physical environments, and also for auditing changes. To change the location of the repository you can set the "spring.cloud.config.server.git.uri" configuration property in the Config Server (e.g. in `application.yml`). If you set it with a `file:` prefix it should work from a local repository so you can get started quickly and easily without a server, but in that case the server operates directly on the local repository without cloning it (it doesn't matter if it's not bare because the Config Server never makes changes to the "remote" repository). To scale the Config Server up and make it highly available, you would need to have all instances of the server pointing to the same repository, so only a shared file system would work. Even in that case it is better to use the `ssh:` protocol for a shared filesystem repository, so that the server can clone it and use a local working copy as a cache.

This repository implementation maps the `{label}` parameter of the HTTP resource to a git label (commit id, branch name or tag). If the git branch or tag name contains a slash ("/") then the label in the HTTP URL should be specified with the special string "(_)" instead (to avoid ambiguity with other URL paths). For example, if the label is `foo/bar`, replacing the slash would result in a label that looks like `foo(_)bar`. The inclusion of the special string "(_)" can also be applied to the `{application}` parameter. Be careful with the brackets in the URL if you are using a command line client like curl (e.g. escape them from the shell with quotes ").

Placeholders in Git URI

Spring Cloud Config Server supports a git repository URL with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it, but remember that the label is applied as a git label anyway). So you can easily support a "one repo per application" policy using (for example):

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

or a "one repo per profile" policy using a similar pattern but with `{profile}`.

Additionally, using the special string "(_)" within your `{application}` parameters can enable support for multiple organizations (for example):

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/{application}
```

where `{application}` is provided at request time in the format "organization(_)application".

Pattern Matching and Multiple Repositories

There is also support for more complex requirements with pattern matching on the application and profile name. The pattern format is a comma-separated list of `{application}/{profile}` names with wildcards (where a pattern beginning with a wildcard may need to be quoted). Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

If `{application}/{profile}` does not match any of the patterns, it will use the default uri defined under "spring.cloud.config.server.git.uri". In the above example, for the "simple" repository, the pattern is `simple/*` (i.e. it only matches one application named "simple" in all profiles). The "local" repository matches all application names beginning with "local" in all profiles (the `/*` suffix is added automatically to any pattern that doesn't have a profile matcher).

the "one-liner" short cut used in the "simple" example above can only be used if the only property to be set is the URI. If you need to set anything else (credentials, pattern, etc.) you need to use the full form.

The `pattern` property in the repo is actually an array, so you can use a YAML array (or `[0]`, `[1]`, etc. suffixes in properties files) to bind to multiple patterns. You may need to do this if you are going to run apps with multiple profiles. Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```

Spring Cloud will guess that a pattern containing a profile that doesn't end in `*` implies that you actually want to match a list of profiles starting with this pattern (so `*/staging` is a shortcut for `["*/staging", "*/staging,*"]`). This is common where you need to run apps in the "development" profile locally but also the "cloud" profile remotely, for instance.

Every repository can also optionally store config files in sub-directories, and patterns to search for those directories can be specified as `searchPaths`. For example at the top level:

```
spring:
  cloud:
    config:
```

```
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: foo,bar*
```

In this example the server searches for config files in the top level and in the "foo/" sub-directory and also any sub-directory whose name begins with "bar".

By default the server clones remote repositories when configuration is first requested. The server can be configured to clone the repositories at startup. For example at the top level:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
                pattern: team-a-*
                cloneOnStart: true
                uri: http://git/team-a/config-repo.git
            team-b:
                pattern: team-b-*
                cloneOnStart: false
                uri: http://git/team-b/config-repo.git
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

In this example the server clones team-a's config-repo on startup before it accepts any requests. All other repositories will not be cloned until configuration from the repository is requested.

Setting a repository to be cloned when the Config Server starts up can help to identify a misconfigured configuration source (e.g., an invalid repository URI) quickly, while the Config Server is starting up. With `cloneOnStart` not enabled for a configuration source, the Config Server may start successfully with a misconfigured or invalid configuration source and not detect an error until an application requests configuration from that configuration source.

Authentication

To use HTTP basic authentication on the remote repository add the "username" and "password" properties separately (not in the URL), e.g.

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

If you don't use HTTPS and user credentials, SSH should also work out of the box when you store keys in the default directories (`~/.ssh`) and the uri points to an SSH location, e.g. "git@github.com:configuration/cloud-configuration". It is important that an entry for the Git server be present in the `~/.ssh/known_hosts` file and that it is in `ssh-rsa` format. Other formats (like `ecdsa-sha2-nistp256`) are not supported. To avoid surprises, you should ensure that only one entry is present in the `known_hosts` file for the Git server and that it is matching with the URL you provided to the config server. If you used a hostname in the URL, you want to have exactly that in the `known_hosts` file, not the IP. The repository is accessed using JGit, so any documentation you find on that should be applicable. HTTPS proxy settings can be set in `~/.git/config` or in the same way as for any other JVM process via system properties (`-Dhttps.proxyHost` and `-Dhttps.proxyPort`).

If you don't know where your `~/.git` directory is use `git config --global` to manipulate the settings (e.g. `git config --global http.sslVerify false`).

Authentication with AWS CodeCommit

AWS CodeCommit authentication can also be done. AWS CodeCommit uses an authentication helper when using Git from the command line. This helper is not used with the JGit library, so a JGit CredentialProvider for AWS CodeCommit will be created if the Git URI matches the AWS CodeCommit pattern. AWS CodeCommit URIs always look like https://git-codecommit.${AWS_REGION}.amazonaws.com/${repopath}.

If you provide a username and password with an AWS CodeCommit URI, then these must be the AWS accessKeyId and secretAccessKey to be used to access the repository. If you do not specify a username and password, then the accessKeyId and secretAccessKey will be retrieved using the AWS Default Credential Provider Chain.

If your Git URI matches the CodeCommit URI pattern (above) then you must provide valid AWS credentials in the username and password, or in one of the locations supported by the default credential provider chain. AWS EC2 instances may use IAM Roles for EC2 Instances.

Note: The aws-java-sdk-core jar is an optional dependency. If the aws-java-sdk-core jar is not on your classpath, then the AWS Code Commit credential provider will not be created regardless of the git server URI.

Git SSH configuration using properties

By default, the JGit library used by Spring Cloud Config Server uses SSH configuration files such as `~/.ssh/known_hosts` and `/etc/ssh/ssh_config` when connecting to Git repositories using an SSH URI. In cloud environments such as Cloud Foundry, the local filesystem may be ephemeral or not easily accessible. For cases such as these, SSH configuration can be set using Java properties. In order to activate property based SSH configuration, the property `spring.cloud.config.server.git.ignoreLocalSshSettings` must be set to `true`. Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
                      -----BEGIN RSA PRIVATE KEY-----
                      MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX
                      IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF
                      ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud
                      1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i
                      oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W
                      DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd
                      fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b
                      BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG
                      EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj
                      5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8
                      +AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe
                      pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG
                      ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ
                      xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW
                      dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi

                      PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX
                      VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z
                      FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R
                      gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4
                      VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV
                      cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee
                      KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N
                      CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs
```

```
69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
-----END RSA PRIVATE KEY-----
```

Table 2.1. SSH Configuration properties

Property Name	Remarks
ignoreLocalSshSettings	If true, use property based SSH config instead of file based. Must be set at as `spring.cloud.config.server.git.ignoreLocalSshSettings`, **not** inside a repository definition.
privateKey	Valid SSH private key. Must be set if `ignoreLocalSshSettings` is true and Git URI is SSH format
hostKey	Valid SSH host key. Must be set if `hostKeyAlgorithm` is also set
hostKeyAlgorithm	One of `ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384 ,ecdsa-sha2-nistp521`. Must be set if `hostKey` is also set
strictHostKeyChecking	`true` or `false`. If false, ignore errors with host key
knownHostsFile	Location of custom .known_hosts file
preferredAuthentications	Override server authentication method order. This should allow evade login prompts if server has keyboard-interactive authentication before `publickey` method.

Placeholders in Git Search Paths

Spring Cloud Config Server also supports a search path with placeholders for the `{application}` and `{profile}` (and `{label}` if you need it). Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: '{application}'
```

searches the repository for files in the same name as the directory (as well as the top level). Wildcards are also valid in a search path with placeholders (any matching directory is included in the search).

Force pull in Git Repositories

As mentioned before Spring Cloud Config Server makes a clone of the remote git repository and if somehow the local copy gets dirty (e.g. folder content changes by OS process) so Spring Cloud Config Server cannot update the local copy from remote repository.

To solve this there is a `force-pull` property that will make Spring Cloud Config Server force pull from remote repository if the local copy is dirty. Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          force-pull: true
```

If you have a multiple repositories configuration you can configure the `force-pull` property per repository. Example:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
                pattern: team-a-*
                uri: http://git/team-a/config-repo.git
                force-pull: true
            team-b:
                pattern: team-b-*
                uri: http://git/team-b/config-repo.git
                force-pull: true
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

The default value for `force-pull` property is `false`.

2.1.2 Version Control Backend Filesystem Use

With VCS based backends (git, svn) files are checked out or cloned to the local filesystem. By default they are put in the system temporary directory with a prefix of `config-repo-`. On linux, for example it could be `/tmp/config-repo-<randomid>`. Some operating systems routinely clean out temporary directories. This can lead to unexpected behaviour such as missing properties. To avoid this problem, change the directory Config Server uses, by setting `spring.cloud.config.server.git.basedir` or `spring.cloud.config.server.svn.basedir` to a directory that does not reside in the system temp structure.

2.1.3 File System Backend

There is also a "native" profile in the Config Server that doesn't use Git, but just loads the config files from the local classpath or file system (any static URL you want to point to with "spring.cloud.config.server.native.searchLocations"). To use the native profile just launch the Config Server with "spring.profiles.active=native".

Remember to use the `file:` prefix for file resources (the default without a prefix is usually the classpath). Just as with any Spring Boot configuration you can embed `${}`-style environment placeholders, but remember that absolute paths in Windows require an extra "/", e.g. `file:///${user.home}/config-repo`

The default value of the `searchLocations` is identical to a local Spring Boot application (so `[classpath:/, classpath:/config, file:./, file:./config]`). This does not expose the `application.properties` from the server to all clients because any property sources present in the server are removed before being sent to the client.

A filesystem backend is great for getting started quickly and for testing. To use it in production you need to be sure that the file system is reliable, and shared across all instances of the Config Server.

The search locations can contain placeholders for `{application}`, `{profile}` and `{label}`. In this way you can segregate the directories in the path, and choose a strategy that makes sense for you (e.g. sub-directory per application, or sub-directory per profile).

If you don't use placeholders in the search locations, this repository also appends the `{label}` parameter of the HTTP resource to a suffix on the search path, so properties files are loaded from each search location **and** a subdirectory with the same name as the label (the labelled properties take precedence in the Spring Environment). Thus the default behaviour with no placeholders is the same as adding a search location ending with `/{label}/`. For example `file:/tmp/config` is the same as `file:/tmp/config,file:/tmp/config/{label}`. This behavior can be disabled by setting `spring.cloud.config.server.native.addLabelLocations=false`.

2.1.4 Vault Backend

Spring Cloud Config Server also supports Vault as a backend.

> Vault is a tool for securely accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, and more. Vault provides a unified interface to any secret, while providing tight access control and recording a detailed audit log.

For more information on Vault see the Vault quickstart guide.

To enable the config server to use a Vault backend you can run your config server with the `vault` profile. For example in your config server's `application.properties` you can add `spring.profiles.active=vault`.

By default the config server will assume your Vault server is running at `http://127.0.0.1:8200`. It also will assume that the name of backend is `secret` and the key is `application`. All of these defaults can be configured in your config server's `application.properties`. Below is a table of configurable Vault properties. All properties are prefixed with `spring.cloud.config.server.vault`.

Name	Default Value
host	127.0.0.1
port	8200
scheme	http
backend	secret
defaultKey	application
profileSeparator	,

All configurable properties can be found in `org.springframework.cloud.config.server.environment.VaultEnvironmentRepository`.

With your config server running you can make HTTP requests to the server to retrieve values from the Vault backend. To do this you will need a token for your Vault server.

First place some data in you Vault. For example

```
$ vault write secret/application foo=bar baz=bam
$ vault write secret/myapp foo=myappsbar
```

Now make the HTTP request to your config server to retrieve the values.

```
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
```

You should see a response similar to this after making the above request.

```
{
   "name":"myapp",
   "profiles":[
      "default"
   ],
```

```
   "label":null,
   "version":null,
   "state":null,
   "propertySources":[
      {
         "name":"vault:myapp",
         "source":{
            "foo":"myappsbar"
         }
      },
      {
         "name":"vault:application",
         "source":{
            "baz":"bam",
            "foo":"bar"
         }
      }
   ]
}
```


Multiple Properties Sources

When using Vault you can provide your applications with multiple properties sources. For example, assume you have written data to the following paths in Vault.

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

Properties written to `secret/application` are available to all applications using the Config Server. An application with the name `myApp` would have any properties written to `secret/myApp` and `secret/application` available to it. When `myApp` has the `dev` profile enabled then properties written to all of the above paths would be available to it, with properties in the first path in the list taking priority over the others.

2.1.5 Sharing Configuration With All Applications

File Based Repositories

With file-based (i.e. git, svn and native) repositories, resources with file names in `application*` are shared between all client applications (so `application.properties`, `application.yml`, `application-*.properties` etc.). You can use resources with these file names to configure global defaults and have them overridden by application-specific files as necessary.

The #_property_overrides[property overrides] feature can also be used for setting global defaults, and with placeholders applications are allowed to override them locally.

With the "native" profile (local file system backend) it is recommended that you use an explicit search location that isn't part of the server's own configuration. Otherwise the `application*` resources in the default search locations are removed because they are part of the server.

Vault Server

When using Vault as a backend you can share configuration with all applications by placing configuration in `secret/application`. For example, if you run this Vault command

```
$ vault write secret/application foo=bar baz=bam
```

All applications using the config server will have the properties `foo` and `baz` available to them.

2.1.6 JDBC Backend

Spring Cloud Config Server supports JDBC (relation database) as a backend for configuration properties. You can enable this feature by adding `spring-jdbc` to the classpath, and using the "jdbc" profile, or by adding a bean of type `JdbcEnvironmentRepository`. Spring Boot will configure a data source if you include the right dependencies on the classpath (see the user guide for more details on that).

The database needs to have a table called "PROPERTIES" with columns "APPLICATION", "PROFILE", "LABEL" (with the usual `Environment` meaning), plus "KEY" and "VALUE" for the key and value pairs in `Properties` style. All fields are of type String in Java, so you can make them `VARCHAR` of whatever length you need. Property values behave in the same way as they would if they came from Spring Boot properties files named `{application}-{profile}.properties`, including all the encryption and decryption, which will be applied as post-processing steps (i.e. not in the repository implementation directly).

2.1.7 Composite Environment Repositories

In some scenarios you may wish to pull configuration data from multiple environment repositories. To do this you can just enable multiple profiles in your config server's application properties or YAML file. If, for example, you want to pull configuration data from a Git repository as well as a SVN repository you would set the following properties for your configuration server.

```
spring:
  profiles:
    active: git, svn
  cloud:
    config:
      server:
        svn:
          uri: file:///path/to/svn/repo
          order: 2
        git:
          uri: file:///path/to/git/repo
          order: 1
```

In addition to each repo specifying a URI, you can also specify an `order` property. The `order` property allows you to specify the priority order for all your repositories. The lower the numerical value of the `order` property the higher priority it will have. The priority order of a repository will help resolve any potential conflicts between repositories that contain values for the same properties.

Any type of failure when retrieving values from an environment repositoy will result in a failure for the entire composite environment.

When using a composite environment it is important that all repos contain the same label(s). If you have an environment similar to the one above and you request configuration data with the label `master` but the SVN repo does not contain a branch called `master` the entire request will fail.

Custom Composite Environment Repositories

It is also possible to provide your own `EnvironmentRepository` bean to be included as part of a composite environment in addition to using one of the environment repositories from Spring Cloud. To do this your bean must implement the `EnvironmentRepository` interface. If you would like to control the priority of you custom `EnvironmentRepository` within the composite environment you should also implement the `Ordered` interface and override the `getOrdered` method. If you do not implement the `Ordered` interface then your `EnvironmentRepository` will be given the lowest priority.

2.1.8 Property Overrides

The Config Server has an "overrides" feature that allows the operator to provide configuration properties to all applications that cannot be accidentally changed by the application using the normal Spring Boot hooks. To declare overrides just add a map of name-value pairs to `spring.cloud.config.server.overrides`. For example

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

will cause all applications that are config clients to read `foo=bar` independent of their own configuration. (Of course an application can use the data in the Config Server in any way it likes, so overrides are not enforceable, but they do provide useful default behaviour if they are Spring Cloud Config clients.)

Normal, Spring environment placeholders with "${}" can be escaped (and resolved on the client) by using backslash ("\") to escape the "$" or the "{", e.g. `\${app.foo:bar}` resolves to "bar" unless the app provides its own "app.foo". Note that in YAML you don't need to escape the backslash itself, but in properties files you do, when you configure the overrides on the server.

You can change the priority of all overrides in the client to be more like default values, allowing applications to supply their own values in environment variables or System properties, by setting the flag `spring.cloud.config.overrideNone=true` (default is false) in the remote repository.

2.2 Health Indicator

Config Server comes with a Health Indicator that checks if the configured `EnvironmentRepository` is working. By default it asks the `EnvironmentRepository` for an application named `app`, the `default` profile and the default label provided by the `EnvironmentRepository` implementation.

You can configure the Health Indicator to check more applications along with custom profiles and custom labels, e.g.

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

You can disable the Health Indicator by setting `spring.cloud.config.server.health.enabled=false`.

2.3 Security

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 pretty much anything.

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 going to be 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).

2.4 Encryption and Decryption

Important

Prerequisites: to use the encryption and decryption features you need the full-strength JCE installed in your JVM (it's not there by default). You can download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction

Policy Files" from Oracle, and follow instructions for installation (essentially replace the 2 policy files in the JRE lib/security directory with the ones that you downloaded).

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 don't 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 accidentally leaking.

If you are setting up a remote config repository for config client applications it might contain an `application.yml` like this, for instance:

application.yml.

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

Encrypted values in a .properties file must not be wrapped in quotes, otherwise the value will not be decrypted:

application.properties.

```
spring.datasource.username: dbuser
spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

You can safely push this plain text to a shared git repository and the secret password is protected.

The server also exposes `/encrypt` and `/decrypt` endpoints (on the assumption that these will be secured and only accessed by authorized agents). If you are editing a remote config file you can use the Config Server to encrypt values by POSTing to the `/encrypt` endpoint, e.g.

```
$ curl localhost:8888/encrypt -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```

If the value you are encrypting has characters in it that need to be URL encoded you should use the `--data-urlencode` option to `curl` to make sure they are encoded properly.

Be sure not to include any of the curl command statistics in the encrypted value. Outputting the value to a file can help avoid this problem.

The inverse operation is also available via `/decrypt` (provided the server is configured with a symmetric key or a full key pair):

```
$ curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

If you are testing like this with curl, then use `--data-urlencode` (instead of `-d`) or set an explicit `Content-Type: text/plain` to make sure curl encodes the data correctly when there are special characters ('+' is particularly tricky).

Take the encrypted value and add the `{cipher}` prefix before you put it in the YAML or properties file, and before you commit and push it to a remote, potentially insecure store.

The `/encrypt` and `/decrypt` endpoints also both accept paths of the form `/*/{name}/{profiles}` which can be used to control cryptography per application (name) and profile when clients call into the main Environment resource.

to control the cryptography in this granular way you must also provide a `@Bean` of type `TextEncryptorLocator`

> that creates a different encryptor per name and profiles. The one that is provided by default does not do this (so all encryptions use the same key).

The `spring` command line client (with Spring Cloud CLI extensions installed) can also be used to encrypt and decrypt, e.g.

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

To use a key in a file (e.g. an RSA public key for encryption) prepend the key value with "@" and provide the file path, e.g.

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

The key argument is mandatory (despite having a `--` prefix).

2.5 Key Management

The Config Server can use a symmetric (shared) key or an asymmetric one (RSA key pair). The asymmetric choice is superior in terms of security, but it is often more convenient to use a symmetric key since it is just a single property value to configure in the `bootstrap.properties`.

To configure a symmetric key you just need to set `encrypt.key` to a secret String (or use an enviroment variable `ENCRYPT_KEY` to keep it out of plain text configuration files).

To configure an asymmetric key you can either set the key as a PEM-encoded text value (in `encrypt.key`), or via a keystore (e.g. as created by the `keytool` utility that comes with the JDK). The keystore properties are `encrypt.keyStore.*` with `*` equal to

- `location` (a `Resource` location),
- `password` (to unlock the keystore) and
- `alias` (to identify which key in the store is to be used).

The encryption is done with the public key, and a private key is needed for decryption. Thus in principle you can configure only the public key in the server if you only want to do encryption (and are prepared to decrypt the values yourself locally with the private key). In practice you might not want to do that because it spreads the key management process around all the clients, instead of concentrating it in the server. On the other hand it's a useful option if your config server really is relatively insecure and only a handful of clients need the encrypted properties.

2.6 Creating a Key Store for Testing

To create a keystore for testing you can do something like this:

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
  -keypass changeme -keystore server.jks -storepass letmein
```

Put the `server.jks` file in the classpath (for instance) and then in your `bootstrap.yml` for the Config Server:

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

2.7 Using Multiple Keys and Key Rotation

In addition to the `{cipher}` prefix in encrypted property values, the Config Server looks for `{name:value}` prefixes (zero or many) before the start of the (Base64 encoded) cipher text. The keys are passed to a `TextEncryptorLocator` which can do whatever logic it needs to locate a `TextEncryptor` for the cipher. If you have configured a keystore (`encrypt.keystore.location`) the default locator will look for keys in the store with aliases as supplied by the "key" prefix, i.e. with a cipher text like this:

```
foo:
  bar: `{cipher}{key:testkey}...`
```

the locator will look for a key named "testkey". A secret can also be supplied via a `{secret:…}` value in the prefix, but if it is not the default is to use the keystore password (which is what you get when you build a keytore and don't specify a secret). If you **do** supply a secret it is recommended that you also encrypt the secrets using a custom `SecretLocator`.

Key rotation is hardly ever necessary on cryptographic grounds if the keys are only being used to encrypt a few bytes of configuration data (i.e. they are not being used elsewhere), but occasionally you might need to change the keys if there is a security breach for instance. In that case all the clients would need to change their source config files (e.g. in git) and use a new `{key:…}` prefix in all the ciphers, checking beforehand of course that the key alias is available in the Config Server keystore.

the `{name:value}` prefixes can also be added to plaintext posted to the `/encrypt` endpoint, if you want to let the Config Server handle all encryption as well as decryption.

2.8 Serving Encrypted Properties

Sometimes you want the clients to decrypt the configuration locally, instead of doing it in the server. In that case you can still have /encrypt and /decrypt endpoints (if you provide the `encrypt.*` configuration to locate a key), but you need to explicitly switch off the decryption of outgoing properties using `spring.cloud.config.server.encrypt.enabled=false`. If you don't care about the endpoints, then it should work if you configure neither the key nor the enabled flag.

3. Serving Alternative Formats

The default JSON format from the environment endpoints is perfect for consumption by Spring applications because it maps directly onto the `Environment` abstraction. If you prefer you can consume the same data as YAML or Java properties by adding a suffix to the resource path (".yml", ".yaml" or ".properties"). This can be useful for consumption by applications that do not care about the structure of the JSON endpoints, or the extra metadata they provide, for example an application that is not using Spring might benefit from the simplicity of this approach.

The YAML and properties representations have an additional flag (provided as a boolean query parameter `resolvePlaceholders`) to signal that placeholders in the source documents, in the standard Spring `${…}` form, should be resolved in the output where possible before rendering. This is a useful feature for consumers that don't know about the Spring placeholder conventions.

there are limitations in using the YAML or properties formats, mainly in relation to the loss of metadata. The JSON is structured as an ordered list of property sources, for example, with names that correlate with the source. The YAML and properties forms are coalesced into a single map, even if the origin of the values has multiple sources, and the names of the original source files are lost. The YAML representation is not necessarily a faithful representation of the YAML source in a backing repository either: it is constructed from a list of flat property sources, and assumptions have to be made about the form of the keys.

4. Serving Plain Text

Instead of using the `Environment` abstraction (or one of the alternative representations of it in YAML or properties format) your applications might need generic plain text configuration files, tailored to their environment. The Config Server provides these through an additional endpoint at `/{name}/{profile}/{label}/{path}` where "name", "profile" and "label" have the same meaning as the regular environment endpoint, but "path" is a file name (e.g. `log.xml`). The source files for this endpoint are located in the same way as for the environment endpoints: the same search path is used as for properties or YAML files, but instead of aggregating all matching resources, only the first one to match is returned.

After a resource is located, placeholders in the normal format (`${…}`) are resolved using the effective `Environment` for the application name, profile and label supplied. In this way the resource endpoint is tightly integrated with the environment endpoints. Example, if you have this layout for a GIT (or SVN) repository:

```
application.yml
nginx.conf
```

where `nginx.conf` looks like this:

```
server {
    listen              80;
    server_name         ${nginx.server.name};
}
```

and `application.yml` like this:

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

then the `/foo/default/master/nginx.conf` resource looks like this:

```
server {
    listen              80;
    server_name         example.com;
}
```

and `/foo/development/master/nginx.conf` like this:

```
server {
    listen              80;
    server_name         develop.com;
}
```

Just like the source files for environment configuration, the "profile" is used to resolve the file name, so if you want a profile-specific file then `/*/development/*/logback.xml` will be resolved by a file called `logback-development.xml` (in preference to `logback.xml`).

If you do not want to supply the `label` and let the server use the default label, you can supply a `useDefaultLabel` request parameter. So, the above example for the `default` profile could look like `/foo/default/nginx.conf?useDefaultLabel`.

5. Embedding the Config Server

The Config Server runs best as a standalone application, but if you need to you can embed it in another application. Just use the `@EnableConfigServer` annotation. An optional property that can be useful in this case is `spring.cloud.config.server.bootstrap` which is a flag to indicate that the server should configure itself from its own remote repository. The flag is off by default because it can delay startup, but when embedded in another application it makes sense to initialize the same way as any other application.

It should be obvious, but remember that if you use the bootstrap flag the config server will need to have its name and repository URI configured in `bootstrap.yml`.

To change the location of the server endpoints you can (optionally) set `spring.cloud.config.server.prefix`, e.g. "/config", to serve the resources under a prefix. The prefix should start but not end with a "/". It is applied to the `@RequestMappings` in the Config Server (i.e. underneath the Spring Boot prefixes `server.servletPath` and `server.contextPath`).

If you want to read the configuration for an application directly from the backend repository (instead of from the config server) that's basically an embedded config server with no endpoints. You can switch off the endpoints entirely if you don't use the `@EnableConfigServer` annotation (just set `spring.cloud.config.server.bootstrap=true`).

6. Push Notifications and Spring Cloud Bus

Many source code repository providers (like Github, Gitlab or Bitbucket for instance) will notify you of changes in a repository through a webhook. You can configure the webhook via the provider's user interface as a URL and a set of events in which you are interested. For instance Github will POST to the webhook with a JSON body containing a list of commits, and a header "X-Github-Event" equal to "push". If you add a dependency on the `spring-cloud-config-monitor` library and activate the Spring Cloud Bus in your Config Server, then a "/monitor" endpoint is enabled.

When the webhook is activated the Config Server will send a `RefreshRemoteApplicationEvent` targeted at the applications it thinks might have changed. The change detection can be strategized, but by default it just looks for changes in files that match the application name (e.g. "foo.properties" is targeted at the "foo" application, and "application.properties" is targeted at all applications). The strategy if you want to override the behaviour is `PropertyPathNotificationExtractor` which accepts the request headers and body as parameters and returns a list of file paths that changed.

The default configuration works out of the box with Github, Gitlab or Bitbucket. In addition to the JSON notifications from Github, Gitlab or Bitbucket you can trigger a change notification by POSTing to "/monitor" with a form-encoded body parameters `path={name}`. This will broadcast to applications matching the "{name}" pattern (can contain wildcards).

the `RefreshRemoteApplicationEvent` will only be transmitted if the `spring-cloud-bus` is activated in the Config Server and in the client application.

the default configuration also detects filesystem changes in local git repositories (the webhook is not used in that case but as soon as you edit a config file a refresh will be broadcast).

7. Spring Cloud Config Client

A Spring Boot application can take immediate advantage of the Spring Config Server (or other external property sources provided by the application developer), and it will also pick up some additional useful features related to `Environment` change events.

7.1 Config First Bootstrap

This is the default behaviour for any application which has the Spring Cloud Config Client on the classpath. When a config client starts up it binds to the Config Server (via the bootstrap configuration property `spring.cloud.config.uri`) and initializes Spring `Environment` with remote property sources.

The net result of this is that all client apps that want to consume the Config Server need a `bootstrap.yml` (or an environment variable) with the server address in `spring.cloud.config.uri` (defaults to "http://localhost:8888").

7.2 Discovery First Bootstrap

If you are using a `DiscoveryClient implementation, such as Spring Cloud Netflix and Eureka Service Discovery or Spring Cloud Consul (Spring Cloud Zookeeper does not support this yet), then you can have the Config Server register with the Discovery Service if you want to, but in the default "Config First" mode, clients won't be able to take advantage of the registration.

If you prefer to use `DiscoveryClient` to locate the Config Server, you can do that by setting `spring.cloud.config.discovery.enabled=true` (default "false"). The net result of that is that client apps all need a `bootstrap.yml` (or an environment variable) with the appropriate discovery configuration. For example, with Spring Cloud Netflix, you need to define the Eureka server address, e.g. in `eureka.client.serviceUrl.defaultZone`. The price for using this option is an extra network round trip on start up to locate the service registration. The benefit is that the Config Server can change its co-ordinates, as long as the Discovery Service is a fixed point. The default service id is "configserver" but you can change that on the client with `spring.cloud.config.discovery.serviceId` (and on the server in the usual way for a service, e.g. by setting `spring.application.name`).

The discovery client implementations all support some kind of metadata map (e.g. for Eureka we have `eureka.instance.metadataMap`). Some additional properties of the Config Server may need to be configured in its service registration metadata so that clients can connect correctly. If the Config Server is secured with HTTP Basic you can configure the credentials as "username" and "password". And if the Config Server has a context path you can set "configPath". Example, for a Config Server that is a Eureka client:

bootstrap.yml.

```
eureka:
  instance:
    ...
    metadataMap:
      user: osufhalskjrtl
      password: lviuhlszvaorhvlo5847
      configPath: /config
```

7.3 Config Client Fail Fast

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 behavior, set the bootstrap configuration property `spring.cloud.config.failFast=true` and the client will halt with an Exception.

7.4 Config Client Retry

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.

To take full control of the retry add a `@Bean` of type `RetryOperationsInterceptor` with id "configServerRetryInterceptor". Spring Retry has a `RetryInterceptorBuilder` that makes it easy to create one.

7.5 Locating Remote Configuration Resources

The Config Service serves property sources from `/{name}/{profile}/{label}`, where the default bindings in the client app are

- "name" = `${spring.application.name}`
- "profile" = `${spring.profiles.active}` (actually `Environment.getActiveProfiles()`)
- "label" = "master"

All of them can be overridden by setting `spring.cloud.config.*` (where `*` is "name", "profile" or "label"). The "label" is useful for rolling back to previous versions of configuration; with the default Config Server implementation it can be a git label, branch name or commit id. Label can also be provided as a comma-separated list, in which case the items in the list are tried on-by-one until one succeeds. This can be useful when working on a feature branch, for instance, when you might want to align the config label with your branch, but make it optional (e.g. `spring.cloud.config.label=myfeature,develop`).

7.6 Security

If you use HTTP Basic security on the server then clients just need to know the password (and username if it isn't the default). You can do that via the config server URI, or via separate username and password properties, e.g.

bootstrap.yml.

```
spring:
  cloud:
    config:
     uri: https://user:secret@myconfig.mycompany.com
```

or

bootstrap.yml.

```
spring:
  cloud:
    config:
     uri: https://myconfig.mycompany.com
     username: user
     password: secret
```

The `spring.cloud.config.password` and `spring.cloud.config.username` values override anything that is provided in the URI.

If you deploy your apps on Cloud Foundry then the best way to provide the password is through service credentials, e.g. in the URI, since then it doesn't even need to be in a config file. An example which works locally and for a user-provided service on Cloud Foundry named "configserver":

bootstrap.yml.

```
spring:
  cloud:
    config:
     uri: ${vcap.services.configserver.credentials.uri:http://user:password@localhost:8888}
```

If you use another form of security you might need to provide a `RestTemplate` to the `ConfigServicePropertySourceLocator` (e.g. by grabbing it in the bootstrap context and injecting one).

7.6.1 Health Indicator

The Config Client supplies a Spring Boot Health Indicator that attempts to load configuration from Config Server. The health indicator can be disabled by setting `health.config.enabled=false`. The response is also cached for performance reasons. The default cache time to live is 5 minutes. To change that value set the `health.config.time-to-live` property (in milliseconds).

7.6.2 Providing A Custom RestTemplate

In some cases you might need to customize the requests made to the config server from the client. Typically this involves passing special `Authorization` headers to authenticate requests to the server. To provide a custom `RestTemplate` follow the steps below.

1. Create a new configuration bean with an implementation of `PropertySourceLocator`.

CustomConfigServiceBootstrapConfiguration.java.

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
    @Bean
    public ConfigServicePropertySourceLocator configServicePropertySourceLocator() {
        ConfigClientProperties clientProperties = configClientProperties();
       ConfigServicePropertySourceLocator configServicePropertySourceLocator =  new ConfigServicePropertySource
        configServicePropertySourceLocator.setRestTemplate(customRestTemplate(clientProperties));
        return configServicePropertySourceLocator;
    }
}
```

1. In `resources/META-INF` create a file called `spring.factories` and specify your custom configuration.

spring.factories.

```
org.springframework.cloud.bootstrap.BootstrapConfiguration = com.my.config.client.CustomConfigServiceBootstrapC
```

7.6.3 Vault

When using Vault as a backend to your config server the client will need to supply a token for the server to retrieve values from Vault. This token can be provided within the client by setting `spring.cloud.config.token` in `bootstrap.yml`.

bootstrap.yml.

```
spring:
  cloud:
    config:
      token: YourVaultToken
```

7.7 Vault

7.7.1 Nested Keys In Vault

Vault supports the ability to nest keys in a value stored in Vault. For example

```
echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' | vault write secret/myapp -
```

This command will write a JSON object to your Vault. To access these values in Spring you would use the traditional dot(.) annotation. For example

```
@Value("${appA.secret}")
String name = "World";
```

The above code would set the `name` variable to `appAsecret`.