Configuration Management

----------------------------------------------------------------------------------------------------------------

Level 400

Problem Statements:

1. In Spring boot application properties you set are unique to the application, it’s easy enough to specify them in application properties file. if you choose to package the properties inside the deployed JAR or WAR file, you must completely rebuild and redeploy the application should those properties need to change. These same constraints are in play should you need to roll back changes to configuration, which is problematic.
2. In microservice-architected applications, property management is spread across multiple codebases and deployment instances, making it unreasonable to apply the same change in every single instance of multiple services in a running application. Updating configuration properties and restarting all those instances manually or even with automated scripts may not be feasible.
3. Some properties are sensitive, such as database passwords and other types of secrets. Although those values can be encrypted when written to an individual application’s properties, the application must include the ability to decrypt those properties before they can be used. Even then, some properties may need to be kept from even the application developers, making it highly undesirable to set them in environment variables or manage them with the same source code control system as the rest of the application code.

Solution:

Spring Cloud Config Server provides centralized configuration with a server that all microservices within an application can rely on for their configuration. Because it’s centralized, it’s a one-stop shop for configuration that’s common across all services, but it’s also able to serve configuration that’s specific to a given service.

Most configuration served by Config Server may not be all that secret. But you might need Config Server to serve properties containing sensitive information such as pass- words or security tokens that are best kept secret in the backend repository.

Config Server offers two options for working with secret configuration properties:

1. Writing encrypted values in configuration files stored in Git
2. Using HashiCorp’s Vault as a backend store for Config Server in addition to (or in place of) Git

Let’s take a look at how each of these options can be used with Config Server to keep configuration properties secret. We’ll start with writing encrypted properties to the Git backend.

Cloud Config Working:

Diagram

Description automatically generated

Config Server exposes a REST API through which clients (other services) can consume configuration properties. We can configure the location of Spring Cloud Config server in our microservice bootstrap.properties so that it will load all the properties when we start the application. The configuration that’s served through the Config Server is housed external to the Config Server, typically in a source code control system such as Git, above image shows how this works. In addition to that, whenever we update the properties we can invoke  /actuator**/refresh** REST endpoint in our microservice so that it will reload the configuration changes without requiring to restart the application.

By storing the configuration in a source code control system such as Git, the configuration can be versioned, branched, labelled, reverted, and blamed, just like application source code. But by keeping the configuration separate from the applications that consume it, it can evolve and be versioned independently of those applications.

We also need to configure various sensitive data like database credentials, keys, tokens etc. Obviously, we don't want to store them in plain text. A better approach would be to store them in an encrypted format and Spring Cloud Config Server provides the ability to encrypt and decrypt the data. Even better we should use secure data storage tools like [**Vault**](https://www.vaultproject.io/). Spring Cloud also provides the integration with Vault so that we can store any sensitive configuration properties in Vault.

Enabling Config Server:

The easiest way to create a new Config server with spring Initializr. The most important thing to do is to specify the Config Server dependency by checking the Config Server check box. This will result in the following dependency being added to the produced project’s pom.xml file:

<dependency>

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

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

</dependency>

Although the Config Server dependency adds Spring Cloud to the project’s classpath, there’s no autoconfiguration to enable it, so you’ll need to annotate a configuration class with *@EnableConfigServer*. Following is an example

@EnableConfigServer

@SpringBootApplication

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

You need to do one more thing to get it started, you must tell application where the configuration properties that it’s to serve can be found. To start, you’ll use configuration that’s served from a Git repository, so you’ll need to set the spring.cloud.config.server.git.uri property with the URL of the configuration repository. Following is an example

spring:

cloud:

config:

server:

git:

uri: <https://github.com/sampleproject/sample-config>

There are several ways to set up properties for the Config Server to serve. The most basic, straightforward option is to commit an application.properties or application.yml or application-name.yml or application-name.properties file to the root path of the Git repository.

Consuming Configuration:

Spring Cloud Config Server also provides a client library that, when included in a Spring Boot application’s build, enables that application as a client of the Config Server. The easiest way to turn any Spring Boot application into a Config Server client is to add the following dependency to the project’s Maven build:

<dependency>

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

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

</dependency>

This same dependency is also available in the Spring Initializr as the check box labelled Config Client

When the application is started, autoconfiguration will automatically register a property source that draws its properties from a Config Server. By default, it assumes that the Config Server is running on localhost and listening on port 8888. But if that’s not the case, you can configure the location of the Config Server by setting the *spring.cloud.config.uri* property

spring:

cloud:

config:

uri: http://localhost:8888

Typically, you’ll only need to set *spring.cloud.config.uri* to specify the location of the configuration server and *spring.application.name* to identify the application to the configuration server.

When the application starts up, the property source provided by the Config Server cli- ent will make a request to the Config Server. Whatever properties it receives will be made available in the application’s environment. What’s more, those properties will be effectively cached; they’ll be available even if the Config Server goes down.

Serving Profile-Specific / application-specific properties:

When a Config Server client starts up, it makes a request to the Config Server with a request path that contains both the application name as well as the name of an active profile. When serving configuration data, Config Server will consider these values and return application-specific and profile-specific configuration to the client.

From a client perspective, consuming application-specific and profile-specific con- figuration properties isn’t much different than if you weren’t using Config Server. An application’s name is specified by setting the *spring.application.name* property and the active profile(s) can be specified by setting the *spring.profiles.active* property. Similarly, there’s not much that needs to be done in the Config Server itself to serve properties that target a specific application or profile.

Following are the examples file names with application name and profile that should be upload to git

Ex:

application-name-profile.yml or application-name-profile.yml

sampleapp-dev.properties or sampleapp-dev.yml

Storing configuration properties secrets:

Config Server offers two options for working with secret configuration properties:

1. Writing encrypted values in configuration files stored in Git
2. Using HashiCorp’s Vault as a backend store for Config Server in addition to (or in place of) Git

Refreshing Configuration properties on fly:

Spring Cloud Config Server supports the ability to refresh configuration properties of running applications with zero downtime. Once the changes have been pushed to the backing Git repository or Vault secret store, each microservice in the application can immediately be refreshed with the new configuration in one of two ways:

1. Manual—The Config Server client enables a special Actuator endpoint at /actuator/refresh. An HTTP POST request to that endpoint on each service will force the config client to retrieve the latest configuration from its backends.
2. Automatic—A commit hook in the Git repository can trigger a refresh on all services that are clients of the Config Server. This involves another Spring Cloud project called Spring Cloud Bus for communicating between the Config Server and its clients.

Each option has its pros and cons. Manual refresh gives more precise control over when services are updated with fresh configuration, but it requires an individual HTTP request to be issued to each instance of each microservice. Automatic refresh applies updated configuration instantly to all microservices in an application, but it’s ultimately triggered from a commit to the configuration repository

Summary

* Spring Cloud Config Server offers a centralized source of configuration data to all microservices that make up a larger microservice-architected application.
* The properties served by Config Server are maintained in a backend Git or Vault repository.
* In addition to global properties, which are exposed to all Config Server clients, Config Server can also serve profile-specific and application-specific properties.
* Sensitive properties can be kept secret by encrypting them in a backend Git repository or by storing them as secrets in a Vault backend.
* Config Server clients can be refreshed with new properties either manually via an Actuator endpoint or automatically with Spring Cloud Bus and Git webhooks.