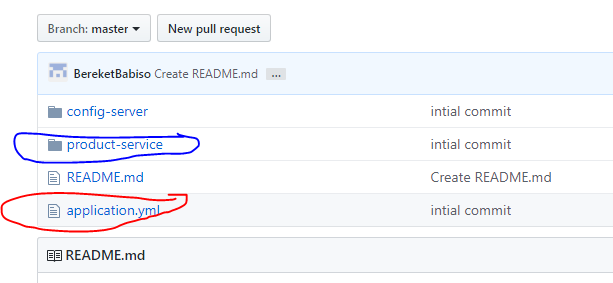
A Guide on Using PCF’s Cloud-Config-Server

Config server aims to separate and centralize sensitive microservice configuration files from the applications. This way it is possible to reuse properties and hide sensitive application properties as well. If you take, for instance, two applications pointing to the same database, then having database related configurations at the centralized config server is easier than having it in both applications. If later some database related configuration is required to be altered, then the change would be done only in one place. On the other hand, since database configuration is sensitive, keeping them only in one place plays crucial role in securing the configuration file.

Without any further ado, below is step-by-step guide to use the pcf cloud config-server as a service in a cloud-based springboot application.

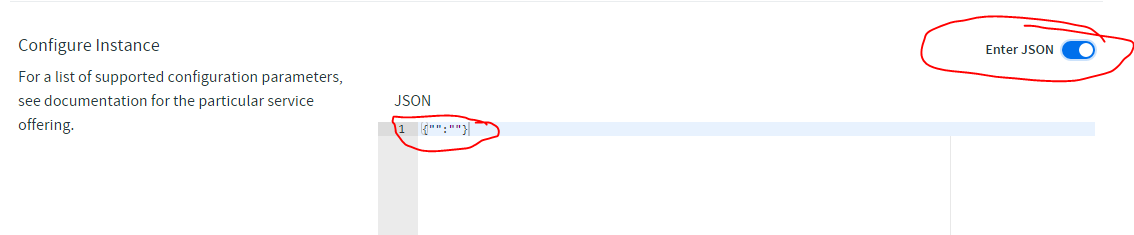
**Step 1:** create a git repository to store all the configuration files. For this tutorial, I created a git repository with the below structure

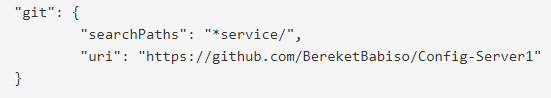


The product-service folder contains product service related configurations. The application.yml is a global configuration file for all the services that use this git store.

Note : <https://github.com/BereketBabiso/Config-Server1> (this is the git link to the store)

**Step2:** In the PCF space, go to the market place and get the config-server service. Give a name to the config-server (the name will later be used in the springboot manifest file to bind the application with the config-server). Then go to the “setting” of the config server and configure as below





The “searchPaths” helps us to inform the config server in which folder in the repository to look for the configuration files. For example, the above search path works for the “product-service” folder in the git repo. If we want to add new folder for customer service we can name it as “customer-service”.

This is pretty much it to create a cloud config server in pcf and bind it to a git repository store. Once the binding is done, click the manage icon of the config server and make sure that you get a result as below



**Step3:** Now create a simple gradle springboot application that uses configuration files from the git repository store through the pcf cloud config server we just created in step2. Give it a name PCFConfigServerConsumer and it should have the below gradle dependencies.

implementation 'org.springframework.boot:spring-boot-starter-web'

implementation 'io.pivotal.spring.cloud:spring-cloud-services-starter-config-client'

compile("org.springframework.boot:spring-boot-starter-actuator")

compile("org.springframework.boot:spring-boot-starter-security")

compile("org.springframework.security.oauth.boot:spring-security-oauth2-autoconfigure")

compile("org.springframework.security:spring-security-rsa")

testImplementation 'org.springframework.boot:spring-boot-starter-test'

Since we want the springboot application to get its configuration files from the config server, we should tell the container that it must get the configuration files first before starting the application we when run it. We can do this by adding a “bootstrap.yml” file in the resource directory. Since the config server uses the application name and the active profile to look for the right configuration files, the bootstrap.yml is required to have the application-name. Below is the bootstrap.yml of PCFConfigServerConsumer

spring:

application:

name: product

cloud:

config:

enabled: **true**

Then add a manifest file for the application to push it to the pcf space in the cloud. The manifest is required to have a config-server name.

---

applications:

- name: product

memory: 1G

instances: 1

path: build/libs/PCFConfigServerConsumer-0.0.1-SNAPSHOT.jar

buildpack: https://github.com/cloudfoundry/java-buildpack

services:

- config-server

env:

SPRING\_PROFILES\_ACTIVE : dev

#routes:

#- route:

random-route: **true**

In this RestApi we only have a rest controller that reads the configuration files from the git repository store through the config server and returns to the calling client. This is modeled in the “ConfigurationProperties”.

Note: since the spring io cloud config-client is secured, do not forget to extend WebSecurityConfigurerAdapter . This configuration is already shown in the application, use the below git link to find out the implementation of the application

<https://github.com/BereketBabiso/PCF-based-ConfigServer-Consumer>