

Spring Cloud Config



Spring Cloud Config



- Review Config Mgmt
- Config Server
- Config Client
- Spring Cloud Services

Config in a Spring Context

Configuration in a Spring context can usually be described as values that wire up Spring beans.

Spring has provided several approaches to setting config, including externalizing (via Command Line arguments, Env Variables, etc.)

Still, gaps exist:

- Changes to config require restarts
- No audit trail
- Config is de-centralized
- No support for sensitive information (no encryption capabilities)

Config in a 12 Factor Context

12 Factor application design states that configuration should be kept in OS environment variables.

In Pivotal Cloud Foundry, this is accomplished in the following ways:

- Using the **cf set-env** command
- Using a manifest with **env:** sections

This works well for applications with less demanding configuration needs.

Challenges When Using Env. Variables

Often times we want more capabilities because of the following:

- Managing many env variables can be a challenge
- Pivotal Cloud Foundry uses immutable containers.
 - Any configuration change requires restarting the app.
 - If you want zero downtime then blue/green deployments are necessary. This may be too much overhead.

More Demanding Config Use Cases

- Changing logging levels of a running application in order to debug a production issue
- Change the number of threads receiving messages from a message broker
- Report all configuration changes made to a production system to support regulatory audits
- Toggle features on/off in a running application
- Protect secrets (such as passwords) embedded in configuration

Externalized, Versioned, Distributed, Configuration

Q: How can these new use-cases be handled?

A: Externalize configuration to a service.

Configuration Mgmt approach should support:

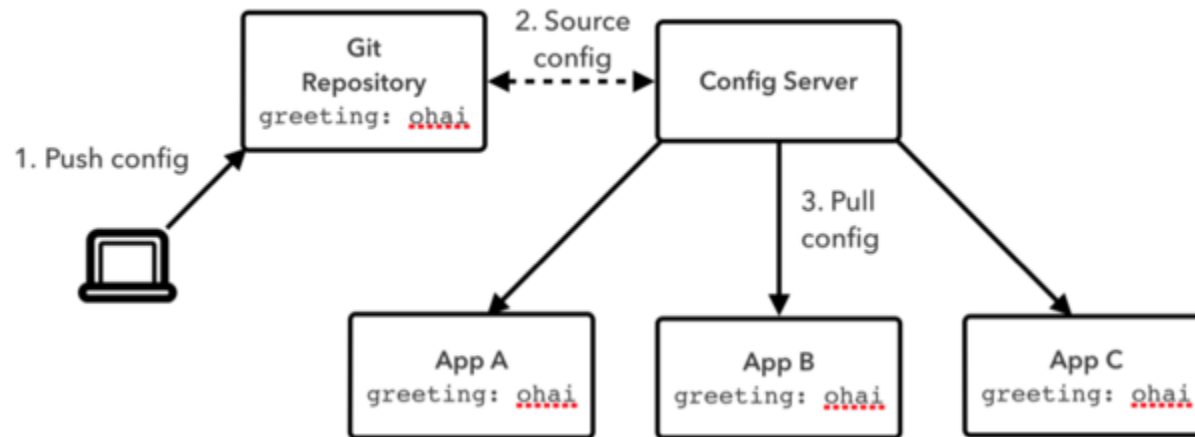
- Versioning
- Auditability
- Encryption
- Refresh without restart

Spring Cloud Config



- Review Config Mgmt
- Server
- Client
- Spring Cloud Services

Configuration Flow



Spring Cloud Config Server

- The Server provides an HTTP, resource-based API for external configuration (name-value pairs, or equivalent YAML content).
- Bind to the Config Server and initialize Spring **Environment** with remote property sources
- Embeddable easily in a Spring Boot application using **@EnableConfigServer**
- Encrypt and decrypt property values (symmetric or asymmetric)

Include Dependency

pom.xml

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-config-server</artifactId>  
</dependency>
```

Embedded Server

The server is easily embeddable in a Spring Boot application using the **@EnableConfigServer** annotation.

This app is a config server:

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

Server Config

- Application configuration data is stored in a backend
- Git, Subversion and File System backends are supported
- Git is the default backend. It's great for auditing changes and managing upgrades
- Setting the git backend is done via the **spring.cloud.config.server.git.uri** configuration property

Sample **application.yml**:

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

Server Endpoints

Config server exposes config on the following endpoints:

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

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

Configuration Files

What files do I put my configuration in?

Config files are stored in the git repository.

Example Input:

- `spring.application.name=foo`
- `spring.active.profiles=dev`

Union of all sources with overriding behavior.

Repo Files:

- `application.yml` - shared between all clients
- `application-dev.yml` - shared between all clients but profile specific (takes precedence over `application.yml`)
- `foo.yml` - app specific (takes precedence over `application.yml` files)
- `foo-dev.yml` - app and profile specific (takes precedence over `foo.yml`)

Spring Cloud Config



- Review Config Mgmt
- Server
- Client
- Spring Cloud Services

Client Application

Typical Spring Configuration without Spring Cloud Config:

```
@SpringBootApplication
@RestController
public class ClientConfigApplication {

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

    @Value("${greeting}") //<-- configuration injected from environment
    private String greeting;

    @RequestMapping("/greeting")
    String greeting() {
        return String.format("%s World", greeting);
    }
}
```

Include Dependency

pom.xml

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

Client Config

For Spring clients to use the configuration server, specify the **spring.cloud.config.uri** configuration property:

Sample **bootstrap.yml**

```
spring:
  cloud:
    config:
      uri: http://my-config-server.io/
```

spring.cloud.config.uri defaults to <http://localhost:8888>

Refreshable Application Components

What can be refreshed in a Spring Application?

- Loggers `logging.level.*`
- `@ConfigurationProperties` beans
- Beans with `@RefreshScope` annotation

@RefreshScope

Annotate bean with **@RefreshScope**:

```
@SpringBootApplication
@RestController
@RefreshScope // <-- Add RefreshScope annotation
public class ClientApplication {

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

    @Value("${greeting}")
    private String greeting;

    @RequestMapping("/greeting")
    String greeting() {
        return String.format("%s World", greeting);
    }

}
```

Refreshing Configuration

Two step process for applications getting new configuration:

1) Update Repository

2) Send a request to the application(s) to pick up new config values

Send a **POST** request to the refresh endpoint in the client app to fetch updated config values:

http://127.0.0.1:8080/refresh

Requires the **Actuator** dependency on the classpath (**pom.xml**).

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Spring Cloud Bus

When running many applications, refreshing each one can be cumbersome.

Instead, leverage Spring Cloud Bus pub/sub notification with RabbitMQ.

Send a POST request to the refresh endpoint to fetch updated config values:

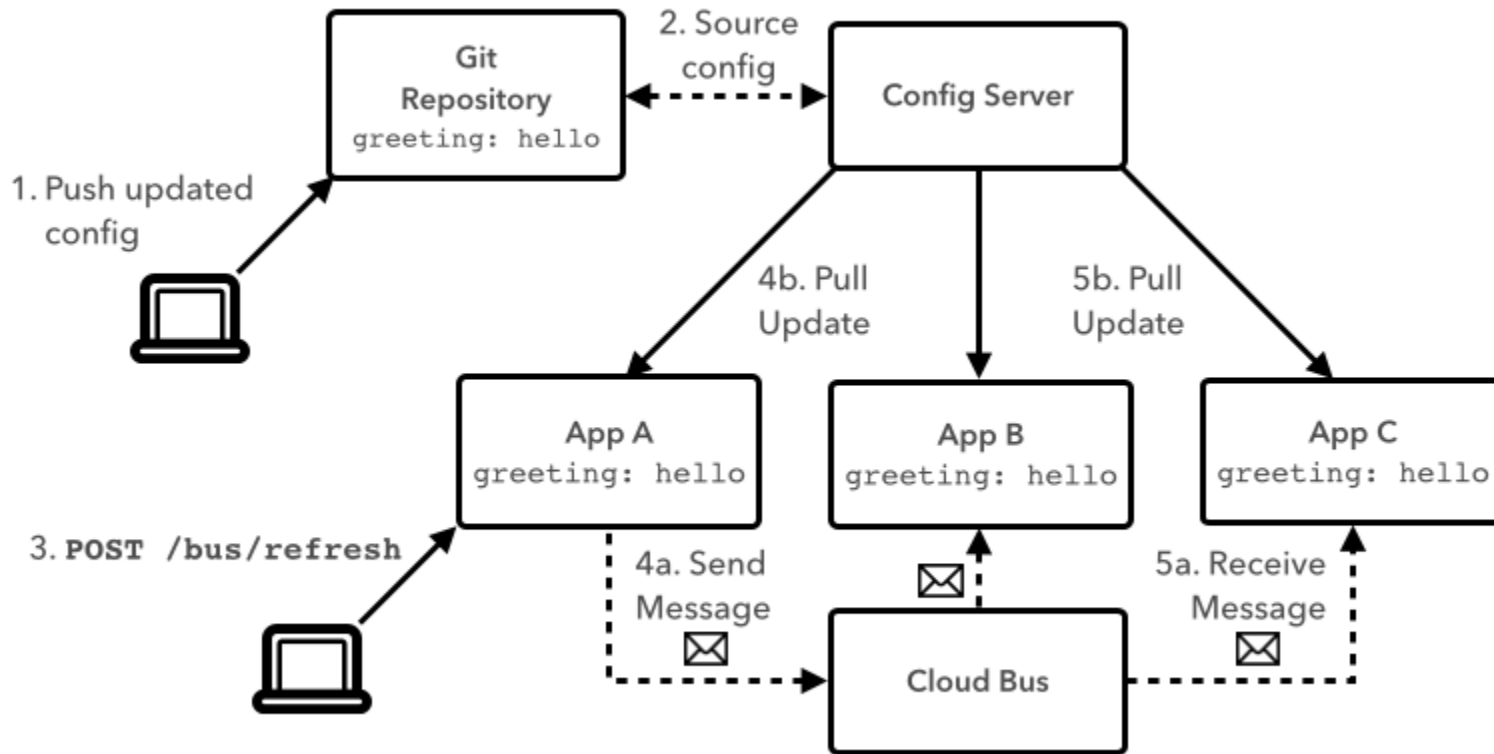
`http://127.0.0.1:8080/bus/refresh`

Requires the **Cloud Bus AMQP** dependency on the classpath (**pom.xml**).

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```



Cloud Bus Diagram



Spring Cloud Config



- Review Config Mgmt
- Server
- Client
- Spring Cloud Services

Spring Cloud Services

- Brings Spring Cloud to Pivotal Cloud Foundry
- Includes: Config Server, Service Registry & Circuit Breaker Dashboard services



Spring Cloud Services: Config Server

1) Include dependency:

pom.xml


```
<dependency>
  <groupId>io.pivotal.spring.cloud</groupId>
  <artifactId>spring-cloud-services-starter-config-client</artifactId>
</dependency>
```

2) Create a Config Server service instance

3) Configure the service instance with a Git Repository

4) Bind the service to the app

Spring Cloud Services: Config Server

 **Spring Cloud Services** for Pivotal Cloud Foundry

Config Server

Instance ID: b2620e4e-6339-4dfd-a7a8-76656d576616

Configuration Source

☒ Git
☐ Subversion

Git URI

Git Branch (default is 'master')

Submit

Pivotal © 2015 Pivotal Software Inc. All rights reserved.

Cloud Bus in Pivotal Cloud Foundry

1) Include dependency:

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

2) Create a RabbitMQ service instance

3) Bind the service to the app